MEASURING AND IMPROVING THE RUNTIME BEHAVIOUR OF ASPECTJ PROGRAMS

by
Christopher Goard

School of Computer Science
McGill University, Montréal

August 2005

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Abstract

AspectJ is a popular aspect-oriented extension to Java, providing powerful new features for the modularizing of crosscutting concerns, promising improved code quality. The runtime cost of these features, however, is currently not well understood, and is a concern limiting even more wide-spread adoption of the language. The crosscutting nature of AspectJ complicates the measurement of these costs.

This thesis presents a methodology for analyzing the runtime behaviour of AspectJ programs, with a particular emphasis on identifying runtime overheads resulting from the implementation of AspectJ features. It presents a taxonomy of overhead kinds and defines some new AspectJ-specific dynamic metrics. A toolset for measuring these metrics is described, including both of the current AspectJ compilers: `ajc` and `abc`, and results for a newly collected set of AspectJ benchmarks are presented.

Significant overheads are found in some cases, suggesting improvements to the code generation strategy of the AspectJ compilers. Initial implementations of some improvements are presented, resulting, for some benchmarks, in order of magnitude improvements to execution time. These improvements have since been integrated in `abc` and `ajc`.

Clearly understanding the runtime behaviour of AspectJ programs should result in both better implementations of the language and more confident adoption by the mainstream.

Résumé

AspectJ est une populaire extension orientée aspect pour Java, offrant de nouveaux outils puissants pour la modularisation des préoccupations transverses, promettant une qualité de code source améliorée. Le coût d'exécution de ces outils n'est pas encore bien compris, ce qui limite l'adoption à grande échelle du language. La nature transverse d'AspectJ complique la mesure de ces coûts.

Ce mémoire présente une méthode permettant d'analyzer l'opération des programmes AspectJ, avec une emphase particulière sur l'identification des coûts d'exécution résultants de l'implémentation des fonctionalités d'AspectJ. Il présente une taxonomie des types de coûts d'exécution et défini un ensemble de nouvelles mesures dynamiques spécifiques à AspectJ. D'es outils pour obtenir ces mesures sont décrits, incluant les compilateurs AspectJ actuels : `ajc` et `abc`. Des résultats pour un ensemble nouvellement assemblé de programmes-étalons son presentés.

Des coûts d'exécution significatifs ont étés trouvés dans certains cas, suggérant des améliorations à la stratégie de génération de code des compilateurs AspectJ. Des implémentations initiales de certaines améliorations sont présentées, résultant, pour certains programmes-étalons, en une augmentation d'un ordre de grandeur de la performance. Ces améliorations ont depuis étées intégrées aux compilateurs `abc` et `ajc`.

Bien comprendre le comportement à l'exécution des programmes AspectJ devrait résulter en de meilleures implémentations du language et une meilleure confiance en son adoption de la part de la communauté des développeurs en général.

Acknowledgements

In completing this thesis, I owe a debt of gratitude to many people. First and foremost amongst whom is my supervisor, Laurie Hendren, whom I thank for her guidance, support, and encouragement. Bruno Dufour's tireless work on **J* deserves special mention, as does the work of my other co-authors on the OOPSLA'04 paper: Clark Verbrugge, Oege de Moor, and Ganesh Sittampalam. Maxime Chevalier-Boisvert and François Villeneuve provided invaluable assistance in translating the abstract. Additional thanks are due to members of the Sable Lab; in particular, I wish to thank Ondřej and Jennifer Lhoták, Nomair Naeem, Ahmer Ahmedani, Grzegorz Prokopski, Chris Picket, Sokhom Pheng, Dayong Gu, and Navindra Umanee. Place Milton and McGill Pizza were the providors of countless greasy breakfasts, and Boustan of many late-night dinners, fueling the writing this thesis. Thanks to the many people at GAMMA for providing regular respite from this ordeal, especially to my other mentor during my time in Montreal, Philip Gelinas. Finally, thanks to all my friends and family for putting up with me over the course of this endeavour.

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1
Introduction

1.1 Motivation

Aspect-oriented programming [KLM+97] shows much promise as an extension to contemporary object-oriented modes of software construction. Of the various implementations of aspect-oriented ideas, AspectJ [KHH+01b] is the most popular, both within industry and within academia. Since its introduction, it has seen a large and active community of developers and researchers grow up around it and is now on the verge of genuine mainstream success and deployment in production environments.

Much thought has been devoted to the ways in which the AspectJ language—and in particular its join point model of pointcuts and advice—can improve the modularity and quality of source code, beyond what is possible with pure Java; an end goal of AspectJ, and of aspect-oriented programming in general, being the reduction of development costs for complex systems. Until recently, however, very little work has been done on examining the runtime efficiency of AspectJ implementations. The corresponding runtime costs of these improvements has remained unknown.

The AspectJ language has matured to the point that examining the runtime behaviour of compiled code has become pertinent. Assessing and establishing the runtime efficiency of compiled AspectJ code, in comparison with its Java equivalents,

may be necessary before the language sees further adoption by the mainstream for production systems. The very nature of AspectJ, however, in particular its use of bytecode weaving to implement the join point model, impedes this examination. The importance of runtime efficiency, and the difficulty of measuring it, are both indicated in the AspectJ FAQ [Xer03]:

> *The issue of performance overhead is an important one. It is also quite subtle, since knowing what to measure is at least as important as knowing how to measure it, and neither is always apparent.*
>
> *We aim for the performance of our implementation of AspectJ to be on par with the same functionality hand-coded in Java. Anything significantly less should be considered a bug.*

In order to perform this examination, a representative set of AspectJ benchmarks is required. Unfortunately, as affirmed by the FAQ, no such benchmark suite exists.

The analysis of the runtime behaviour of AspectJ, and the assembly of the requisite AspectJ benchmark suite, are of current importance for the continued growth of AspectJ, and consequent validation of aspect-oriented notions of software development.

This thesis presents a framework for analyzing the dynamic behaviour of AspectJ programs, and identifies some of the runtime costs incurred by the language.

1.2 Contributions

The specific contributions of this thesis are as follows:

- The two current AspectJ compilers, `ajc` and `abc`, have been augmented to annotate the generated class files with additional metadata that enables a variety of measurements and analyses to be performed on the generated bytecode and its execution.
- A set of AspectJ benchmarks has been collected from various public sources and assembled into a benchmark suite.

- A new set of of AspectJ-specific dynamic metrics, to explain the runtime behaviour of AspectJ programs, and in particular to identify and account for runtime overhead induced by AspectJ language features, has been defined and implemented in the **J* [DDHV03] dynamic metrics framework.
- A taxonomy of runtime overhead kinds, consonant with the division of AspectJ language features, has been defined.
- Contrary to conventional wisdom concerning AspectJ, some significant runtime overheads have been found for certain benchmarks. The language features and usage patterns resulting in these overheads are identified and explained.
- Improvements to the code generation strategy of `ajc`, which reduce the identified runtime overheads, are presented and implemented. Comparisons are made to the stock version of `ajc`. The ideas behind these improvements have since been incorporated in recent release versions of `ajc`. `abc`'s development has been informed by early versions of this work, and incorporates these and other optimizations. Comparisons between these compiler versions are made.

These contributions should be of direct value to both AspectJ users and AspectJ compiler writers. AspectJ users should benefit from these contributions as they provide guidance as to what language features and idioms may impose performance penalties. Conversely, they also allow users to apply other features and idioms with the confidence that significant performance penalties are not being incurred. Compiler writers can benefit from this work as it provides a means of identifying future improvements to the language's compilers. It suggests improved code generation strategies and improved static analyses that should result in more efficient bytecode.

1.3 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 is a brief introduction to the AspectJ language, and the AspectJ concepts required to understand the rest of this work. Those already conversant with the language can safely skip this chapter. Chapter 3 describes the dynamic measurements that are made on the AspectJ benchmark programs, including the definition of some new AspectJ-specific metrics. Chapter 4 presents the toolset that was collected, written, and assembled to perform these measurements. Chapter 5 describes the categorization of AspectJ overheads, and the metadata that is required of the AspectJ-specific dynamic metrics and that is attached to classfiles produced by the augmented compilers. Chapter 6 defines the dynamic algorithms required of the metric analyses. Chapter 7 presents the experimental results. The benchmarks measured are described, and the measurements collected are analyzed. Comparisons are made between compiler implementations, both between `ajc` and `abc`, and between `ajc` and a modified version of `ajc` that implements some compiler optimizations presented in this chapter to reduce some of the measured overheads. Chapter 8 is a survey of related work and chapter 9 concludes this work and suggests some avenues for future work.

Chapter 2
AspectJ

This chapter provides a brief introduction to aspect-oriented programming and the AspectJ language. Section 2.1 describes aspect-oriented programming in general. It describes the problems aspect-oriented programming intends to solve, and the basic strategy with which it attempts to do so. Section 2.2 provides a brief introduction to the AspectJ language, focusing on those features most relevant to this thesis. Section 2.3 presents a larger example of an aspect that incorporates many of the features described in section 2.2 and which illutrates the value of aspect-oriented programming and AspectJ.

2.1 Aspect-Oriented Programming

A software system, in general, is composed of multiple concerns. A concern is any design-level notion—such as a feature or a requirement—that results in implementation at the source code level. In most software systems, it is very desirable to achieve good separation of concerns. A system exhibits good separation of concerns when, for each concern in a system, there is a direct correspondance between the design-level idea and the implementation-level module expressing it. Ideally, a single concern should be implemented in a single implementation unit, and a single implementation unit should implement a single concern. This is related to the concepts of cohesion and coupling. A system that shows good separation of concerns

should show high cohesion and loose coupling amongst its modules.

Separation of concerns is a desirable property because a system that exhibits it will, in general, be easier to read, understand, maintain, and evolve. Making these qualities easier to achieve allows for the development of larger, more complex systems at lower costs. The desires to achieve improved code quality, and thus reduced development costs, have motivated the evolution of programming paradigms. Each new paradigm has provided the programmer with new tools with which to further abstract and modularize the concerns being implemented.

The object-oriented paradigm, although providing the developer with many techniques for improving code quality, is unable to achieve complete separation of concerns. The constructs it provides for modularizing concerns—classes, objects, and methods—are insufficient, as the implementations of many concerns end up cutting across their boundaries. This results in the scattering of implementations across multiple modules and the tangling of implementations within single modules, both detrimental to code quality. These concerns, whose implementations span object-oriented modular units, are called *crosscutting concerns* [KLM+97].

Typically, the core concerns, or domain logic, of a well-written object-oriented system are well-modularized. It is concerns such as logging, authentication, authorization, persistence, transactional integrity, and so forth, that tend to be crosscutting in nature. Crosscutting concerns are not just an artifact of poorly-factored code: even the best-designed and implemented programs may have crosscutting concerns, and refactoring the code to modularize them will result in the scattering and tangling of previously well modularized concerns.

Aspect-oriented programming is an extension of the object-oriented paradigm that provides new constructs for the modularization of crosscutting concerns. It provides new means for specifying concerns separately, and for composing them together to produce a whole program. By achieving a more direct correspondence between design-level and implementation-level constructs, it promises improved code quality with a consequent reduction in the cost of designing, developing, and maintaining complex software systems.

2.2 AspectJ

AspectJ [KHH+01b] is an aspect-oriented extension of the Java programming language. It resulted from research into aspect-oriented programming at Xerox Parc in the 80s and 90s [KLM+97] and saw its first release in 1998. It is now being developed as part of the Eclipse project [Asp]. Of the several different implementations of aspect-oriented ideas, AspectJ is, as of this writing, by far the most popular, both in industry and in academia.

One of the goals of the AspectJ project is for it to function as a large scale software engineering experiment to validate the ideas of aspect-oriented programming in real-world contexts. Consequently, its design has been driven by the desire to develop a large and active developer community by making the language easy to learn for current Java programmers and by making it easy to incorporate elements of AspectJ into extant Java systems. As such, AspectJ is a strict extension to Java: every valid Java program is a valid AspectJ program.[1] Furthermore, AspectJ compiles to normal Java bytecode that can be executed in a standard JVM, not requiring a specialized runtime environment.

AspectJ extends Java with a new top-level construct: the aspect. The aspect is AspectJ's unit of modularization for crosscutting concerns. A concern whose implementation, in Java, was inevitably scattered across multiple classes or methods, entangled with the implementations of other concerns, should, in AspectJ, be neatly encapsulated within an aspect.

AspectJ is an asymmetric aspect-oriented language [HOT02] in that it distinguishes between core and crosscutting concerns, specifying them differently. Core concerns continue to be implemented in pure Java, modularized within classes and methods. Their implementation is referred to as the *base program*. Crosscutting concerns are implemented in aspects, using an extended syntax of Java. The aspects and base program are composed together to produce the complete program.

[1] In some implementations of AspectJ there are some exceptions due to the introduction of several new keywords to Java. Java programs that use these keywords as identifiers are not valid AspectJ programs.

The features AspectJ provides for implementing crosscutting concerns in aspects can be classified into two groups: dynamic crosscutting features and static crosscutting features. The dynamic crosscutting features are those that implement crosscutting concerns by modifying the runtime behaviour of a program; static crosscutting features modify the static type structure of a program. The following sections will provide a brief introduction to these AspectJ features.

2.2.1 Dynamic Crosscutting

An aspect is analagous to a class in many ways. Like a class, it can have methods and fields. It can extend another class or aspect and can itself be extended. It can be concrete or abstract. An aspect, however, may also contain several special AspectJ constructs: pointcuts, advice, and intertype declarations. The first two implement dynamic crosscutting, and is discussed in this section; the latter implements static crosscutting and is discussed in the next section.

The dynamic crosscutting features of AspectJ are those that implement crosscutting concerns by means of modifying the dynamic behaviour of the program. The nature of these features can be illustrated by analogy to the observer pattern. Conceptually, an aspect may be considered an observer, with the execution of the whole program the subject. The aspect observes the execution of the whole program, and at particular points within the execution, modifies the behaviour of the program by executing new code. The points at which new code can be injected are called *join points*, and the code that is injected is called *advice*. A *pointcut* is a pattern that selects join points of interest, and every piece of advice has an associated pointcut.

To actually implement AspectJ in this fashion would be terribly inefficient. It would also require special VM support (which would conflict with AspectJ's goal of easy adoption by Java developers). Instead of a literal implementation of aspects as observers, aspects and base program are composed statically in a form of partial evaluation [MKD03]. This is known as *weaving*.

A *join point shadow* is the static counterpart of a join point. Or, equivalently, a join point is a particualar execution of a join point shadow. The weaver inserts

instructions at join point shadows to execute the advice that would apply to the corresponding join points. Since a single join point shadow may correspond to an arbitrary number of join points, and since not all of these join points may be matched by a particular pointcut, the weaver often needs to add a runtime check to the code inserted at the join point shadow. This is known as a *dynamic residue*. If the dynamic residue specifically tests the applicability of advice at a given join point, it is called an *advice guard*.

Figure 2.1 is a high-level illustration of this process. The base program, which implements core concerns in Java, and the aspect, which implements crosscutting concerns, are specified separately. The weaver composes the aspect and the base program, resulting in a final program with advice woven into and across the modular units of the base program. The final program is equivalent to what could have been produced with Java were one willing to accept the scattered implementation of the functionality captured in the aspect.

A simple example of AspectJ source code is shown in Listing 2.1. It illustrates several basic AspectJ features, and will be referred to later in this section.

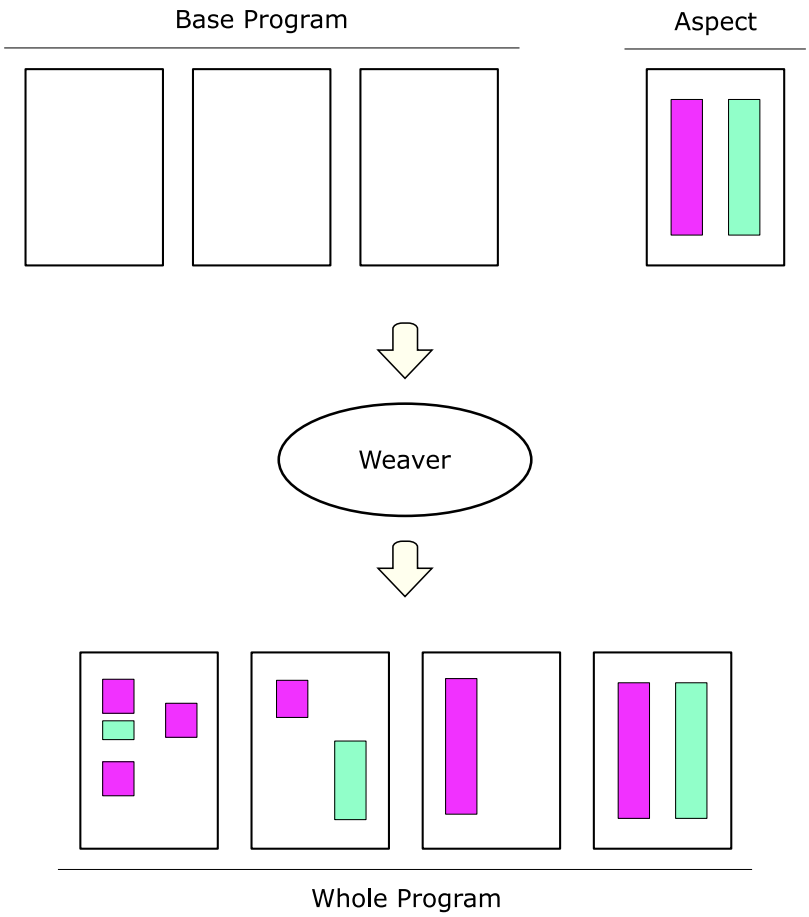


Figure 2.1: *Weaving of base program and aspect*

```
public class Example {

  public static void main(String[] args) {
    Example e = new Example();
    e.bar();
    e.foo();
  }

  public void foo() {
    System.out.println("foo");
    bar();
  }

  public void bar() {
    System.out.println("bar");
  }
}

aspect ExampleAspect {

  pointcut barInFoo(): call(void Example.bar())
                  && cflow(call(void Example.foo()));

  before(): barInFoo()
  {
    System.out.println("foo->bar");
  }
}
```

Listing 2.1: *Example AspectJ program with* **cflow**

Join Points

Join points are the most fundamental of the concepts AspectJ adds to Java. A join point is a particular point in the execution of a program, a specific runtime event. An aspect-oriented language's *join point model* defines what runtime events are exposed as join points. In AspectJ's case, the following events are exposed as join points:

- method call and execution
- constructor call and execution
- field get and set
- class initialization
- object initialization and pre-initialization
- exception handling
- advice execution

Not every possible join point is exposed. These particular events have been chosen because they are relatively stable in the face of compiler optimizations and some code refactorings. Other potential join points, such as entry into a loop or other control flow structure [HG05], are much more volatile in the face of such code transformations and so are not exposed.

It is important to realize that a join point is not an atomic point, but rather a region of execution. A join point has a beginning, it has an end, and it can contain other join points. Figure 2.2 is an annotated UML sequence diagram illustrating this point with some example join points. The `foo` method execution join point is contained by the corresponding call join point, and all of the join points are contained by that for the execution of `main`.

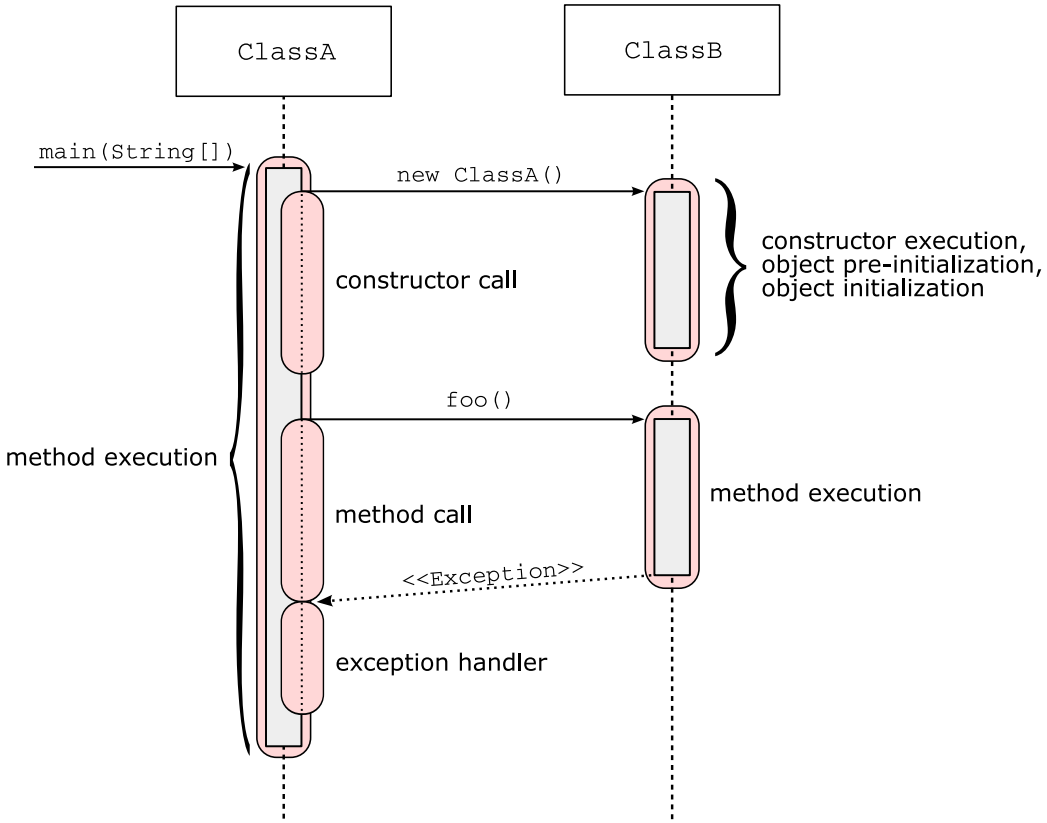


Figure 2.2: *Several kinds of join point*

Pointcuts

A pointcut is a pattern that matches join points. A pointcut may also specify some context that should be exposed to advice at a join point—the target object or the arguments of a method call join point, for example.

Pointcuts are specified by the programmer in the pointcut definition language, whose syntax is distinct from that for the rest of AspectJ. A pointcut is either a primitive pointcut or a compound expression composed of other pointcuts and boolean operators.

Primitive pointcuts can be classified into three groups: those that match join points by their kind; those that match join points based on their static context;

and those that match join points based on their dynamic context. The first two groups can be matched statically, while matching of the third may require dynamic residues.

Matching by kind: Each kind of join point listed above (method call, method execution, field get, *etc.*) has an associated primitive pointcut that selects join points of that kind. Most of these pointcuts also take as argument a pattern that matches type or signature. For example, `call(* foo*())` would select all method call join points for which the method signature matches the given pattern (no parameters, name starts with "foo".)

Matching by static context: The **within** and **withincode** pointcuts match join points based on static context: if a join point's shadow is lexically located within a type or method matching the given pattern, it matches the pointcut.

Matching by dynamic context: The **cflow** pointcut takes as argument another pointcut. If a join point is executing within the dynamic context of any join point matching the argument pointcut, it matches.

For example, consider the program in Listing 2.1. The pointcut is `call(void Example.bar()) && cflow(call(void Example.foo()))`. The **cflow** fragment matches all join points within the dynamic context of a call to `foo()`—that is, all join points for which a call to `foo()` exists on the call stack. This includes the call to `foo()` itself. The whole thing selects calls to `bar()` that occur below a call to `foo()`.[2]

The **cflowbelow** pointcut differs in that it would not match the call to `foo()` itself, unless it was a recursive call.

target and **this** pointcuts match join points based on the runtime types of the **target** and **this** objects, respectively. They can also be used to expose these

[2] In this simple example, **withincode** could be used instead to achieve the same semantics with less runtime overhead, as it is a static pointcut that won't require a dynamic residue. See section 2.3 for a more complex example with a use of **cflow** that cannot be replaced by **withincode**.

objects to advice. The **args** pointcut is similar, matching on and exposing the arguments at a join point.

The **if** pointcut can contain a boolean expression that may access any static data in the running program. If it evaluates to true for a join point, then that join point matches.

Advice

Advice is the construct that defines crosscutting behaviour. One way to think of it is as the scattered implementation of a crosscutting concern extracted horizontally from a system and packaged into a unit that is very much like a method. Equivalently, advice is the code that is inserted into an executing program at particular join points. Every advice declaration in an aspect is associated with a pointcut identifying the join points at which it should be executed.

There are several kinds of advice:

- before advice
- after returning advice
- after throwing advice
- after advice
- around advice

before and **after** advice execute before and after the advised join points. **after** advice is executed regardless of how a join point is exited, whether normally or by exception. Specialized kinds of **after** advice, **after returning** and **after throwing**, will execute only after a normal return or only after returning by exception, respectively.

around advice might more easily be understood as “instead-of advice”. It executes in place of the original join point, with the option to execute the original join point any number of times from within the advice body. The **proceed** keyword in

an **around** advice body indicates that the original join point should be executed at that point.

Advice bodies have access to reflective information about the join points they advise. The keywords **thisJoinPoint** and **thisJoinPointStaticPart** and **thisEnclosingJoinPointStaticPart** each return objects containing this reflective information.

Aspects

Aspects, as described at the beginning of this chapter, are the basic modular units of crosscutting concerns, and are very similar to classes in many ways. In addition to the members a normal Java class can contain, an aspect can contain advice and pointcut declarations. For example, the aspect in Listing 2.1 declares a single named pointcut and a single piece of **before** advice that is associated with that pointcut.

Like classes, aspects are instantiated as objects. By default, an aspect is a singleton. Any fields used by advice defined in the aspect are shared by all executions of the advice, at all join points. Aspect instances, however, can also be associated on a per-object and a per-cflow basis. An aspect can be declared to be **perthis** or **pertarget**, in which case an instance will be associated with each **this** or **target** object at join points matching a given pointcut. An aspect can also be declared **percflow**, in which case an instance is associated with each matching control flow pattern.

In addition to pointcuts and advice, aspects can also contain intertype declarations which modify the static structure of a program. These are explained in the following subsection.

2.2.2 Static Crosscutting

The static crosscutting features of AspectJ are those that implement crosscutting concerns by modifying the static type structure of a program. An aspect can do this by introducing new members—fields, methods, or constructors—to a class or interface. It can also declare new parents for any class or interface, making it extend from a new supertype or implement a new interface. These features are also called *intertype declarations.*

For example, consider a class `C` which extends a class `A`. If class `B` also extends class `A`, then an aspect may declare `B` to be the new superclass of `C` with the statement `declare parents: C extends B;`.

An aspect can also perform *exception softening*, which is the conversion of checked exceptions to unchecked exceptions. The `declare soft` statement takes two arguments: the type of a checked exception and a pointcut. At all join points matching the pointcut, any checked exceptions of the given type are caught, wrapped in an unchecked exception of type `org.aspectj.SoftException`, and rethrown.

2.3 An AspectJ Example

This section presents a larger example of AspectJ, which makes use of a number of the features described in the previous sections. It has been taken from Ramnivas Laddad's book, *AspectJ in Action* [Lad03], pages 346–350. It shows the implementation, as an abstract aspect, of a reusable protocol for authentication and authorization based upon the Java Authentication and Authorization Service (JAAS) [Sun], and the specialization of this protocol with a small concrete aspect for a particular application. Authentication and authorization are concerns whose implementations are typically scattered across an application, intruding into core domain logic. They demonstrate clearly the potential improvements AspectJ can make to code quality.

The abstract aspect in Listing 2.2 defines the basic protocol for both authentication and authorization. It is intended to be extended by a concrete aspect, which defines the pointcut identifying operations requiring authorization (`authOperations`) and the function which codifies the authorization policy (`getPermission`). An example concrete aspect specializing this protocol for a simple banking application is shown in Listing 2.3.

The first piece of **before** advice in the abstract aspect performs authentication. If authentication has not yet been performed, the `_authenticatedSubject` field will be null, and an authentication function will be called. If it succeeds, a `Subject` representing the authenticated user will be assigned to the field.

Authentication is performed by the `authenticate()` function using a JAAS `LoginContext` object. This object takes two parameters: a configuration name ("Sample") and callback function (`TextCallBackHandler2()`). The callback function acquires and returns the authentication data (e.g. user name and password) and the configuration name selects the authentication policy, which is defined externally. When authentication succeeds, `_authenticatedSubject` is set; when it fails, a `LoginException` is thrown.

By using an aspect like this, just-in-time authentication can be implemented without having to intrude upon the domain logic of the application.

Once authentication has been performed, actions must be authorized. Again, the actions requiring authorization are specified by the pointcut. The permissions required to execute each action are defined by the `getPermission` function. This function takes as argument a `JoinPoint.StaticPart` object which provides reflective information about the advised join points (method name, for example), which can be used to differentiate actions requiring different permissions. The concrete aspect in Listing 2.3 shows an example of specifying the `getPermission` method for a particular application.

The next two pieces of advice implement the authorization checks. The **around** advice executes first, and when its **proceed** statement (representing the actions requiring authorization, and here wrapped in a JAAS action object) is executed, so is the second piece of **before** advice.

It is possible for one action requiring authorization to be called from another. The **before** advice checks permissions for each, but only the root action needs to be called via `Subject.doAsPrivileged` by the **around** advice. The additional **cflowbelow** pointcut on the **around** advice excludes all actions occuring within another authorized action, thus eliminating unnecessary checks.

Understanding the details of the JAAS implementation in this example is not vital to understanding the value provided by AspectJ. In brief, the **around** advice executes an action requiring authorization on behalf of an authenticated subject, and the **before** advice checks that the subject has the sufficient permission to execute that particular action, throwing an exception if it doesn't.

JAAS allows for the authentication and authorization policies of a system to be defined external to the program, simplifying the implementation of access control. In a standard Java implementation, however, the calls to check access would still be scattered throughout the program. This is especially undesirable for a security concern; if a developer forgets to add an authorization check in accordance with some policy, the security of the whole system could be compromised. AspectJ, however, complements the benefits provided by JAAS by modularizing the implementation of access control and centralizing the implementation of a security policy.

This particular example illustrates another benefit of the increased separation of concerns made possible by AspectJ: a developer who understands the domain logic of the application may not be an expert in security, and an expert in security may not understand the domain logic of the application. By separating the two, each concern can be developed by people with the appropriate expertise.

```
public abstract aspect AbstractAuthAspect {
   private Subject _authenticatedSubject;
   public abstract pointcut authOperations();

   before() : authOperations() {
      if(_authenticatedSubject != null) {
         return;
      }
      try {
         authenticate();
      } catch (LoginException ex) {
         throw new AuthenticationException(ex);
      }
   }

   public abstract Permission getPermission(
                     JoinPoint.StaticPart joinPointStaticPart);

   Object around()
      : authOperations() && !cflowbelow(authOperations()) {
```

```
      try {
         return Subject
            .doAsPrivileged(_authenticatedSubject,
                        new PrivilegedExceptionAction() {
                              public Object run() throws Exception {
                                 return proceed();
                              }}, null);
      } catch (PrivilegedActionException ex) {
         throw new AuthorizationException(ex.getException());
      }
   }

   before() : authOperations() {
      AccessController.checkPermission(
                  getPermission(thisJoinPointStaticPart));
   }

   private void authenticate() throws LoginException {
      LoginContext lc = new LoginContext("Sample",
                                 new TextCallbackHandler2());
      lc.login();
      _authenticatedSubject = lc.getSubject();
   }
}
```

Listing 2.2: *An abstract aspect defining an authentication and authorization protocol*

```
public aspect BankingAuthAspect extends AbstractAuthAspect {
   public pointcut authOperations()
      : execution(public * banking.Account.*(..))
      || execution(public * banking.InterAccountTransferSystem.*(..));

   public Permission getPermission(
               JoinPoint.StaticPart joinPointStaticPart) {
      return new BankingPermission(
               joinPointStaticPart.getSignature().getName());
   }
}
```

Listing 2.3: *A concrete instance of the abstract protocol defined in Listing 2.2*

Chapter 3
Metrics

As explained in chapter 1, the runtime cost of AspectJ's features has remained largely unknown, although it has generally been assumed to be negligible. However, the manner in which aspects and base program are composed statically by the weaver, as described in chapter 2, suggests that some runtime overhead should be present, at least in the form of dynamic residues. This chapter presents the key measurements used in this work to assess this belief, providing a quantitative means either to confirm that overhead is negligible or to identify its nature and significance.

These measurements can be grouped into three categories: execution time, Java-based dynamic metrics, and AspectJ-specific dynamic metrics. They are briefly introduced below in this order. The AspectJ metrics are the most significant contribution, and they, in particular, are considered in greater detail in subsequent chapters.

3.1 Execution Time

Execution time is the most coarse-grained measurement made, but also the most telling: the significance of runtime overhead is proportional to its impact on total execution time. Execution time comparisons are made between several variations on a benchmark, including:

- Between an AspectJ benchmark and a Java version of equivalent functionality. This measurement should indicate the presence of AspectJ overhead.
- Between a complete AspectJ program and its base (Java) program. This should indicate whether a benchmark's execution is aspect-heavy, or dominated by its base code.
- Between a benchmark's total execution time, the time spent in garbage collection, and the time spent in the JIT compiler.
- Between versions of a benchmark that differ slightly in implementation of the aspect, in particular in the definition of pointcuts. This can identify costly usage patterns if small changes result in large execution time differences.
- Between instances of a benchmark compiled with different compilers and compiler configurations. `ajc`, `abc`, and a version of `ajc` modified to include some simple optimizations are used. Furthermore, `abc` is used with different optimizations enabled.

3.2 Dynamic Metrics

Comparisons of execution times, while capable of identifying the existence of performance problems in generated code and of evaluating the effectiveness of improved code generation strategies, cannot identify what particular AspectJ features may result in performance penalties. Dynamic metrics are more specific measurements of the dynamic behaviour of a program. They can be used to both identify performance problems on their own, and to explain and isolate performance problems identified by execution time measurements.

In this work, the **J* dynamic analysis framework [DDHV03] is used to calculate dynamic metrics. It provides a number of stock Java-based dynamic metrics that can be calculated for any program running in a JVM (and hence for both Java and AspectJ programs). In addition to these general dynamic metrics, this work defines some new AspectJ-specific dynamic metrics, implemented as extensions to **J*.

**J* supports the concept of *metric spaces*. Dynamic metrics are calculated on execution traces, which are sequences of runtime events. By default, they are calculated for the entire execution of a program. It can be useful, however, to calculate them for only a part of the execution, a subset of the runtime events. These subsets are called metric spaces, and are defined by partitioning schemes. **J* provides two basic partitioning schemes, both used in this work:

Whole program: This is the default partitioning scheme. All runtime events for the entire execution of the program contribute to the calculation of each metric.

Static Application/Library: This scheme distinguishes the code written by the user and produced by the compiler (application code) from that in runtime libraries (library code). The distinction is made by matching package names, and is configurable by adjusting the package name filters. For this thesis, code executed in the Java standard library and in the AspectJ runtime library is considered part of the library space.

The metrics described in the following sections are calculated and reported for each of these spaces: whole program, application, and library.

3.3 General Metrics

**J* provides implementations for a large number of general (Java-based) dynamic metrics, not all of which are relevant to this work. The following general dynamic metrics are used:

size.loadedClasses.value, size.load.value, size.run.value: These metrics give an indication of the static size of the program measuring the number of loaded classes, the number of loaded bytecode instructions, and the number of bytecode instructions executed at least once. The latter two, together, can provide a measure of code coverage, or dead code. A large difference in the first two metrics between AspectJ and Java versions of a program indicates code bloat.

base.instructions.value: This metric is a count of the total number of bytecode executions. Its value is at least as large as that of *size.run.value*. It gives a VM-neutral approximation of execution time, the unit of which being the kilobytecode (kbc).

The relationship between the number of executed bytecode instructions and execution time is, of course, somewhat tenuous, for several reasons. First, not all bytecode instructions are of equivalent cost. Second, the JIT compiler can significantly reduce the consequence of a large number of bytecode executions in ways that are difficult to predict. Nevertheless, this metric is the basis for several others that are particularly useful for assessing AspectJ overheads (the *tag mix* metric, for example).

base.objects.value, base.bytes.value, memory.objectAllocationDensity: These metrics describe the allocation behaviour of a program. *base.objects.value* counts the number of heap allocations made, *base.bytes.value* counts the total number of bytes allocated, and *memory.objectAllocationDensity* measures the allocation rate, indicating the number of allocations made per kbc.

As mentioned above, not all bytecodes are of equal cost, and some may be optimized away completely by the JIT. As such, it can be useful to restrict the count of instruction executions to expensive instructions that tend not to be optimized away. The *base.objects.value* metric is additionally useful in this capacity because the executions it represents, object allocations, tend to be expensive and tend not to be optimized away as readily as, for example, invoke instructions, which can be inlined.

3.4 AspectJ Metrics

The general metrics, while useful, are still incapable of explaining any AspectJ overheads present, or of reporting on behaviour as it relates to specific AspectJ language features. Therefore, in addition to these general metrics, a number of new AspectJ-specific metrics have been defined and implemented in **J*. These metrics make use

of AspectJ-specific metadata attached to class files, and report values related to specific AspectJ features. In this subsection, the key metrics are briefly defined. Chapter 4 explains the tools used to calculate them, chapter 5 explains the metadata and overhead kinds in more detail, and chapter 6 presents some of the more complicated computations required to calculate these metrics.

3.4.1 Instruction Kind Metrics

TagMix: The *tagmix* metric is a partition of bytecode executions into bins representing the different roles of the instructions in implementing AspectJ language features. Each bin corresponds to an *instruction kind*. The different instruction kinds are described in detail in chapter 5. Each bin is reported as both a percentage of total executions and as an absolute count. This metric is reported in both an execution and an allocation flavour. The former reports executions of any kind (that is, it partitions *base.instructions.value*,) while the latter reports only executions that result in space being allocated on the heap (that is, it partitions *size.objects.value*.)

As mentioned in section 3.3, this metric does not correspond directly to execution times, but is still quite useful, as is shown in chapter 7, especially in conjunction with execution time comparisons. Section 9.1.1 suggests some ways in which more accurate profiling of AspectJ overhead could be performed.

Aspect Overhead: The implementation of certain AspectJ language features results in some runtime overhead. The tagmix metric differentiates overhead from non-overhead executions—this metric is a summary, reporting the ratio of overhead executions to total executions. It is reported both for bytecode executions of any kind and for allocations. It indicates the efficiency of the AspectJ language implementation. A high value suggests that improvements can be made to the compiler. A high value may also indicate that the runtime cost of the AspectJ features could outweigh their benefits.

(A high value can, however, be misleading on its own, as it doesn't necessarily

imply a longer execution time, due to the effects of JIT compilation.)

Advice to Application Ratio: The advice to application metric indicates how much of the program's non-overhead execution is spent in advice. Benchmarks with large advice bodies, or advice bodies that execute very frequently, may spend the bulk of their time executing advice. This metric does not report on overhead.

Advice to Overhead Ratio, Overhead to Advice Ratio: These metrics indicate the ratio of non-overhead advice executions to overhead executions, and vice-versa. In a sense, they identify the runtime cost of implementing behaviour in advice.

Library Ratio: This metric indicates the percentage of executions made from within the AspectJ runtime library.

3.4.2 Advice Guard Metrics

Advice Execution: The advice execution metric is a partition of advice guards into three categories: those that always (for a run of the program) evaluate to true and are succeeded by execution of their associated advice, those that always evaluate to false, and those that sometimes evaluate to true and sometimes to false.

If it is found that a particular guard always evaluates to true or always evaluates to false, it may be true that the guard will always evaluate to true or false across all inputs to the program, and suggests that more sophisticated static analysis could completely remove this guard.

3.4.3 Shadow and Source Metrics

Advice Execution per Shadow: This metric reports the number of advice executions per join point shadow. It can be used to identify join point shadows at which advice is very frequently executed, and shadows at which advice is very

rarely executed. This metric, and the others in this subsection, could provide useful profiling information to programmers.

Hot Shadows: The hot shadows metric indicates the minimum number of shadows contributing to 80% of shadow executions. That is, it will indicate whether a small number of join point shadows are dominating the execution of a program or not.

Advice Execution per Source: A *source*, as further described in section 5.3, is an instance of an AspectJ construct that can result in woven bytecode instructions. This metric reports the number of advice executions per source; that is, the number of times each particular advice is executed.

Hot Advice: The hot advice metric indicates the minimum number of advice definitions contributing to 80% of advice executions. If the value is small, it indicates that there are hot advice, that is, advice bodies that are being executed with disproportionate frequency.

3.5 Summary

In summary, three kinds of measurements can be made: execution time, general dynamic metrics, and AspectJ-specific dynamic metrics. The AspectJ-specific dynamic metrics primarily identify AspectJ overhead, but can also provide other useful information, such as profiling information. They have been newly defined for this work, and the next several chapters examine them in more detail.

Chapter 4
Tools

This chapter describes the tools that were collected, modified, and created in order to study the dynamic behaviour of AspectJ programs and to perform the measurements described in chapter 3. The relationship between these tools is illustrated in Figure 4.1.

The toolchain consists of the following parts:

- An AspectJ benchmark suite consisting of representative AspectJ programs collected from a variety of public sources.
- AspectJ compilers modified to annotate the generated classfiles with additional metadata required by the dynamic metrics described in chapter 3. The compilers used are modified versions of `ajc` 1.2 and `abc` 1.0.2.
- A version of the **J* dynamic analysis framework, extended to compute the new AspectJ dynamic metrics.
- Various support tools for examining and manipulating the tagged classes, and for managing the entire process of metric computation.

The AspectJ compilers, and the modifications made to them, are described in more detail in section 4.1. The **J* dynamic analysis framework is described in section 4.2.

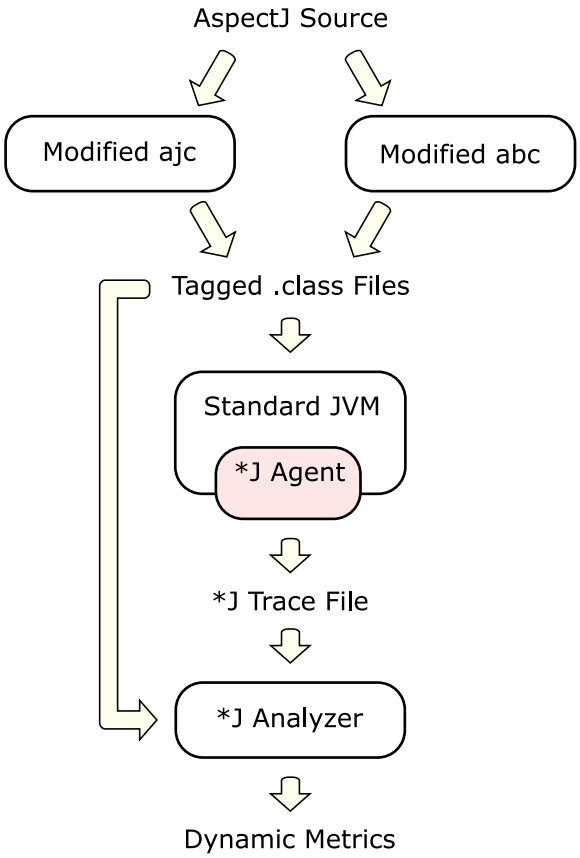


Figure 4.1: *Overview of metric collection tools*

4.1 AspectJ Compilers

There are currently two compilers for the AspectJ language. The first, `ajc` [Asp], is the original compiler, created by the language designers and now maintained as part of the Eclipse project. The second is the Aspect Bench Compiler (`abc`) [aG], which has been developed at McGill and Oxford Universities.

In order to implement the dynamic metrics described in the previous chapter, these compilers have been modified, as part of this thesis, to annotate the programs they produce with necessary metadata. Both compilers have been used so that their code generation strategies can be compared.

Although the design and architecture of these two compilers differ in the details (described further below), an important commonality is that they both have a distinct weaving phase in which aspects and base program are composed to produce pure Java bytecode representing the whole program. As mentioned in section 2.2, this is like a form of partial evaluation.

Both compilers distinguish between two forms of weaving: that which implements the static crosscutting features and modifies the static type structure of the program, and that which implements the dynamic crosscutting features and modifies method bodies. In each case, new instructions or methods may be generated in order to implement the required semantics—these are AspectJ overhead. It is these overhead instructions that we tag with additional metadata, and the weavers of both compilers have been augmented to do so.

This metadata is described in detail in chapter 5. The rest of this section describes in further detail the two compilers, and provides for each a tagging example.

4.1.1 ajc

`ajc` is the original AspectJ compiler and the reference implementation for the language. Its design has focused on fast incremental compilation and integration with the Eclipse suite of developer tools. Since version 1.1, it has performed aspect weaving at the bytecode level. (Previous versions performed weaving at the source code level.) The `ajc` architecture consists of a front-end compiler and a back-end weaver. The front-end compiler is an extended version of Eclipse's JDT compiler. It takes as input AspectJ source code and produces as output standard Java class files annotated with special attributes. These attributes contain all the aspect-specific information (pointcut definitions, for example) required by the weaver.

The major changes made to the classes being woven are performed by two kinds of *munger*. The first kind is the *type munger*, which changes the static type structure of the program, implementing intertype declarations. The second kind is the *shadow munger*, which manipulates join point shadows, implementing the dynamic crosscutting features of aspects. Each source-level instance of an AspectJ

construct requiring modification of the input bytecode has a corresponding munger instance. For example, a particular advice declaration would correspond to a particular shadow munger instance.

The modified weaver tags instructions that are generated by mungers with three pieces of metadata, as further described in chapter 5: instruction kind, shadow ID, and source ID. These tags indicate the role of the instructions in implementing AspectJ language features, identify which particular construct has resulted in their generation, and identify the particular join point shadow into which they are being woven.

Not all of the instructions that we wish to annotate with this metadata are generated by mungers during the weaving stage. Existing instructions in aspect classes, generated during the front-end AspectJ compilation, may also represent overhead that should be tagged. The front-end compiler could be modified to tag these instructions as they are generated, in the same manner that instructions are tagged during weaving, but since `ajc` supports the weaving of binary aspects for which the source may be unavailable, it is desirable to instead perform all tagging during the weaving stage. Therefore, at the beginning of the weaving stage, a "pretagging" operation is performed on all aspect classes, and instructions produced by the front-end compiler that should be tagged are tagged. Since the front-end compiler automatically generates special names for advice bodies and other methods implementing special AspectJ constructs, this is accomplished by searching for bytecode patterns in methods whose names match these naming conventions. An example case is that of an **around** advice body. The advice body is implemented as a method on the aspect class. For this method, we isolate the instructions implementing the **proceed** call, which is implemented as a call to a specially-named method, and tag them appropriately.

Tagging in ajc

Because `ajc` stores aspect information in classfile attributes so that it can support the weaving of binary aspects, `ajc` already has some infrastructure in place for

creating classfile attributes. This has been extended to support instruction tagging. Several new classes have been added to the `AjAttribute` class: `InstructionTagAttribute`, `InstructionKindAttribute`, `InstructionSourceAttribute`, and `InstructionShadowAttribute`. Tagging utility functions have been added to `weaver.bcel.Utility`. Unique instance IDs have been added to the `Shadow` and `ShadowMunger` classes, implementing shadow and source IDs, respectively.

The following code listing illustrates a simple example: tagging the instructions added to implement per-object aspect instance binding. This is a method in `weaver.bcel.BcelShadow`.

```
public void weavePerObjectEntry(final BcelAdvice munger,
                        final BcelVar onVar)
{
  final InstructionFactory fact = getFactory();

  InstructionList entryInstructions = new InstructionList();
  InstructionList entrySuccessInstructions = new InstructionList();
  onVar.appendLoad(entrySuccessInstructions, fact);

  entrySuccessInstructions.append(
    Utility.createInvoke(fact, world,
      AjcMemberMaker.perObjectBind(munger.getConcreteAspect())));

  InstructionList testInstructions =
    munger.getTestInstructions(
      this,
      entrySuccessInstructions.getStart(), range.getRealStart(),
      entrySuccessInstructions.getStart());

  // tag the dynamic residue:
  Utility.tagInstructionList(
    testInstructions,
    AjAttribute.InstructionKindAttribute.PEROBJECT_ENTRY_TEST,
    this,
```

```
    munger,
    true);

  entryInstructions.append(testInstructions);
  entryInstructions.append(entrySuccessInstructions);

  // tag the aspect instance binding instructions:
  Utility.tagInstructionList(
    entryInstructions,
    AjAttribute.InstructionKindAttribute.PEROBJECT_ENTRY,
    this,
    munger);

  List oldIl = Utility.instructionListToList(range.getBody());
  range.insert(entryInstructions, Range.InsideBefore);
  List newIl = Utility.instructionListToList(range.getBody());

  // tag BCEL artifacts:
  Utility.tagUntaggedNewInstructions(
    oldIl,
    newIl,
    AjAttribute.InstructionKindAttribute.BCEL,
    this,
    munger);
}
```

Listing 4.1: *Tagging per-object aspect instance binding instructions in* `ajc`

This code tags any dynamic residue generated with the PEROBJECT_ENTRY_TEST tag, the instructions that bind the aspect instance with the PEROBJECT_ENTRY tag, and certain instructions that are artifacts of BCEL with the BCEL tag. In each case, the shadow and munger are passed to the tagging function, from which the shadow and source IDs are read.

4.1.2 abc

Where `ajc`'s primary design goals are fast incremental compilation and integration with developer tools, `abc`'s are extensibility [ACH+05a] and optimization [ACH+05b]. The motivation for its development is two-fold. First, research into aspect-oriented languages and AspectJ is active and ongoing. Development of new language features requires a suitable workbench, and integration with a "real-world" aspect-oriented language, like AspectJ, is of great value. The `abc` compiler was developed to be such a workbench, providing an extensible framework in which a wide-variety of extensions can be made to AspectJ with a minimum of effort. Second, `abc` has been designed as an optimizing implementation of AspectJ. It implements some basic optimizations for AspectJ code generation and provides a framework enabling the development of new analyses and optimizations. This facet of `abc` is explained further in chapter 7.

`abc` is built upon several existing tools [ACH+04]. Its front-end is based on Polyglot [NCM03], an extensible compiler front-end framework, and its back-end is based on SOOT [VRGH+00], a Java bytecode analysis and transformation framework. Polyglot simplifies the development of language extensions, and SOOT simplifies the development of new compiler analyses and optimizations.

The basic architecture of `abc` is illustrated in Figure 4.2. Some notable differences between `abc` and `ajc` are as follows. The front-end produces a pure (but possibly incomplete) Java AST with an associated `AspectInfo` data structure, which contains the information describing the AspectJ constructs. The AST and `AspectInfo` data structure are input to the weaver, whose output is in the Jimple intermediate representation used by SOOT. The weaver first performs static weaving. The Jimple skeleton generated from the AST is modified as required by all `declare parents` statements and all intertype member declarations: the inheritence structure is changed and empty methods are added. This process is denoted "skeleton weaving" in Figure 4.2.

Next, the Jimple skeleton is filled out with method bodies. AspectJ constructs that contain code, such as advice bodies and **if** pointcuts, are implemented here as

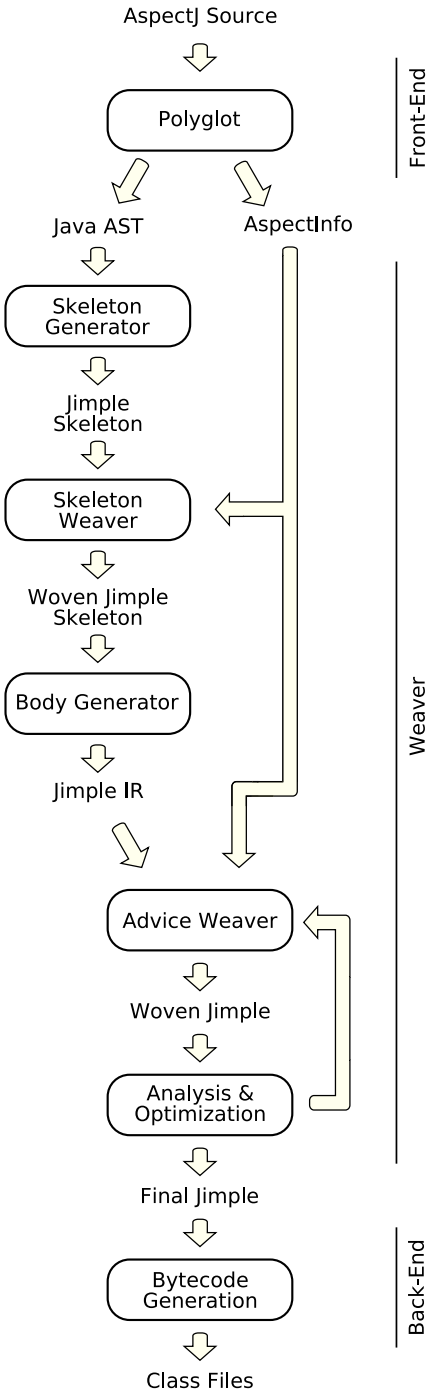


Figure 4.2: *Overview of abc's architecture*

method bodies. Next, the weaving of dynamic features, such as advice, is performed on the Jimple bodies. This is indicated as "advice weaving" in the figure.

As in `ajc`, in addition to tagging instructions generated by the weaver, some instructions generated before advice weaving need to be tagged—the bodies of "normal" methods on aspect classes, and the return statements of advice bodies, for example.

Tagging in abc

In `abc`, much of the tagging functionality is defined in the package `abc.weaving.tagkit`. The tagging is implemented using SOOT's annotation framework, and each type of tag to be attached to a bytecode instruction extends `InstructionTag`, which implements the SOOT interface `Tag`. The `Tagger` class contains a number of utility functions for adding tags to Jimple statements.

What follows is a simple example illustrating how tagging is performed for some of the bookkeeping code required to implement **cflow** pointcuts. In `abc`, the bookkeeping code is added by implementing it as a piece of synthetic advice. A data structure, representing the validity of the pointcut and storing any bound context, must be maintained. For any pointcut **cflow** (*P*), every entry to and exit from join points matching *P* must trigger updates to this data structure. The bookkeeping instructions are inserted at the entry points by constructing a piece of synthetic **before** advice and weaving it into the program. The `abc.weaving.aspectinfo.CflowSetup` class implements a synthetic advice declaration in this manner. The `makeAdviceExecutionStmts` method generates the instructions to be inserted at the relevant join point shadows, returning them as a `Chain`[1]. The instructions in this chain are tagged appropriately with instruction kind, shadow ID, and source ID tags.

```
public Chain makeAdviceExecutionStmts(
  AdviceApplication adviceappl,
  LocalGeneratorEx localgen,
```

[1] A `Chain` is a SOOT data structure similar to a `List`.

```
  WeavingContext wc)
{
  CflowSetupWeavingContext cswc=(CflowSetupWeavingContext) wc;
  Chain c = new HashChain();
  SootMethod m = localgen.getMethod();
  Local cflowInstance = getMethodCflowLocal(localgen, m);
  Local cflowLocal = getMethodCflowThreadLocal(localgen, m);

  if (cswc.doBefore) {
    // PUSH
    Chain getInstance = codeGen()
      .genInitLocalLazily(localgen, cflowLocal, cflowInstance);
    c.addAll(getInstance);
    List/*<Value>*/ values = new LinkedList();
    Iterator it = cswc.bounds.iterator();
    while (it.hasNext()) {
      Value v = (Value)it.next();
      values.add(v);
    }
    ChainStmtBox pushChain =
      codeGen().genPush(localgen, cflowLocal, values);
    c.addAll(pushChain.getChain());
    pushStmts.put(adviceappl, pushChain.getStmt());

    // tag the entry instructions with a kind tag:
    Tagger.tagChain(c, InstructionKindTag.CFLOW_ENTRY);

    } else {
      // POP
      ChainStmtBox popChain =
        codeGen().genPop(localgen, cflowLocal);
      c.addAll(popChain.getChain());
      popStmts.put(adviceappl, popChain.getStmt());

      // tag the exit instructions with a kind tag:
      Tagger.tagChain(c, InstructionKindTag.CFLOW_EXIT);
    }
```

```
    // tag the instructions with source and shadow IDs:
    Tagger.tagChain(c,
      new InstructionSourceTag(adviceappl.advice.sourceId));
    Tagger.tagChain(c,
      new InstructionShadowTag(adviceappl.shadowmatch.shadowId));
    return c;
}
```

Listing 4.2: *Tagging* **cflow** *bookkeeping instructions in* `abc`

For both `abc` and `ajc`, accurate instruction tagging can require some more significant changes to the code, such as passing necessary context information, but these simple examples should give a basic idea of how tagging is performed in both compilers.

4.2 *J Dynamic Analysis Framework

The **J* framework is a tool for performing offline dynamic analyses of Java programs. It was originally intended for the calculation of dynamic metrics, but is also capable of many other dynamic analyses. It consists of two main components: the **J* trace collection agent, and the **J* analyzer. The trace collection agent interfaces to a running JVM via the JVMPI, receiving runtime events, and encoding them in an execution trace file. The analyzer is a Java program that processes the execution trace produced by the agent, performing analyses and computing dynamic metrics. It processes the execution trace sequentially, feeding each encoded event into its pipeline of operations. Each operation in the pipeline is either a service, required by subsequent operations, or a metric computation.

In order to implement the metrics defined in chapter 3, **J* has been extended in three ways:

1. The class file reader has been extended to read the metadata attached to classfiles produced by the modified AspectJ compilers.

2. A tag-propagation analysis has been written to assign appropriate tags to each instruction execution event. This algorithm takes as input the static tags added to bytecode instructions by the compiler, described in the previous section, and "propagates" them to runtime instruction execution events appropriately, so that the instruction executions can be properly accounted.

3. The AspectJ-specific dynamic metrics defined in chapter 3 have been implemented as **J* analayses.

The tag-propagation algorithm and the implementation of some of the dynamic metrics (particularly in the presence of `abc`'s advice inlining optimizations) are fairly significant extensions to **J*. These computations are described in detail in chapter 6, as their understanding first requires an understanding of the static metadata attached to woven classes, which is described in chapter 5.

Chapter 5
Static Tags

This chapter details the metadata attached to code compiled with the modified compilers described in chapter 4. There are three basic types of metadata tags attached to instructions: *instruction kind tags*, described in section 5.1, which indicate the role of instructions generated by the aspect weaver; *instruction shadow tags*, described in section 5.2, which identify the join point shadow into which instructions have been woven; and *instruction source tags*, described in section 5.3, which indicate what particular instance of an AspectJ construct is responsible for the woven instruction. Section 5.4 describes the *inline count*, *inlined shadow/source list*, and *proceed tag*, which are added to the instructions of inlined advice and proceed bodies. Finally, section 5.5 describes how all of these tags are encoded in the class files.

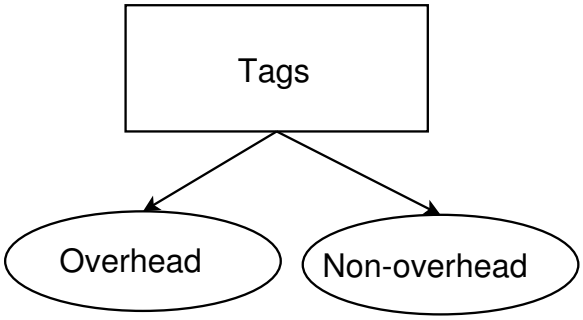
5.1 Instruction Kind Tags

Each bytecode instruction in a compiled AspectJ program has a particular role with relation to the implementation of AspectJ language features. An instruction may correspond directly to Java code written by the user, or it may be overhead introduced to support a specific AspectJ language feature, such as advice execution. Hereafter, these roles are referred to as *instruction kinds*.

During the weaving stage of compilation, many instructions are generated in order to implement the semantics of AspectJ. These instructions are weaving-induced overhead, the execution of which contributes to AspectJ's runtime cost. For each instruction, the nature of this overhead—that is, the instruction kind—is identified by an *instruction kind tag* attached to the instruction by the weaver.

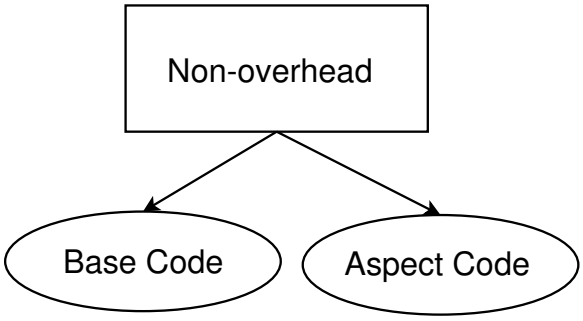
This section describes the various instruction kinds with which instructions are associated. Instruction kinds can be categorized hierarchically, and the tags are presented below, by category. Figure 5.1, at the end of this section, illustrates the complete tree of instruction kind categories.

5.1.1 Instruction Kinds



Every instruction is either overhead or non-overhead. Overhead instructions are those instructions generated and inserted by the weaver, used to implement particular AspectJ features, such as advice, **cflow** pointcuts, intertype declarations, *etc.* Non-overhead instructions are those corresponding directly to Java code written by the user, either in the base program or in the aspect. (One can think of overhead instructions as approximately those that can be traced back to the special AspectJ syntax in the original source code, and non-overhead instructions those that trace back to Java syntax.)

5.1.2 Non-overhead tags

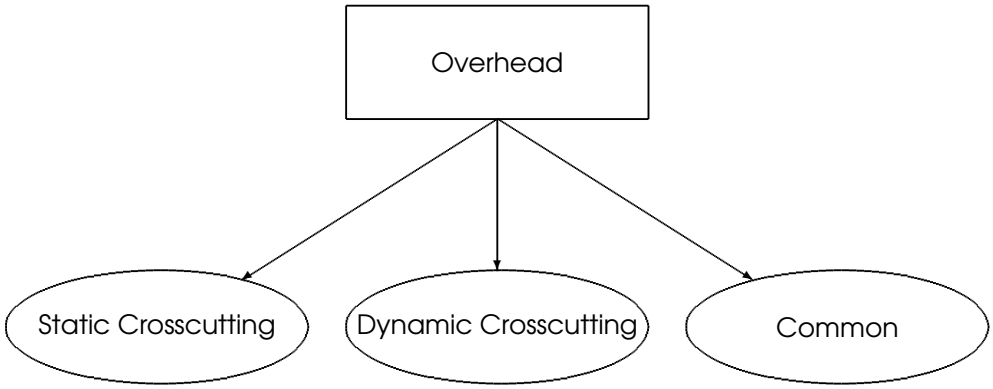


There are two basic kinds of non-overhead instruction: BASE_CODE and ASPECT_CODE. BASE_CODE instructions are those that would exist if compilation and weaving of aspects were omitted completely, and represent all of the functionality described by the programmer in normal Java classes. ASPECT_CODE is that code which is defined by the user in an aspect (in Java syntax), either in advice bodies, introduced methods, or normal methods in an aspect class. ASPECT_CODE instructions are also all those instructions in the base program that execute within the dynamic scope of an ASPECT_CODE instruction. So, for example, any methods in the base program called from advice bodies are considered ASPECT_CODE.

In addition to these two basic kinds, there are special subkinds of each: INLINED_ADVICE and INLINED_PROCEED. INLINED_ADVICE represents the non-overhead instructions that are part of an inlined advice body and INLINED_PROCEED represents the non-overhead instructions that are part of a proceed body. They are counted as ASPECT_CODE and BASE_CODE, respectively.

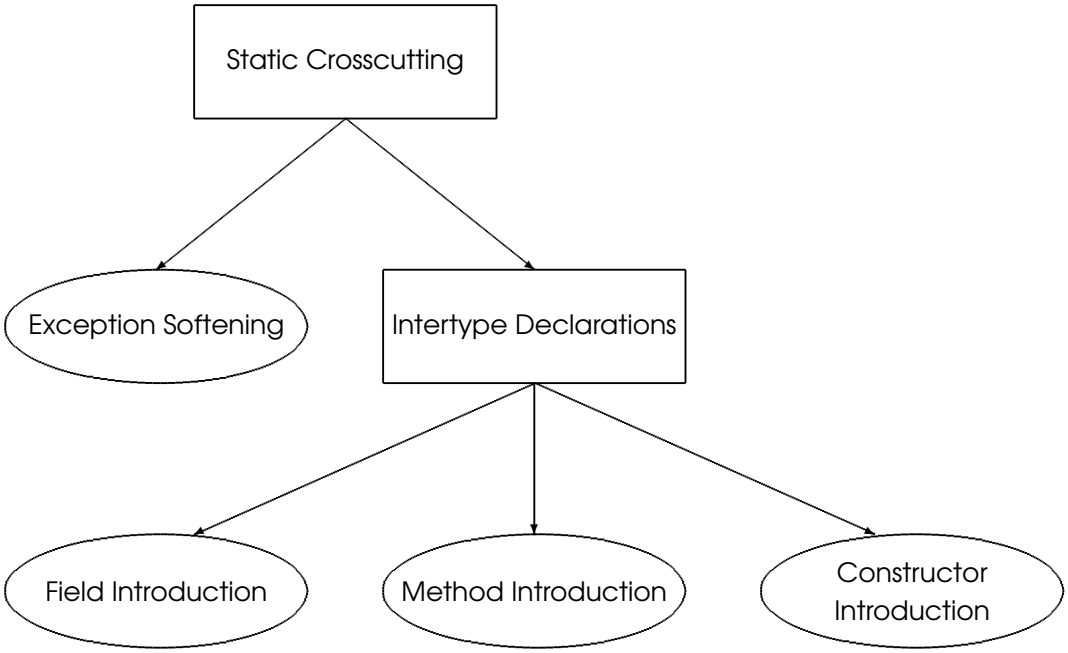
This distinction between BASE_CODE and ASPECT_CODE is more arbitrary than the distinction between overhead and non-overhead instructions, or the distinction between each kind of overhead instruction. How to distinguish the two—how to count "base code" executed below an advice body, for instance—depends on what measurements one eventually wants to make. This policy is codified in the propagation scheme described in section 6.1.

5.1.3 Overhead tags



Overhead instruction kinds can be categorized into three groups: those that implement the dynamic crosscutting features of AspectJ, those that implement the static crosscutting features of AspectJ, and those that are common to the implementations of both.

5.1.4 Static overhead tags



The static crosscutting features of AspectJ, somewhat surprisingly, also incur some runtime overhead. This overhead generally takes the form of dispatch methods introduced into the target classes of intertype declarations. The various overhead kinds relating to static crosscutting are:

EXCEPTION_SOFTENER The `declare soft` declaration in an aspect takes as parameters a type pattern and a pointcut. It causes any exceptions matching the given type pattern that occur within join points matched by the given pointcut to be wrapped in an unchecked *org.aspectj.SoftException*. The implementation of this feature requires the addition of instructions to catch the checked exceptions matching the type pattern, at the appropriate join point shadows, and the instantiation and throwing of the new unchecked exception. These instructions are of this kind.

INTERMETHOD Intertype method declarations result in a dispatch method being added to the target class. This dispatch method calls the actual body of the introduced method, which is compiled in the aspect class. The instructions of the dispatch method are of this kind.

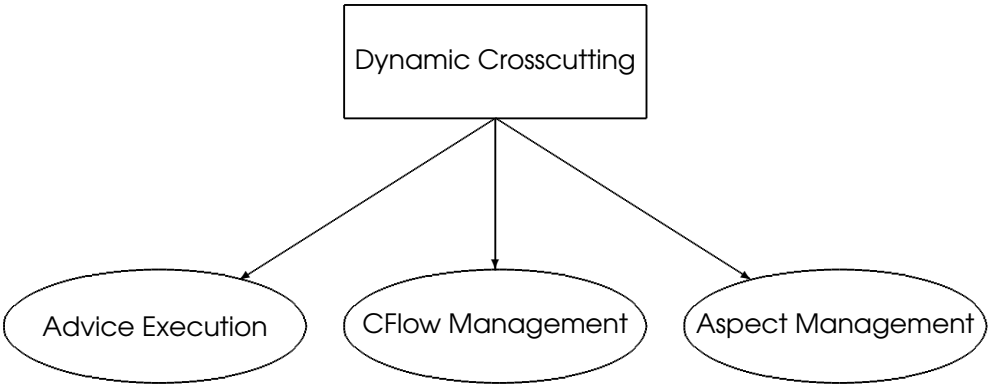
INTERFIELDGET, INTERFIELDSET Intertype field declarations may result in accessor methods being added to the target class. All references to the introduced fields are, at the bytecode level, made through these accessor methods. The instructions making up these accessors are of these kinds.

INTERFIELDINIT Intertype field declarations require initialization code to be added to either the target class's constructor or to its static initializer. This code may invoke methods on the aspect class to initialize the values of introduced fields. The initialization code is of this instruction kind.

INTERCONSTRUCTOR_PRE, INTERCONSTRUCTOR_POST If an aspect has an intertype constructor declaration, two methods are compiled in the aspect class: `preInterConstructor` and `postInterConstructor`. A new constructor, invoking these two methods, is added to the target class. These methods, and their invocation, are of this kind.

INTERCONSTRUCTOR_CONVERSION An introduced constructor may have instructions used to wrap constructor arguments in an `Object` array and to box and unbox primitive constructor arguments. These instructions are of this kind.

5.1.5 Dynamic overhead tags



The overheads due to the dynamic features of AspectJ can be grouped into three categories: those that implement the execution of advice, those that manage the per-object and per-cflow instances of aspects, and those that maintain the abstraction of the call stack required for the implementation of **cflow** pointcuts.

5.1.6 Advice execution tags

ADVICE_EXECUTE Advice bodies get compiled as methods in the aspect class. During weaving, invoke instructions calling these methods are added to the relevant join point shadows. The invocations of these advice body methods, and related instructions, are tagged as being of this kind.

ADVICE_ARG_SETUP Before an advice body method in an aspect can be executed, the aspect instance must be acquired and put on the stack. Furthermore, local state may need to be exposed to the advice body. The instructions that are woven in to perform these tasks are of this kind, which is closely related to ADVICE_EXECUTE.

ADVICE_TEST When it cannot be statically determined whether an advice body should be executed at all join points corresponding to the join point shadow at which the advice invocation instructions have been added, then those invocation instructions are wrapped in a test. This test is called an advice guard, which is a kind of dynamic residue. The instructions comprising this guard are of this kind.

AROUND_PROCEED, AROUND_CALLBACK The instructions required to implement the execution of the advised join point from within the body of an around advice—that is, the instructions required to implement the **proceed** call—are of these kinds. (AROUND_CALLBACK is basically synonymous with AROUND_PROCEED, but is only found in **around** closures.)

CLOSURE_INIT There are currently several different ways **around** advice is implemented, and a given compiler may choose from several different strategies for weaving **around** advice. Some strategies involve the creation of closure classes. This instruction kind represents the instantiation of these closure classes.

AFTER_RETURNING_EXPOSURE **after** and **after returning** advice may expose the value returned by the advised join point to the body of the advice. The instructions that implement this return value exposure are of this kind.

AFTER_THROWING_HANDLER The implementation of **after throwing** advice requires the generation of exception handling code that catches any uncaught exceptions thrown in the advised join points, for the advice body to be executed, and for the original exception to be rethrown after the execution of advice. The instructions responsible for catching and rethrowing the exception are of this kind.

AROUND_CONVERSION The implementation of **around** advice may require the arguments to and/or the return value from a **proceed** call to be stored within

an object array. In this case, the instructions that store and retrieve the arguments from this array, and that box and unbox arguments of primitive type, are of this kind.

THISJOINPOINT Advice bodies can reflectively examine the join point at which they are executing by examining a data structure representing it. A reference to this data structure is made available by the `thisJoinPoint` and `thisJoinPointStaticPart` keywords. Instructions that create this data structure, making it available to advice bodies, are generated by the weaver. These instructions have this kind.

5.1.7 Aspect instance management tags

PEROBJECT_ENTRY By default, aspect instances are singletons. They can, however, be associated on a per-object basis—either with the current executing object, or with the target object—at join points selected by a given pointcut. The instructions, inserted at join point shadows matched by the pointcut, to manage these aspect instances, are of this kind.

PEROBJECT_GET, PEROBJECT_SET If an aspect instance can be associated with an object, then that object needs accessor methods implementing the acquisition and setting of the instance. The instructions comprising these methods are of these kinds.

PERCFLOW_ENTRY, PERCFLOW_EXIT Aspect instances can also be associated with **cflow** constructs. The instructions inserted to manage per-cflow aspect instances have these kinds.

CFLOW_TEST This kind is congruent to ADVICE_TEST. It represents the dynamic residue at a **cflow** update shadow consequent of another **cflow** pointcut. The instructions implementing this residue are of this type.

PEROBJECT_ENTRY_TEST This kind is also congruent to ADVICE_TEST. When not all of the join points at a join point shadow match the pointcut associated with a

per-object aspect declaration, then a runtime test is woven in at the join point shadow. The instructions implementing this test are of this type.

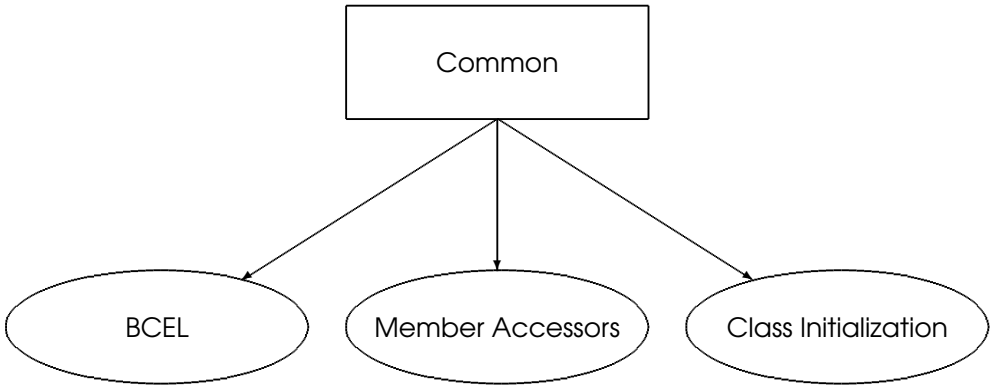
5.1.8 Cflow management tags

CFLOW_ENTRY, CFLOW_EXIT The **cflow** and **cflowbelow** pointcuts require that a representation of the call stack be managed during execution of the program. If the **cflow** pointcut does not expose any state, then this representation may be as simple as a counter per pointcut, otherwise it may be an actual stack; the particular implementation depends on the compiler and the situation. The instructions that update this representation (regardless of implementation) on entering and leaving join points matching the associated pointcut, are of these kinds.

5.1.9 Common dynamic tags

GET_CFLOW_LOCAL, GET_CFLOW_THREAD_LOCAL A **cflow** pointcut may result in multiple tests being woven into a single method body. A naive implementation may require the **cflow** state object be acquired at each test. An optimized implementation, however, might acquire it a single time on entry to the method, caching it to a local. Since a single pointcut may be used for both advice and per-cflow tests, the instructions responsible for caching the stack to a local are common to both kinds. They are of these kinds.

5.1.10 Common tags



BCEL This kind is an artifact of the use of the BCEL (Bytecode Engineering Library) [Fou] within `ajc`. The insertion of instructions by the weaver occasionally results in the addition of spurious `nop` instructions that are purely artifacts of BCEL and do not implement AspectJ features. These spurious instructions are of this kind.

PRIV_METHOD, PRIV_FIELD_GET, PRIV_FIELD_SET An aspect may be declared privileged, in which case it can access the private members of other classes. In order to implement privileged aspects, public wrapper methods for each private method the aspect may call, and public accessor methods for each private field the aspect may reference, are added to each method and field's containing class. The instructions in these methods are of these kinds.

CLINIT The static initializer of the aspect classes have instructions of this kind. The static initializer may setup the default singleton instance of the aspect, or it may setup the data structures used to model the call stack for the implementation of **cflow** pointcuts. Instructions added to the static initializers of base program classes are also of this kind. Instructions in the static initializer that initialize static join point information, however, are of THISJOINPOINT kind.

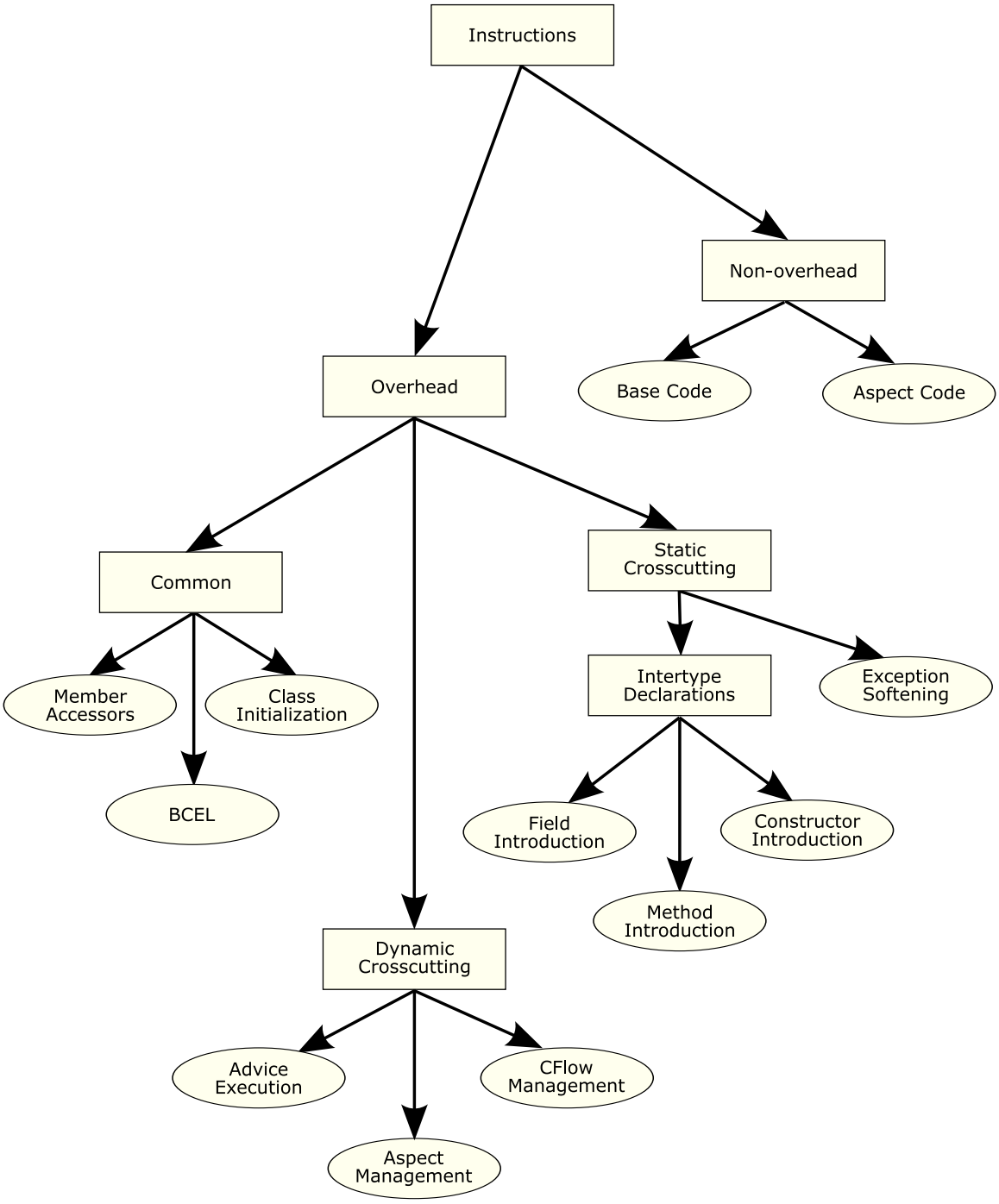


Figure 5.1: *Complete taxonomy of instruction kind categories*

5.2 Shadow ID Tags

Each join point shadow at which the weaver has inserted instructions has a unique ID called the *shadow ID*. It is unique across the entire program. Every instruction generated by the weaver is assigned the shadow ID of the join point shadow to which it has been added. The tag holding this ID is called the *instruction shadow tag*. In effect, the instruction shadow tag identifies the location of an instruction.

For example, consider the code in Listing 5.1. It declares a class with two method calls in the `main` method, and an aspect, which declares two pieces of advice, which apply to the two method calls in `main`.

Listing 5.2 shows the result of compiling and weaving the code in Listing 5.1. The instruction shadow tag is listed (in square brackets to the left of each bytecode instruction) for those instructions that have been woven into a join point shadow. There are two shadows, one comprising the call to `foo1`, the other the call to `foo2`. Instructions at the first shadow have ID 1, instructions at the second, ID 2.

5.3 Source ID Tags

The specific instances of AspectJ language features that result in the generation of instructions by the weaver are referred to, in this thesis, as *sources*. Sources include such constructs as advice declarations, **cflow** pointcuts, intertype declarations, and per-object aspect declarations. The implementation of each of these may require instructions to be woven into the base program. Each source in a program has a unique ID. The instructions that result from the implementation of a particular source are tagged by the weaver with an *instruction source tag*, the value of which is the source ID.

For example, consider the aspect in Listing 5.3. It contains four sources, each given a unique ID. Source 1 is the **perthis** declaration. It will require having instructions woven in before each method call in the `Foo` class, associating the correct instance of `TheAspect` with the target of the call, such that `aspectOf`, when

```
public class Base {
  public static void main(String[] args) {
    foo1(); // shadow 1
    foo2(); // shadow 2
  }

  public static void foo1() {
    // do something
  }

  public static void foo2() {
    // do something
  }
}

aspect TheAspect {
  // advice 0:
  before(): call(void Base.foo*()) {
    // do something
  }

  // advice 1:
  before(): call(void Base.foo*()) {
    // do something
  }
}
```

Listing 5.1: *AspectJ program with multiple join point shadows*

```
[1: ] invokestatic TheAspect.aspectOf()
[1: ] invokevirtual TheAspect.before$0()
[1: ] invokestatic TheAspect.aspectOf()
[1: ] invokevirtual TheAspect.before$1()
[ : ] invokestatic Base.foo1()
[2: ] invokestatic TheAspect.aspectOf()
[2: ] invokevirtual TheAspect.before$0()
[2: ] invokestatic TheAspect.aspectOf()
[2: ] invokevirtual TheAspect.before$1()
[ : ] invokestatic Base.foo2()
[ : ] return
```

Listing 5.2: *Bytecode listing of the* `main` *method from Listing 5.1. Instruction shadow tags are indicated with square brackets.*

called, will return the correct instance.

Source 2 is the **cflow** fragment of the pointcut declaration. The **cflow** primitive pointcut will require instructions that manage the **cflow** state objects. These instructions will be tagged with the source ID of the **cflow** pointcut.

Source 3 is the advice declaration which uses the previously declared pointcut. The instructions woven in to test the validity of the **cflow** pointcut and to execute the advice will have this source ID. Note the instructions that test the **cflow** state will have the source ID of the advice declaration, which is different from that of the **cflow** declaration.

Source 4 is an advice declaration that applies to the same join points as source 3. Its instructions will be woven in at the same shadow, but have a different source ID.

Listing 5.4 is a pseudo-bytecode listing of the result of weaving this aspect into the base program class listed in Listing 5.2. The instructions with instruction source tags have them listed, in square brackets, to the left of the bytecode generated by the weaver to implement each of the above-listed features.

```
public aspect TheAspect
  // source 1:
  perthis(call(void Base.foo*()))
{

  // source 2:
  pointcut pc(): call(void Base.foo2())
              && cflow(call(void Foo.foo1()));

  // source 3:
  before(): pc {
    // do something
  }

  // source 4:
  before(): pc {
    // do something else
  }
}
```

Listing 5.3: *Different sources in an aspect*

Listing 5.5 shows the code in Listing 5.1 with both shadow and source IDs, in square brackets, to the left of each instruction. The first value is the shadow ID and the second value is the source ID. This listing illustrates how a given source may result in instructions with the same source ID being woven into different shadows, and how different sources may result in instructions being woven into the same shadow, and thus being tagged with the same shadow ID. What is important to note is that a particular advice execution, for example, at a particular join point shadow, is uniquely identified by the shadow/source ID pair.

```
public static void main(String[] arg0)
  0: [ : ] /* store cflowCounter$0 to local */
  6: [ : ] new Base
  9: [ : ] astore_0
 10: [ : ] aload_0
 11: [ : ] invokespecial Base.<init>:()V
 14: [ :1] aload_0
 15: [ :1] invokestatic TheAspect.abc$perTargetBind(Object)
 18: [ :2] /* cflowCounter$0.push() */
 42: [ : ] aload_0
 43: [ : ] invokevirtual Base.foo1()
 46: [ :2] /* cflowCounter$0.pop() */
 76: [ :1] aload_0
 77: [ :1] invokestatic TheAspect.abc$perTargetBind()
 80: [ :3] aload_0
 81: [ :3] invokestatic TheAspect.hasAspect()
 84: [ :3] ifeq -> 118
 87: [ :3] /* if cflowCounter$0 == 0 then goto -> 118 */
111: [ :3] aload_0
112: [ :3] invokestatic TheAspect.aspectOf(Object)
115: [ :3] invokevirtual TheAspect.before$0()
118: [ :4] aload_0
119: [ :4] invokestatic TheAspect.hasAspect()
122: [ :4] ifeq -> 156
125: [ :4] /* if cflowCounter$0 == 0 then goto -> 156 */
149: [ :4] aload_0
150: [ :4] invokestatic TheAspect.aspectOf()
153: [ :4] invokevirtual TheAspect.before$1()
156: [ : ] aload_0
157: [ : ] invokevirtual Base.foo2()
160: [ : ] return
```

Listing 5.4: *Pseudo-bytecode of aspect in Listing 5.3 woven into base program in Listing 5.1. Instruction source tags are indicated with square brackets.*

```
[1:1] invokestatic TheAspect.aspectOf()
[1:1] invokevirtual TheAspect.before$0()
[1:2] invokestatic TheAspect.aspectOf()
[1:2] invokevirtual TheAspect.before$1()
[ : ] invokestatic Base.foo1()
[2:1] invokestatic TheAspect.aspectOf()
[2:1] invokevirtual TheAspect.before$0()
[2:2] invokestatic TheAspect.aspectOf()
[2:2] invokevirtual TheAspect.before$1()
[ : ] invokestatic Base.foo2()
[ : ] return
```

Listing 5.5: *Woven bytecode listing of the* `main` *method from Listing 5.1. Instruction shadow and source tags, in that order, are indicated with square brackets.*

5.4 Inlined Advice Tags

Advice bodies, if they are small enough, may be inlined by the compiler as an optimization. In the case of **around** advice, the method containing the advised shadow, normally called when execution of a **proceed** statement occurs, may also be inlined. When this happens, information required by the metric calculations could be lost. The metadata described in this section, attached to classes by the inlining optimizer, ensure that the information is not lost. Inlining of advice requires three new instruction tags: the *inline count*, the *inlined shadow/source list*, and the *proceed tag*.

The inline count of an instruction is incremented each time it is inlined, either as part of an advice body, or as part of a proceed method. It is used to identify when an advice body or a proceed body is entered or exited.

An instruction's inlined shadow/source list indicates which inlined advice bodies an instruction belongs to. It is non-empty when the instruction is part of an advice body that has been inlined into a join point shadow. Each time instructions from an advice body are inlined, all of the instructions in the body add the shadow/source ID pair of the call site to their list of inlined shadow/source IDs, associating them to the advice execution. The attribute is a list because an instruction can be inlined multiple times in the case of advice applying to advice.

The proceed tag identifies an instruction as being part of a proceed body. An instruction can be identified as belonging to an inlined advice body when it has a non-empty inlined shadow/source list. (Note that its instruction kind, shadow, and source tags can be anything.) Distinguishing between an inlined advice body and a proceed body, however, requires the proceed tag. Whenever a proceed body is inlined, the proceed tag is attached to all of the body's instructions. (Currently, the tag is single-valued, as that is sufficient for calculating the metrics in this thesis. Tagging each body with a unique ID, however, is a conceivably useful extension.)

In addition to these three new tags, an appropriate kind tag needs to be assigned to the inlined instructions. This is described in section 6.1.5.

What follows is an example illustrating inlining of a simple advice body. Consider the bytecode in Listing 5.6. The bytecode instruction at offset 10 invokes the method `Aspect.before$0`, implementing the **before** advice that applies at join points corresponding to this shadow. The advice body has only a single statement: `System.out.println("before foo()")`. It is, consequently, a candidate for inlining. Listing 5.7 shows the result of inlining this advice. The instructions at offsets 11-16 are those that were in the method `Aspect.before$0`, retaining their original instruction kind tags, if they had any, or receving the kind tag INLINED_ADVICE if they didn't, as is the case here.

The inlined instructions each have an inline count of 1, and a single entry in the inlined shadow/source tag list: the pair `(1:3)`, which is the shadow/source tag of the original invoke instruction at offset 10 in Listing 5.6. This identifies these instructions as being part of the advice that applies at this join point.

If, instead of being inlined into `main`, but rather into another advice body, and an invoke instruction for this advice with shadow/source ID `(2:4)` were inlined elsewhere (at offset 10), then the instructions currently at offsets 11–16 would have the new list `[(1:3), (2:4)]` and an inline count of 2, while all the others would have the list `[(2:4)]` and an inline count of 1. This is illustrated in Listing 5.8.

```
public static void main(String[] arg0)
  0: [ : ] new DoubleInline
  3: [ : ] dup
  4: [ : ] invokespecial DoubleInline.<init>()
  7: [1:3] invokestatic Aspect.aspectOf()
 10: [1:3] invokevirtual Aspect.before$0()
 13: [ : ] invokevirtual DoubleInline.foo()
 16: [ : ] return
```

Listing 5.6: *Method body with no advice inlining. Shadow and source tags are indicated with square brackets.*

```
public static void main(String[] arg0)
  0: [ : ][ ] new DoubleInline
  3: [ : ][ ] dup
  4: [ : ][ ] invokespecial DoubleInline.<init>()
  7: [1:3][ ] invokestatic Aspect.aspectOf()
 10: [1:3][ ] pop
 11: [ : ][(1:3)](1) getstatic System.out
 14: [ : ][(1:3)](1) ldc "before foo()"
 16: [ : ][(1:3)](1) invokevirtual PrintStream.println(String)
 19: [ : ][ ] invokevirtual DoubleInline.foo()
 22: [ : ][ ] return
```

Listing 5.7: *Method body with inlined advice body. Shadow and source tags are indicated with square brackets. Inlined shadow/source list indicated by second set of square brackets, followed by inlined count in parentheses.*

```
 10: [ : ][(2:4)](1) new DoubleInline
 13: [ : ][(2:4)](1) dup
 14: [ : ][(2:4)](1) invokespecial DoubleInline.<init>()
 17: [1:3][(2:4)](1) invokestatic Aspect.aspectOf()
 20: [1:3][(2:4)](1) pop
 21: [ : ][(2:4),(1:3)](2) getstatic System.out
 24: [ : ][(2:4),(1:3)](2) ldc "before foo()"
 26: [ : ][(2:4),(1:3)](2) invokevirtual PrintStream.println(String)
 29: [ : ][(2:4)](1) invokevirtual DoubleInline.foo()
 32: [ : ][(2:4)](1) return
```

Listing 5.8: *Multiple inlining of advice.*

5.5 Tag Representation

Since dynamic metrics are calculated in a separate program, the tags defined in the preceding sections need to be encoded into the class files produced by the compiler, from which they can later be read. This section defines the binary format of these tags and how they are encoded in Java class files.

Java class file attributes can be attached to several different class file structures: classes, fields, methods, and code attributes. (A code attribute is a special attribute attached to a method, to which other attributes can be attached.) The class file attributes used to encode the tags described in this chapter are presented below, according to the class file structure to which they are attached.

5.5.1 Code Attributes

The instruction tags presented in the previous sections are all encoded as code attributes in the class files produced by the compiler. As per the JVM spec [LY99], all attributes have the following structure:

```
attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
}
```

`attribute_name_index` is an index into the constant pool, pointing to the attribute's name. The attribute names for the above tags are:

1. `ca.mcgill.sable.InstructionKind`
2. `ca.mcgill.sable.InstructionShadow`
3. `ca.mcgill.sable.InstructionSource`
4. `ca.mcgill.sable.InstructionInlineCount`

5. `ca.mcgill.sable.InstructionInlineProceed`

6. `ca.mcgill.sable.InstructionInlineShadowSource`

The info field for the first five of these attributes is a stream of `(offset:short, value:int)` pairs. The special value, -1, indicates no tag. The offset indicates the beginning of a range of bytecode instructions that have the given value. (Thus, no two adjacent pairs should have the same value). For example, the instruction tags listed in Listing 5.9 would be encoded as `(3,1)(5,2)(6,-1)`.

```
1: [ ] instr
2: [ ] instr
3: [1] instr
4: [1] instr
5: [2] instr
6: [ ] instr
7: [ ] instr
```

Listing 5.9: *Instruction tags on pseudo-bytecode*

The inlined shadow/source attribute encodes a list for each instruction, and so its encoded format is similar, but with variable length values. The info field for this attribute is a stream of `(offset:short, len:int, (shadow:int, source:int), ...)` tuples, where `len` is the count of `(shadow, source)` pairs succeeding it.

5.5.2 Class Attributes

In addition to the instruction tags described above, the following class attributes are optionally added by the compiler:

1. `ca.mcgill.sable.InstructionShadow_map`

2. `ca.mcgill.sable.InstructionSource_map`

The info field of each of these attributes is a stream of `(id: int, cp_index: int)` pairs. The `id` field corresponds to an instruction shadow or instruction source ID, and the `cp_index` is an index into the constant pool pointing to a human-readable string describing that ID. These attributes are not required of any of the analyses, and so are optional.

5.5.3 Method Attributes

1. `ca.mcgill.sable.ProceedMethod`

This attribute has no data. It identifies a method as representing the execution of a **proceed** statement within an around advice body. Its use is further described in section 6.1.

Chapter 6
Computing Metrics

This chapter describes the various dynamic analyses used to measure the runtime behaviour of AspectJ programs. Section 6.1 describes the algorithm used to assign an instruction kind to every bytecode execution for a run of a program. Section 6.2 describes the algorithm used to detect advice guard success and failure.

6.1 Tag Propagation

The compiler assigns instruction kind tags only to a subset of the bytecode instructions that may be executed. Other instructions must have the appropriate tags assigned dynamically during execution or analysis. This process of dynamic tag assignment is called *tag-propagation*, and it is explained below. Section 6.1.1 explains why it is necessary, section 6.1.2 defines the algorithm, and section 6.1.3 presents an example.

6.1.1 Why do we need to propagate tags?

It is generally not possible for the compiler to statically tag all of a program's bytecode instructions with an appropriate instruction kind tag at weave-time for several reasons. It may be that some of the code is not available to the weaver, such as library code or dynamically loaded classes; or it may be that the correct value of a

bytecode instruction's kind tag depends on runtime context. If every execution of a given bytecode instruction will be of the same instruction kind, then it could be statically tagged by the compiler; this is not true of every instruction. Consider the following examples:

1. The Java standard library: clearly a programmer may make use of the standard library in his code, either in the base program or in advice bodies. The AspectJ runtime library, however, also makes use of the standard library. For instance, the implementation of **cflow** in `ajc` uses the Java collections framework. When the `push` method on `java.util.Stack` is called by the programmer, it is clearly not AspectJ overhead; when the same method is called in the AspectJ runtime library by the **cflow** management code, however, it is. The appropriate instruction kind tag will therefore need to be assigned to the body of `Stack.push()` at runtime.

2. Consider the code in Listing 6.1. The method `BaseProgram.bar()` is called from three different contexts, and its execution should be considered of a different kind in each case. When called at site 1, its execution qualifies as BASE_CODE. When called from site 2, its execution qualifies as ASPECT_CODE, being that it is executed below an advice body. When called from site 3, it should be counted as ADVICE_TEST, being part of an advice guard. It is therefore insufficient to statically tag the body of `bar()` at weave-time. The correct instruction kind tag must be determined dynamically.

3. The special method `aspectOf`, defined in aspect classes, is used to acquire the appropriate aspect instance. Instructions invoking this method are woven into a join point shadow as part of advice execution. When this method is called, in preparation for the execution of a piece of advice, it should be counted as ADVICE_ARG_SETUP. However, this method can also be called directly by the programmer to acquire an aspect instance, perhaps in order to access a field or to invoke a non-static "normal" method on the aspect. Since the call has been made explicitly by the programmer, in this case, it should

```
class BaseProgram {
   public void foo() {
      // call site 1:
      BaseProgram.bar()
   }

   public static boolean bar() {
      return true;
   }
}

aspect TheAspect {
   before(): call(void BaseProgram.foo()) {
      // call site 2:
      BaseProgram.bar();
   }

   after(): call(void BaseProgram.foo())
         // call site 3:
         && if(BaseProgram.bar())
   {
      // do something
   }
}
```

Listing 6.1: *The correct instruction kind for the body of* `bar()` *depends on calling context.*

not be counted as AspectJ overhead. (A good example of this sort of direct call to `aspectOf` can be seen in the AspectJ implementation of the Observer design pattern in [HK02].)

These examples demonstrate the need for dynamic assignment of instruction kind tags to bytecode instructions. The following section describes how these tags are assigned.

6.1.2 How do we propagate tags?

The tag-propagation algorithm ensures that during the analysis of a program's execution, each bytecode execution is associated with a single instruction kind, appropriately indicating its overhead nature. While each bytecode execution has exactly one instruction kind, any given bytecode may have several different instruction kinds associated with it, over the course of the program's execution.

Dynamic and static tags

The tags assigned by the propagation algorithm are called *dynamic tags*, in contrast to the tags assigned to bytecode instructions by the compiler, which are called *static tags*. It is usually, but not always, the case that if a bytecode instruction has a static tag then it will be assigned a dynamic tag of the same value for each of its executions.

Current and default tags

A single bytecode instruction may have several different dynamic tags assigned to it over the run of a program. In the presence of recursion, it may need to retain multiple dynamic tags. The *current tag* is that which represents the instruction kind for the current execution.

Conceptually, every bytecode instruction can be thought of as having a stack of instruction kind tags per thread of execution. If the bytecode instruction has a static tag, assigned by the compiler, then that tag is the bottom value on the stack.

Otherwise, the bottom value is the special tag NO_TAG. The value on the top of the stack is the current tag. If, during analysis, an instruction's current tag is NO_TAG, then the *default tag*, BASE_CODE, will be substituted.

Caller and propagated tags

Tag-propagation occurs on method calls. The body of the called method may contain untagged instructions, which should be tagged, or it may contain tagged instructions, some of whose tags should be overwritten. The *caller tag* is the kind tag at the call site. The *propagated tag* is the tag that is assigned to untagged instructions in the method body, or which may overwrite certain tagged instructions in the method body. The propagated tag is determined by the caller tag.

Advice depth counter

around advice may apply to join points within base code or it may apply to join points within aspect code. It is important to distinguish between these two cases so that the correct tag can be propagated on execution of a **proceed** statement. The *advice depth counter* indicates whether a **proceed** statement corresponds to the return of execution to base code or to aspect code, and thus what the correct value is for the propagated tag.

The advice depth counter is incremented every time an advice body is entered and decremented every time an advice body is exited. In the case of **around** advice, this means that it is also decremented on every call to **proceed** and incremented on return. A value greater than zero indicates that execution is currently within advice.

The **proceed** statement is implemented in both compilers as a method call to a special *proceed method*. The proceed method is identified by the compiler and is given the class file attribute `ca.mcgill.sable.ProceedMethod`. Entry into a proceed method corresponds to the execution of the **proceed** statement in an around advice body. Exit from this method corresponds to the return of execution to the advice. The advice depth counter is adjusted appropriately on entry to and

```
function propagate(caller tag, advice depth):
   if caller tag ∈ {ADVICE_EXECUTE, INTERMETHOD }:
      propagated tag ← ASPECT_CODE
   else if caller tag = AROUND_PROCEED :
      if advice depth > 1:
         propagated tag ← ASPECT_CODE
      else:
         propagated tag ← BASE_CODE
   else:
      propagated tag ← caller tag
   return propagated tag
```

Listing 6.2: *The propagation function*

exit from this method.

The propagation function

The propagation function maps the caller tag to the propagated tag. It is called by the propagation algorithm on method entry to determine the correct value of the propagated tag. It is defined in Listing 6.2.

For most caller tags, the propagation function will return a propagated tag of the same value. ADVICE_EXECUTE and INTERMETHOD, however, return ASPECT_CODE, since they apply to invoke instructions that call user-defined code, not additional overhead. Likewise, the AROUND_PROCEED tag propagates either BASE_CODE or ASPECT_CODE, depending on the value of the advice depth counter (as described above), for the same reason. All other tags, for which *propagate(caller tag, advice depth)* = *caller tag*, for all values of *advice depth*, are called *self-propagating tags*.

The replacement function

On entering a method, some instructions may already have tags, either static tags assigned by the compiler, or, if the method is being called recursively, dynamic tags previously assigned by the propagation algorithm. The new tag to be pushed on each instruction's instruction stack—that is, the value of each instruction's current tag for the execution of this method—is determined by the *replacement function*, shown in Listing 6.3. Some of the special cases in the function are explained below.

1. Lines 2-3: All BASE_CODE instructions executing below ASPECT_CODE instructions are to be considered ASPECT_CODE, e.g. in the case of a base code method being called from an advice body.

2. Lines 4-5: The special aspect methods `hasAspect` and `aspectOf` can be called by the user from base code, as explained in example 3 in section 6.1.1. In this case, they should be counted as ASPECT_CODE. The BASE_CODE tag will not overwrite the static ASPECT_CODE tag in this case.

3. Lines 6-8: Otherwise, any call made from self-propagating overhead code will propagate the caller tag. For example, a call to `aspectOf` (statically tagged ASPECT_CODE) from ADVICE_ARG_SETUP instructions, or a call to any normal method from an **if** pointcut or from within **cflow** management instructions.

Note that if the current tag is NO_TAG, this is interpreted as the default tag, BASE_CODE, so those conditions that match when the current tag is BASE_CODE also match "untagged" instructions.

The propagation algorithm

Tag-propagation consists of two basic steps:

1. calculating the propagated tag on method entry (the propagation function)

2. calculating the correct dynamic tag for each bytecode instruction execution (the replacement function)

Listing 6.4 defines the propagation algorithm.

```
function replace(current tag, propagated tag):
    if propagated tag = ASPECT_CODE and current tag = BASE_CODE :
        new tag ← propagated tag
    else if propagated tag = BASE_CODE and static tag = ASPECT_CODE
        new tag ← ASPECT_CODE
    else if current tag ∈ {ASPECT_CODE, BASE_CODE }
          and propagated tag is self-propagating:
        new tag ← propagated tag
    else:
        new tag ← current tag
    return new tag
```

Listing 6.3: *The replacement function*

6.1.3 A simple propagation example

The code in Listing 6.5 is of a simple AspectJ program that will be used to illustrate the behaviour of tag propagation. When compiled, two class files are produced: one corresponding to the `Main` class, the other to the `MainAspect` aspect. Their bytecode is shown in listings 6.6 and 6.7, respectively. For each bytecode listing, the static instruction kind tags assigned by the compiler are indicated.

Figure 6.1 illustrates, step-by-step, tag propagation at work. It consists of several subfigures, each of which represents a different point in the execution/analysis. The current point of execution/analysis is indicated by the small arrow, absent in the first subfigure. The current kind tag for each instruction is indicated to the instruction's left. The tag is highlighted when the propagation algorithm has resulted in a change. As each bytecode instruction is executed, its instruction kind is counted. The box at the bottom right of each subfigure contains the current tally for each kind. Not every instruction is shown, but the tallies account for them anyway. In the case where a call is made into the standard library, and an indeterminate number of instructions are executed, this is indicated by appending a $+$ to the tally.

```
On Method Entry:
   // update advice depth counter
   if caller tag = ADVICE_EXECUTE :
      increment advice depth
   if method is proceed method:
      decrement advice depth

   // compute the propagated tag
   propagated tag ← propagate(current tag, advice depth)

   // compute the new dynamic tag for each instruction in method
   for each instruction in method
      new tag ← replace(current tag, propagated tag)
      push new tag on instruction's tag stack

On Method Exit:
   decrement/increment advice depth if required
   pop tag stack for all instructions in method
```

Listing 6.4: *The propagation algorithm*

The arrow in `aspectOf` indicates the branch. In this execution, the branch is not taken, and only the executed instructions are shown. The fact that all instructions in the class are tagged on method entry is indicated by the tag to the left of the ellipsis.

```
public class Main {
   public static void main(String[] args) {
      Main m = new Main();
      m.sayHello();
   }

   public void sayHello() {
      System.out.println("Hello.");
   }
}

aspect MainAspect {
   before(): call(void Main.sayHello()) {
      System.out.println("before sayHello");
   }
}
```

Listing 6.5: *A simple AspectJ program*

```
public static void main(String[] arg0)
             new Main
             dup
             invokespecial Main.<init>()
ADVICE_ARG_SETUP  invokestatic MainAspect.aspectOf()
ADVICE_EXECUTE    invokevirtual MainAspect.before$0()
             invokevirtual Main.sayHello()
             return

public void sayHello()
             getstatic java/lang/System.out
             ldc "Hello."
             invokevirtual java/io/PrintStream.println(String)
             return

public void <init>()
             aload_0
             invokespecial java/lang/Object.<init>()
             return

static void <clinit>()
             return
```

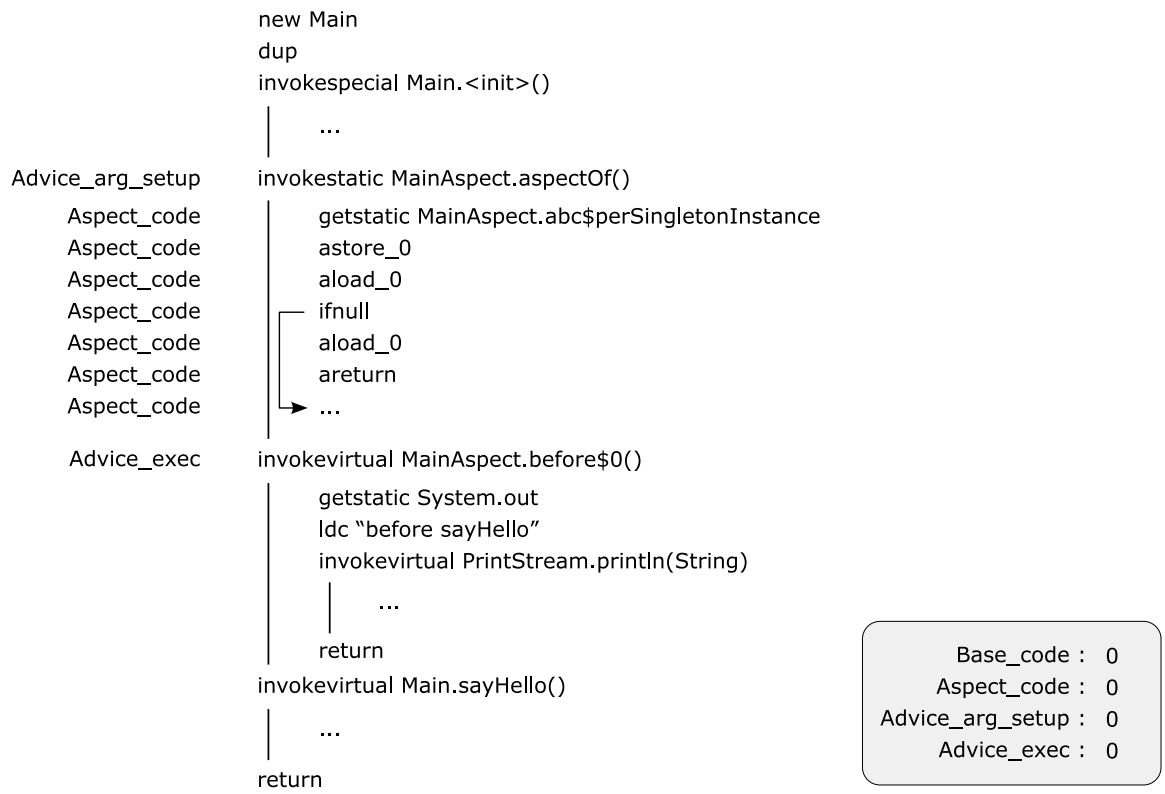
Listing 6.6: *Main.class*

```
public final void before$0()
             getstatic java/lang/System.out
             ldc "before sayHello"
             invokevirtual java/io/PrintStream.println(String)
             return

public void <init>()
             aload_0
             invokespecial java/lang/Object.<init>()
             return

public static MainAspect aspectOf()
      ASPECT_CODE  getstatic MainAspect.abc$perSingletonInstance
      ASPECT_CODE  astore_0
      ASPECT_CODE  aload_0
      ASPECT_CODE  ifnull -> 10
      ASPECT_CODE  aload_0
      ASPECT_CODE  areturn
      ASPECT_CODE  new org/aspectj/lang/NoAspectBoundException
      ASPECT_CODE  dup
      ASPECT_CODE  ldc "MainAspect"
      ASPECT_CODE  getstatic MainAspect.abc$initFailureCause
      ASPECT_CODE  invokespecial org/aspectj/lang/NoAspectBoundException
                          .<init>(String;Throwable)
      ASPECT_CODE  athrow
```

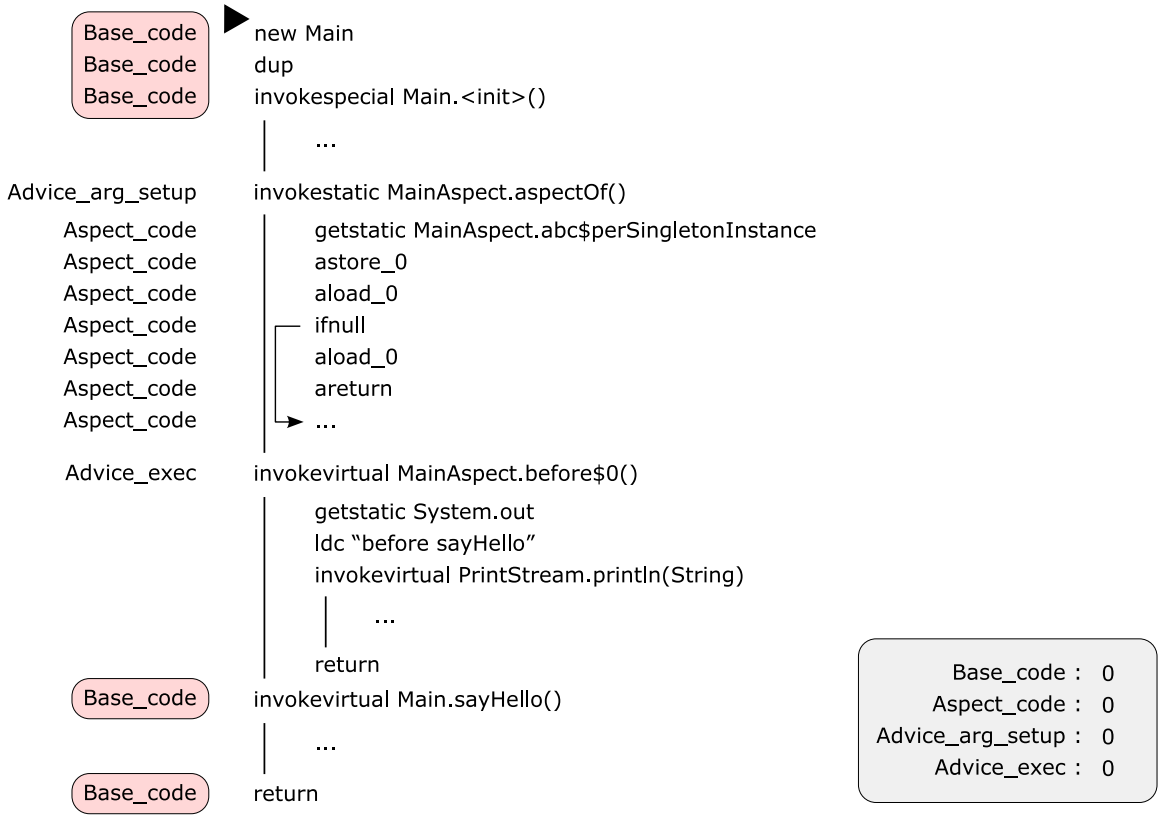
Listing 6.7: *MainAspect.class*



(a) static tags

Figure 6.1: *Propagation example*

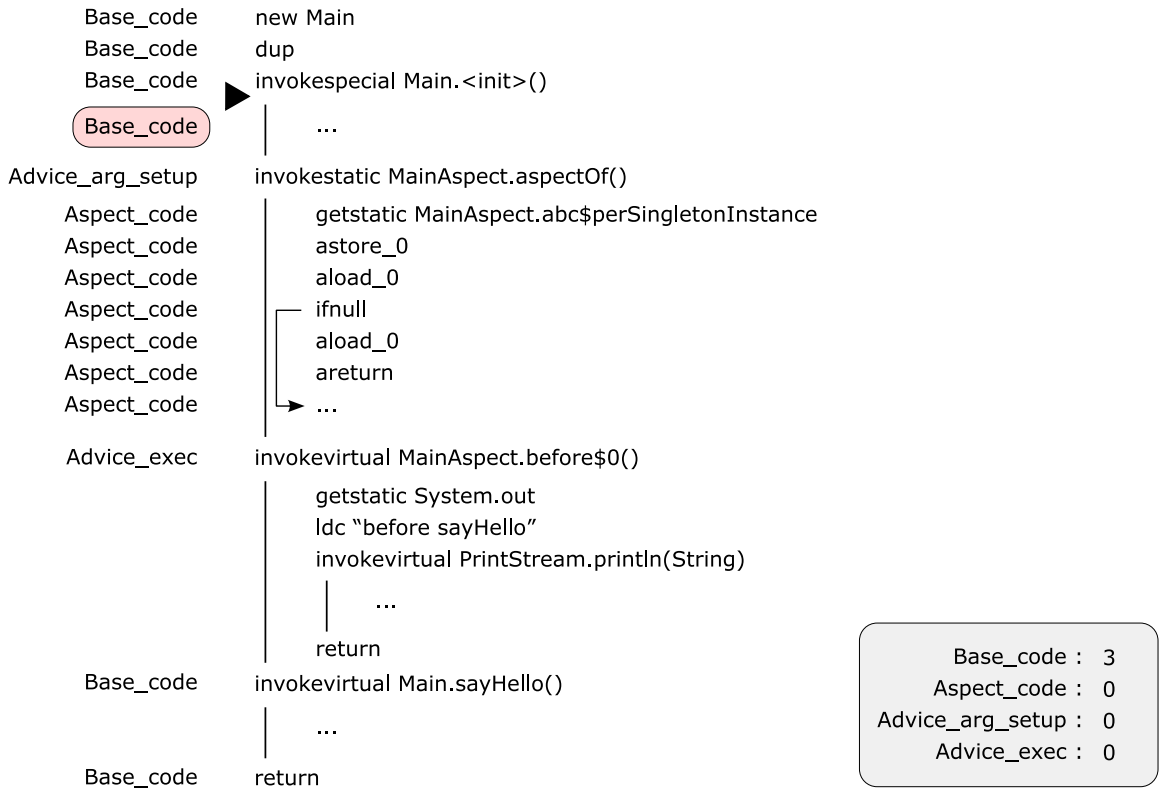
Figure 6.1(a) shows the state of the program before execution. The tags listed are the static tags assigned to the bytecode instructions by the compiler.



(b) execution enters `main`, default tag assigned

Figure 6.1: *Propagation example (cont.)*

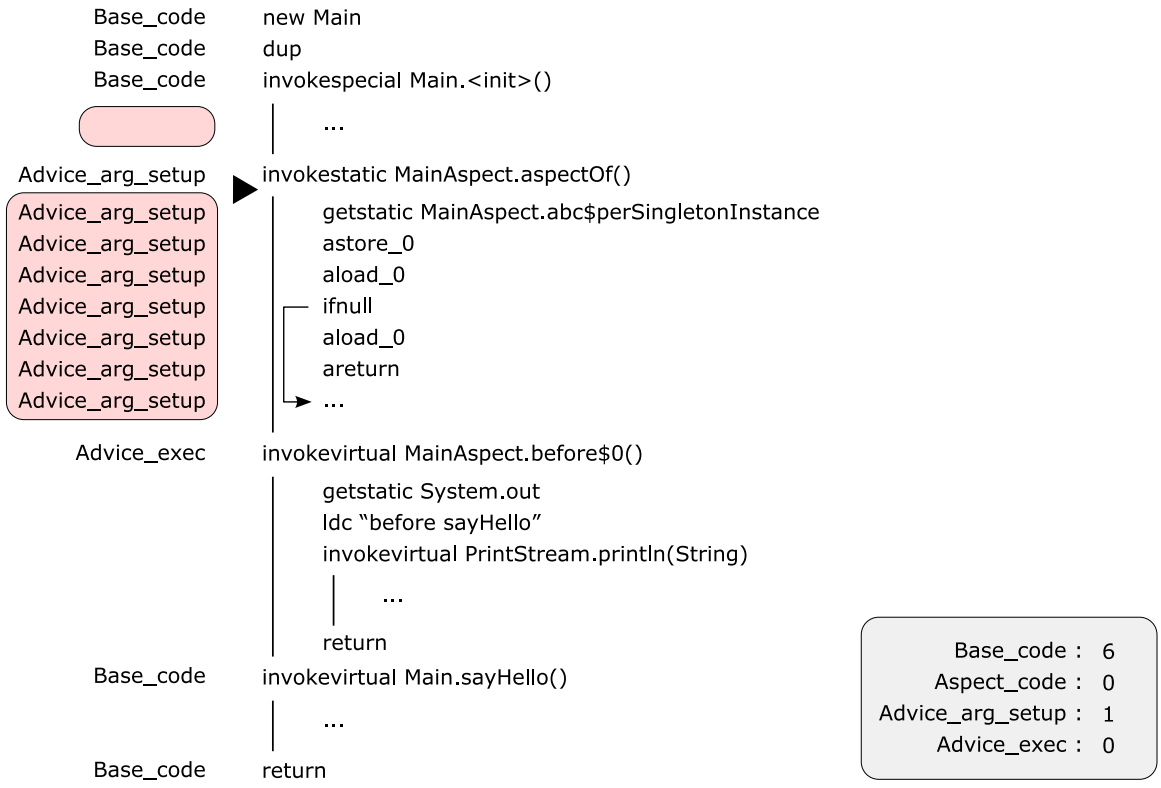
On entering the body of `Main.main`, in Figure 6.1(b), all untagged instructions are assigned the default tag BASE_CODE. (Equivalently, their current tag is NO_TAG, as described in section 6.1.2, which is interpreted as the default tag as each instruction is counted.)



(c) BASE_CODE propagated into constructor

Figure 6.1: *Propagation example (cont.)*

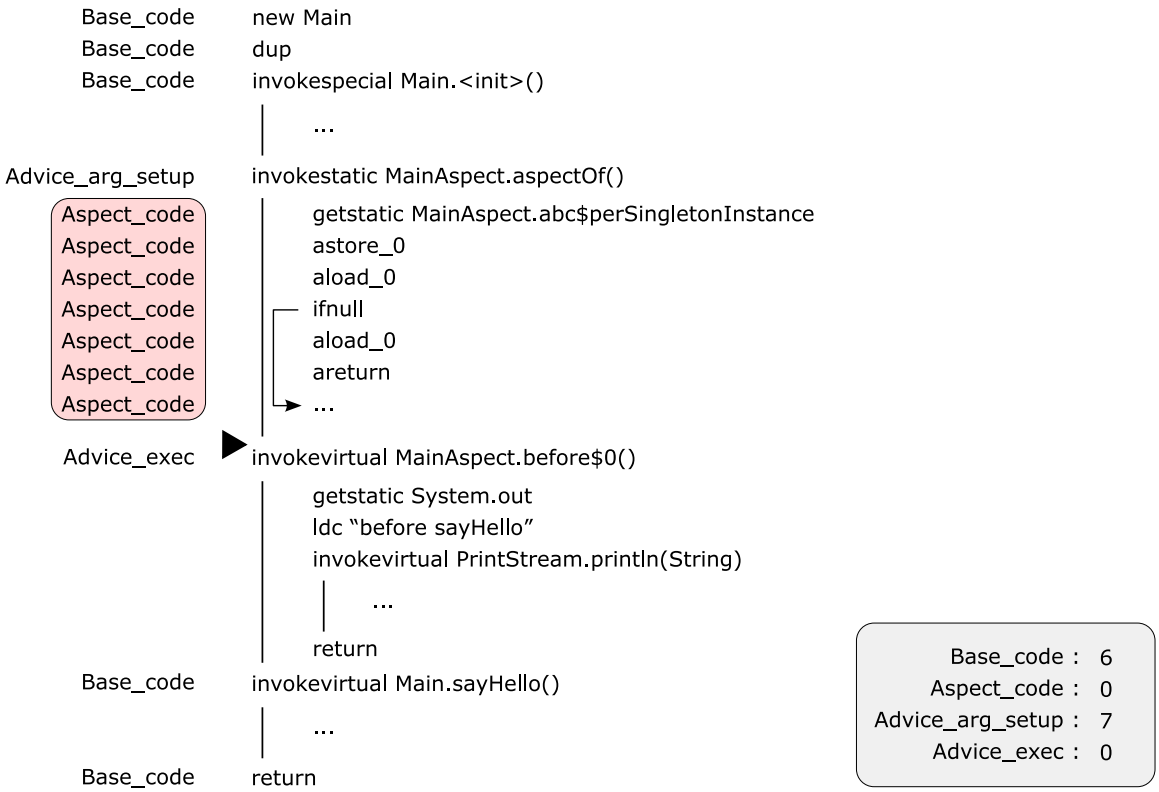
In Figure 6.1(c), the executions of the first three bytecode instructions have been counted. The tally for BASE_CODE has been updated appropriately. Execution has currently entered the body of the constructor for `Main`, which is untagged. The caller tag in this case is BASE_CODE, which, according to the propagation function, maps to itself as the propagated tag. Consequently it is pushed to all of the instructions in the constructor (the body of which is not shown here, but indicated with an ellipsis.)



(d) ADVICE_ARG_SETUP propagated into `aspectOf`, overwrites static tag

Figure 6.1: *Propagation example (cont.)*

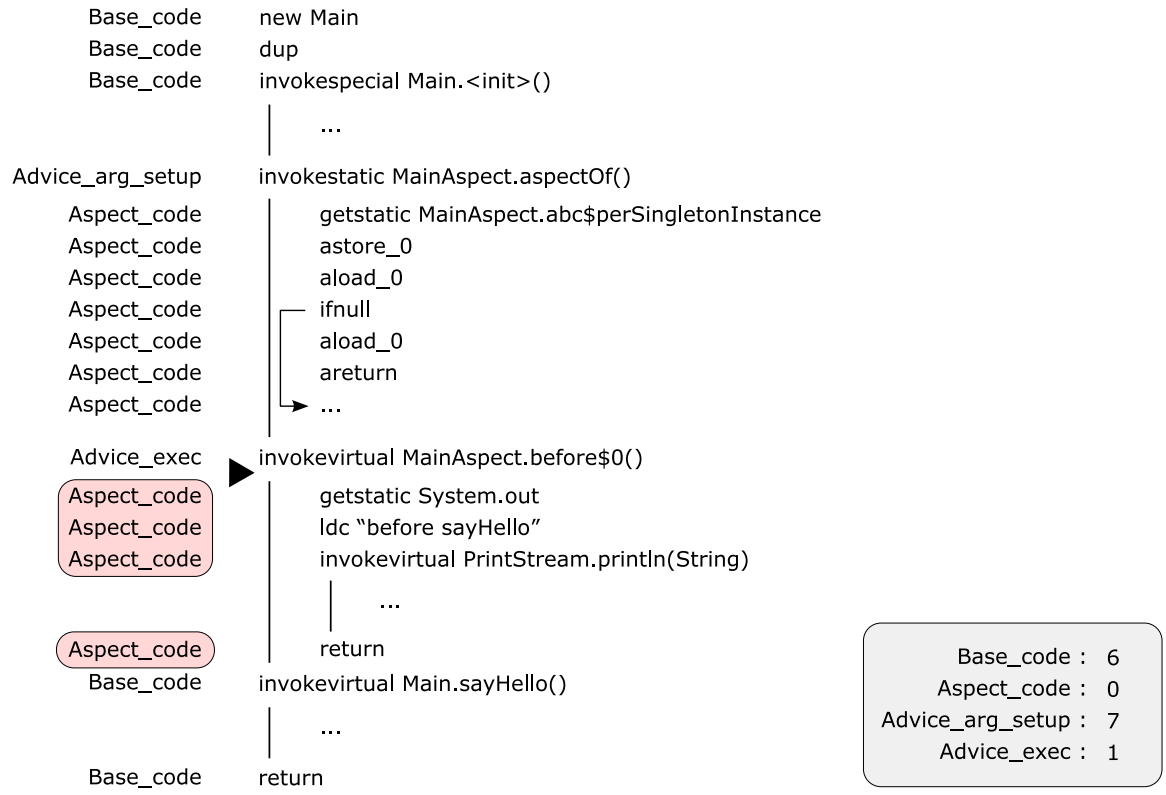
In Figure 6.1(d), execution has reached a call site with the static kind tag ADVICE_ARG_SETUP. This call to `MainAspect.aspectOf` was inserted by the weaver to acquire the aspect instance in preparation for the execution of the before advice. The caller tag here is ADVICE_ARG_SETUP, which also propagates itself. The body of the `aspectOf` method, however, has already been statically tagged ASPECT_CODE, but according to the replacement function, a self-propagating tag, of which is ADVICE_ARG_SETUP, can overwrite a static ASPECT_CODE tag. Each instruction in the body of the `aspectOf` method is therefore assigned the ADVICE_ARG_SETUP tag.



(e) `aspectOf` instruction tag stacks popped on leaving method

Figure 6.1: *Propagation example (cont.)*

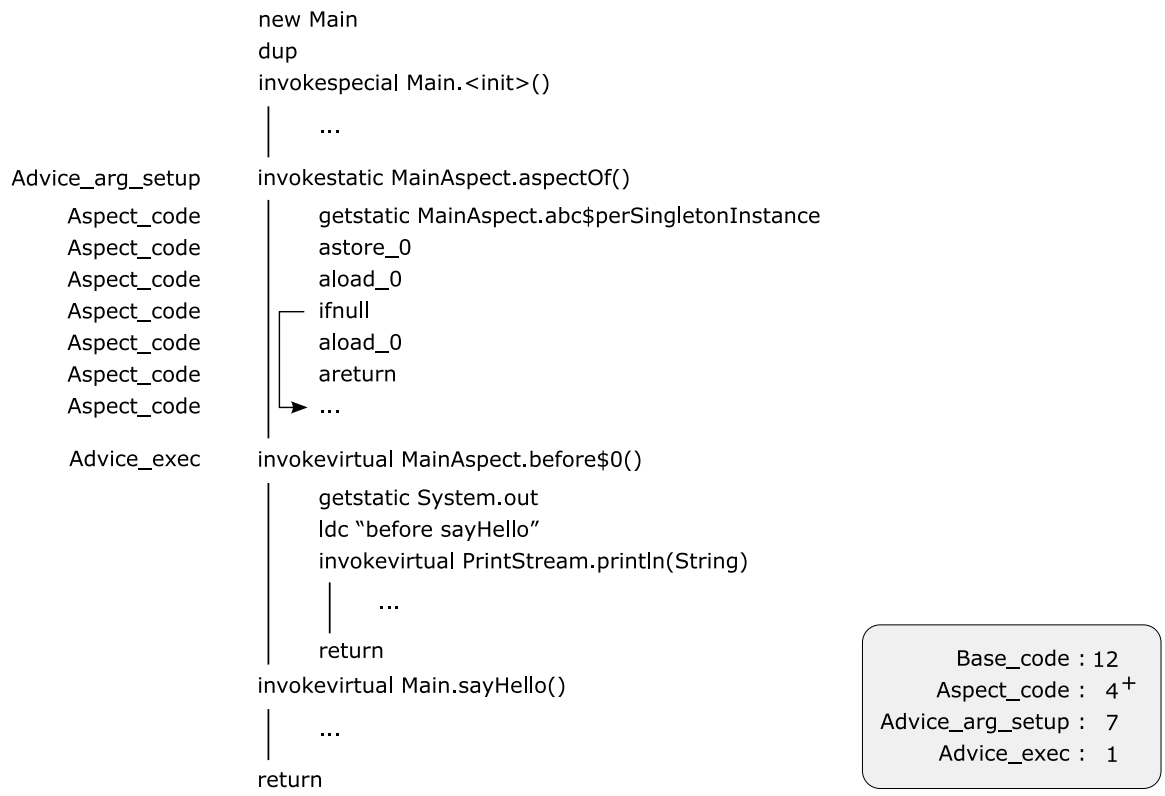
In Figure 6.1(e), execution has returned from the `aspectOf` method. On leaving the method body, the current tag for each instruction is reset to its previous value, ASPECT_CODE. That is, the stack of dynamic tags for each instruction is popped.



(f) Execution of the advice body

Figure 6.1: *Propagation example (cont.)*

The aspect instance having been acquired, the advice body itself is executed next. In Figure 6.1(f), execution has entered the advice body, `MainAspect.before$0`. In this case, the propagated tag differs from the caller tag. The propagation function indicates that ADVICE_EXECUTE propagates ASPECT_CODE, which is assigned to the untagged instructions in the advice body.



(f) Execution of `main` finished

Figure 6.1: *Propagation example (cont.)*

In Figure 6.1(f), execution of `main` has completed, and all dynamic instruction kind tags have been popped. The final tally of executions is given. For this example, the + for ASPECT_CODE indicates however many instructions were executed by the `println` method.

6.1.4 Propagation in the presence of around advice

As indicated in section 6.1.2, **around** advice requires the propagation algorithm to maintain an advice depth counter. This section briefly presents a simple example showing tag propagation in the presence of **around** advice.

Listing 6.8 is an extension to the program in Listing 6.5. The aspect defines an additional piece of around advice which executes around calls to methods named

```
public class Main {
   public static void main(String[] args) {
      Main h = new Main();
      h.sayHello();
   }

   public void sayHello() {
      System.out.println("Hello.");
   }
}

aspect MainAspect {
   before(): call(void Main.sayHello()) {
      System.out.println("before sayHello");
   }

   void around(): call(* *.println(..)) {
      System.out.print("around println\n");
      proceed();
   }
}
```

Listing 6.8: *New around advice*

"println". It prints out a short message to `System.out` and proceeds with the execution of the original call to `println`. Notice that it applies at two places: to the call to `System.out.println` in the method `Main.sayHello()`, and to the call to `System.out.println` in the **before** advice body. The propagation algorithm will treat these two cases differently.

When execution reaches the call to `sayHello`, it will proceed to the before advice body defined in `MainAspect`. On execution of the method `before$0`, the value of the *aspect code depth* counter will be incremented from 0 to 1. The before advice body calls `System.out.println`, which is advised by the around

advice, which is therefore executed. On execution of the around advice, the *aspect code depth* counter is again incremented, now to 2. Since the value of this counter is greater than 1 when execution reaches the **proceed** statement, ASPECT_CODE is propagated to the **proceed** call instead of BASE_CODE, since it is proceeding back into an advice body, not back into the base program. The counter is decremented when **proceed** is called, and incremented on return.

In contrast, when execution reaches the call to `println` in `sayHello`, the value of the *aspect code depth* counter is 0. This gets incremented to 1 on execution of the around advice body, and so when the call to **proceed** is reached, since *aspect code depth* ≤ 1, BASE_CODE will be propagated back to the call to `println`.

6.1.5 Propagation and advice inlining

The `abc` compiler implements a number of optimizations that that the `ajc` compiler does not. Among these is advice inlining. Advice inlining results in the bodies of small advice being inlined directly into the advised join point shadow, in place of the original invoke instruction. For example, Listing 6.9 lists the bytecode resultant from compiling the program in Listing 6.5 with this optimization turned on.

Inlining advice bodies has several consequences:

1. It requires the compiler to statically propagate the instruction kind tag on the invoke appropriately to the body of the method being inlined.

2. It complicates the identification of advice bodies and **proceed** statments. Previously, execution of an advice body or **proceed** statement corresponded to method calls. In the presence of inlining, this is no longer necessarily true. In order to identify inlined advice and proceed methods, additional metada is required. This is described in more detail in section 5.4.

3. It requires the introduction of new instruction kind tags to identify inlined advice and proceed methods.

Static propagation for the inlining of advice bodies is very simple: every untagged instruction in the body being inlined is assigned the tag INLINED_ADVICE,

```
public static void main(String[] arg0)
                [ : ]  new Main
                [ : ]  dup
                [ : ]  invokespecial Main.<init>:()V
   ADVICE_ARG_SETUP [1:3]  invokestatic MainAspect.aspectOf()
   ADVICE_ARG_SETUP [1:3]  pop
     INLINED_ADVICE [ : ]  getstatic java/lang/System.out
                ([1:3])
     INLINED_ADVICE [ : ]  ldc "before sayHello"
                ([1:3])
     INLINED_ADVICE [ : ]  invokevirtual java/io/PrintStream.println(String)
                ([1:3])
                [ : ]  invokevirtual Main.sayHello()
                [ : ]  return
```

Listing 6.9: *`main` from program in Listing 6.5, compiled with `abc` with advice inlining enabled*

which is equivalent to ASPECT_CODE. The methods corresponding to **proceed** statements and **if** pointcuts can also be inlined by the same facility; they are distinguished from advice bodies by the caller tag. In the case of **if** pointcut methods, the caller tag (ADVICE_TEST, or one of the other dynamic residue kind tags) is propagated to the untagged body instructions. In the case of a proceed method, INLINED_PROCEED is propagated. INLINED_PROCEED is equivalent either to BASE_CODE or to ASPECT_CODE, depending on the value of the advice depth counter. As explained in section 5.4, the instructions in the body of the proceed method are also tagged with the proceed tag and have their inline count tag incremented.

Maintaining the advice depth counter is the only complication to dynamic tag-propagation raised by advice inlining. When there is no advice inlining, entry to and exit from an advice body corresponds to entry to and exit from a method. Likewise for the execution of **proceed** statements. When advice bodies can be inlined, however, any instruction may represent entry to or exit from an advice or proceed

body. Furthermore, although it is not possible given the code generation strategies of current AspectJ compilers (because an advice invocation is always preceded by a call to `aspectOf`, and therefore can't be the first instruction in an advice body), it is conceivable (if advice bodies were compiled as static methods, for example) for a single instruction to represent entry into multiple advice bodies. Single instructions can currently, however, represent exit from multiple advice or proceed bodies. A strategy for detecting entry to and exit from inlined advice and proceed bodies is described below.

To detect the execution of an inlined advice body, we keep track of the last instruction executed in this method (for the first instruction of a method, the previous instruction is null.)

If the previous instruction is null, and the current instruction has a non-zero inline count (the number of shadow/source pairs in the instruction's list of inlined shadow/source pairs), then execution has entered at least one new advice body. The advice body, or bodies, that are currently being executed are those whose shadow/source IDs are in the current instruction's inlined shadow/source list.

If the current instruction's inline shadow/source list matches the previous instruction's inline shadow/source list, then we are still executing the same advice bodies. If it differs, then we may have entered a new advice body, left an advice body, or both left and entered an advice body.

To determine which advice executions have ended and which have begun, we compare the inline shadow/source lists of the current and previous instructions. We eliminate the common tail from the lists; the remainder of the first list is the list of advice bodies we have left, and the remainder of the second list is the the list of the advice bodies we have entered.

For example, let instruction `prev` have the list of IDs `[5, 3, 2, 1]` and let instruction `curr` have the list of IDs `[5, 4, 2, 1]`. The common tail of both lists is the list `[2, 1]`, indicating that both instructions are being executed within advice body execution 2, which in turn is being executed within advice body execution 1. The remaining lists are `[5, 3]` and `[5, 4]`. This means that the the execution of `curr` corresponds to having finished executing advice 5 and 3, and

begun executing advice 4 and 5.

To detect entry to an inlined proceed body, we keep track of the inline count of the previous instruction. If the current instruction has a proceed tag, and its inline count is greater than that of the previous instruction, and the current instruction doesn't represent entry to or exit from an advice body (that is, the inlined shadow/source list is unchanged), then a new proceed body has been entered, and the advice depth counter is updated appropriately.

There are a couple more complex cases to handle. A single instruction could represent entry to multiple proceed bodies (in the case of **around** advice on **around** advice), or entry to both an advice body and a proceed method (**around** advice the first statement of which being a **proceed**), or simultaneous exit from and entry to a proceed method (contiguous **proceed** statements). In the first case, the inline count will differ by more than one, and the advice depth counter should be updated appropriately. In the second case, advice entry or exit will first be detected. When this happens, the inline count of the current instruction should first be decremented by the number of advice bodies entered, and incremented by the number of advice bodies exited, before it is compared to that of the previous instruction. The difference after this adjustment indicates the number of proceed bodies entered. The final case is not detected, because it does not affect the value of the advice depth counter.

Given this strategy for detecting advice and proceed entry and exit events, the propagation algorithm need only be modified to update the advice depth counter on these events, in addition to method entry and exit.

6.2 Advice Guard Identification

As explained in chapter 2, instead of actually observing the execution of a program for join points matched by a given pointcut and executing advice appropriately when found, which would be a very inefficient way to implement the join point

model, the AspectJ compiler (either `ajc` or `abc`) instead performs a kind of partial evaluation, weaving advice execution instructions into the join point shadows whose execution gives rise to join points matched by particular pointcuts. Since it is not the case that the execution of a particular shadow will always result in a matching join point, it is often necessary to compile in a runtime check that determines dynamically whether the join point corresponding to a particular execution of the shadow matches. These checks are called dynamic residues, or guards.

A naive implementation of AspectJ might weave in guards at every join point shadow, while an optimized implementation could use more sophisticated static analysis to eliminate the need for most guards.

The advice execution metric requires us to have a way to identify when we have entered a guard, and to determine whether that guard has evaluated to true (resulting in the execution of its advice) or false.

6.2.1 The simple case

When the compiler weaves advice execution instructions that require a guard into a join point shadow, the instructions implementing that guard are tagged with the ADVICE_TEST instruction kind tag. (Correspondingly, if the compiler is not weaving in an advice execution proper, but rather **cflow** management instructions with a guard, then that guard is differentiated by being tagged CFLOW_TEST. Likewise for PEROBJECT_ENTRY_TEST. The advice execution metric, however, only considers advice guards.)

Since a given piece of advice can be applied to a particular join point only once, a guard is uniquely identified by a pair of shadow and source identifiers.

In order to implement the advice execution metric, we must identify three events: entering a guard; leaving a guard and executing the corresponding advice (that is, guard success); and leaving a guard without executing its corresponding advice (that is, guard failure.)

For each instruction execution, we keep track of the previous instruction's kind, shadow, and source tag.

For each pair of instruction executions, there are 6 possible transitions:

1. not in a guard → in a guard
2. still in the same guard
3. in a guard → guard failure → in a different guard
4. in a guard → guard success → advice execution
5. in a guard → guard failure → advice execution
6. in a guard → guard failure → something else

We identify these cases as follows:

1. If the current instruction's kind tag is ADVICE_TEST, and the previous instruction's kind tag is not, then we have entered a new guard, identified by the current instruction's shadow and source IDs.
2. If both the current and previous instructions' kind tags are ADVICE_TEST, and their source and shadow IDs match, then we have not finished executing the current guard, and do nothing.
3. If both the current and previous instructions' kind tags are ADVICE_TEST, but either their source or shadow IDs do not match, then we have entered a new guard, and the previous guard failed.
4. If the current instruction's kind tag is ADVICE_EXECUTE, and the previous instruction's kind tag is ADVICE_TEST, and if the source and shadow IDs of the current and previous instructions match, then the guard identified by the shadow and source IDs was successful.
5. Correspondingly, if the shadow and source IDs do not both match, the guard has failed.

6. If the previous instruction's kind tag is ADVICE_TEST, and the current instruction's kind tag is anything but ADVICE_TEST or ADVICE_EXECUTE, the previous guard failed.

6.2.2 Advice guard identification and advice inlining

Guard identification in the presence of advice inlining is more complicated. As described in section 6.1.5, the compiler may optimize the generated code by inlining small advice bodies directly into the advised join point shadows. To accommodate this inlining, the guard detection algorithm is extended in a manner similar to the extension of the tag-propagation algorithm.

Once we can detect the execution of an inlined advice body (see section 6.1.5), we need to be able to associate a guard with an inlined advice body. An advice body, even when inlined, will still be identified by the shadow/shadow ID pair of its original invoke instruction. We make the association as follows.

Guard instructions will have a shadow/source ID. The guard itself may have been inlined, and its inline count is equivalent to the size of its inlined shadow/-source ID list.

For the first instruction executed after a guard, we compare the shadow/source ID of the guard to the shadow/source ID in the instructions inlined shadow/source ID list at the position corresponding to the inline count of the guard. (The inline count is usually 0, so we usually look at the first pair in the list.) If the shadow/-source IDs match, then the advice execution corresponds to the guard.

Consider the following examples. Several execution traces are given, some for guard successes and some for guard failures. For simplicity, the shadow/source ID pair is presented as a single number. The instruction kind tag is listed first, followed by the shadow/source ID, followed by the inlined shadow/source ID list.

In this example, the guard has not been inlined. The shadow/source ID of the guard matches the inlined shadow/source ID of the advice execution. Therefore, the guard has succeeded.

```
ADVICE_TEST     [1] [ ] instr
INLINED_ADVICE  [ ] [1] instr
```

In the next case, although advice is executed immediately after a guard, it is not the advice associated with that guard. Since the advice that is associated with the guard was not executed, the guard failed.

```
ADVICE_TEST     [1] [ ] instr
INLINED_ADVICE  [ ] [2] instr
```

In this final example, the guard and the inlined advice body have both been inlined. The inline count of the guard instructions is 1. Therefore, the shadow/-source ID of the guard must be compared to the shadow/source ID at position 1 (the second one from the end) in the inlined shadow/source ID list for the next advice instruction. In this case, it matches, and the guard succeeds.

```
ADVICE_TEST     [1] [3] instr
INLINED_ADVICE  [ ] [3,1] instr
```

Chapter 7
Experimental Results

In this chapter, experimental results and analyses for a number of benchmarks are presented, each for a number of compiler configurations. These benchmarks span a variety of uses for AspectJ, each exercising different features of the language. They have been collected from a variety of public sources.

As discussed in chapter 1, it has been generally believed that AspectJ programs should show little runtime overhead. Although this is true of some of the benchmarks analyzed, others show very large runtime overheads. These overheads are, for the most part, due to two AspectJ features: **cflow** pointcuts and **around** advice. Lesser overheads are also observed for use of **thisJoinPoint** and aspect instance binding.

Section 7.1 presents results for the benchmarks as compiled by `ajc` 1.2. Those benchmarks that show relatively little runtime overhead are examined first, followed by those showing significant runtime overheads. These overheads are analyzed and explained, and potential solutions for reducing them are suggested.

Section 7.2 presents results generated with `abc`, an optimizing AspectJ compiler, the development of which was prompted, in part, by early versions of this work. Some of its optimization strategies are explained, and its results are compared with those for `ajc`.

Finally, some results for `ajc` 1.2.1, which incorporates some of the optimizations suggested here, and the latest development version of `abc` (as of August, 2005) are

presented in section 7.3.

To produce the **J* trace files used for dynamic metrics calculation, all benchmarks were run in Sun's Java 2 Runtime Environment, Standard Edition version 1.4.0_04-b04. All execution time results were produced in the same JRE on a 1.80GHz Intel Pentium 4 with 1GB of RAM running Linux 2.4.20. (More recent versions of the JRE could not be used for calculating dynamic metrics due to flaws in the JVMPI implementation of the VM.)

7.1 ajc Results

This section presents results for the benchmarks compiled with `ajc` 1.2. Section 7.1.1 presents the overall data. Section 7.1.2 examines the results for those benchmarks that show relatively little overhead. Section 7.1.3 presents the results for those benchmarks that show significant overhead, and explains the nature of that overhead.

7.1.1 Overall Data

An overview of the key data for benchmarks compiled with `ajc` 1.2 is presented in Tables 7.1 and 7.2. Table 7.1 presents execution time measurements and general Java dynamic metrics. Table 7.2 presents the key AspectJ-specific dynamic metrics. For the tag mix metrics in this table, only the relevant instruction kind tags are listed.

Data for the following benchmarks are presented. (More detailed descriptions of the benchmarks are provided in the subsequent discussion.)

dcm-sim: Calculates a dynamic coupling metric for a base program.

prodline: Implements a product line of related graph algorithms.

bean: Aspects add Java Bean functionality to a base program.

lod-sim: Tests a base program for correct Law of Demeter object form. Shows high **cflow**-related overhead.

figure: Simulation of a simple figure editor. Shows high **cflow**-related overhead.

nullcheck-sim: Tests a program for the "on error condition, return `null` from method" anti-pattern. Shows high **around**-related overhead.

	dcm-sim	*prodline*	*bean*	*lod-sim*	*figure*	*nullcheck-sim*
PROGRAM SIZE (APPLICATION ONLY)						
Classes Loaded	55	28	5	63	15	138
Instructions Loaded	16553	3289	560	27187	594	8577
Code Coverage (%)	45	60	67	58	64	41
PROGRAM SIZE (WHOLE PROGRAM)						
Classes Loaded	393	325	375	385	301	456
Instructions Loaded	112475	83823	99947	120933	75258	101011
EXECUTION TIME MEASUREMENTS (WHOLE PROGRAM)						
# instr. (million bytecodes)	3642	2213	158	4814	2871	1938
Total time - client (sec)	10.65	1.85	1.93	136.44	20.17	9.01
JIT time - client (sec)	0.36	0.12	0.08	0.64	0.08	0.11
GC time - client (sec)	0.52	0.03	0.03	91.88	0.10	2.17
Slowdown vs. handcoded(×)			**1.05**		**37.35**	**3.92**
Time - client_noinline (sec)	11.31	2.15	2.13	97.93	30.12	10.10
Slowdown vs. handcoded (×)			1.15		25.18	4.30
Time - interpreter (sec)	72.64	21.94	6.79	201.10	136.86	67.23
Slowdown vs. handcoded (×)			1.20		27.63	3.86
EXECUTION SPACE MEASUREMENTS (WHOLE PROGRAM)						
Mem. Alloc. (million bytes)	367	30	110	1004	374	1535
Obj. Alloc. Density (per kbc)	2.13	0.35	21.60	7.30	5.58	19.34
#Garbage Collections	373	38	144	1104	489	1526

Table 7.1: *Overall data: general metrics*

	dcm-sim	*prodline*	*bean*	*lod-sim*	*figure*	*nullcheck-sim*
ASPECTJ METRICS SUMMARIZING OVERHEAD						
AspectJ Overhead % (whole)	**4.82**	**0.73**	**14.25**	**97.69**	**95.78**	**50.15**
#overhead/#advice (whole)	0.05	0.01	0.18	49.25	229.17	19.49
#advice/#total (whole)	0.94	0.99	0.79	0.02	0.00	0.03
AspectJ Runtime Lib % (whole)	3.09	0.01	0.00	94.01	91.67	3.88
AspectJ Overhead % (app)	16.66	11.25	39.48	98.01	83.43	50.29
#overhead/#advice (app)	0.21	0.13	0.92	146.87	50.33	19.49
ASPECTJ TAG MIX FOR ALL INSTRUCTIONS (WHOLE PROG.) (%)						
BASE_CODE	1.41	0.06	7.16	0.32	3.80	47.27
ASPECT_CODE	93.77	99.21	78.59	1.98	0.42	2.57
INTERMETHOD		0.23	0.51			
INTERFIELDINIT		0.09	1.27			
INTERCONSTRUCTOR_PRE		0.07				
INTERCONSTRUCTOR_POST		0.23				
INTERCONSTRUCTOR_CONVERSION		0.03				
ADVICE_EXECUTE	0.29	0.002	0.76	0.005	0.14	1.29
ADVICE_ARG_SETUP	1.03	0.02	5.19	0.15	0.35	22.51
ADVICE_TEST				0.35	20.62	
THISJOINPOINT	2.05			0.03		
AROUND_CONVERSION	1.15	0.004				0.64
AROUND_CALLBACK		0.01				13.49
AROUND_PROCEED	0.30	0.01	2.53			5.79
CLOSURE_INIT		0.007				6.43
AFTER_RETURNING_EXPOSURE				0.002		
AFTER_THROWING_HANDLER				0.001		
CFLOW_ENTRY				46.00	35.39	
CFLOW_EXIT				50.83	39.29	
PERCFLOW_ENTRY				0.14		
PERCFLOW_EXIT				0.12		
CLINIT				0.001		
INLINE_ACCESS_METHOD			2.66			
ASPECTJ TAG MIX FOR ALLOCATIONS ONLY (WHOLE PROG.) (%)						
AspectJ Overhead (total)	**73.29**	**45.88**	**3.56**	**99.70**	**99.98**	**99.90**
BASE_CODE	0.39	0.57	3.74	0.03	0.02	0.10
ASPECT_CODE	26.32	53.54	92.69	0.27		
INTERFIELDINIT			3.56			
INTERCONSTRUCTOR_PRE		18.72				
INTERCONSTRUCTOR_POST		21.06				
INTERCONSTRUCTOR_CONVERSION						
ADVICE_ARG_SETUP		4.07		0.02		66.62
THISJOINPOINT	54.25			0.24		
AROUND_CONVERSION	19.04					
AROUND_PROCEED		2.03				33.28
CFLOW_ENTRY				98.97	99.98	
PERCFLOW_ENTRY				0.47		
PEROBJECT_ENTRY				0.009		
PEROBJECT_SET						
CLINIT	0.005	0.001		0.002		0.001
ASPECTJ METRICS FOR SHADOWS (WHOLE PROGRAM) (%)						
Hot Shadows (for 90%)	5.26	50.00	100.00	27.43	100.00	7.69
Hot Sources (for 90%)	25.00	14.29	100.00	66.67	100.00	100.00
Advice Execution Const.(%)				100.00	100.00	

Table 7.2: *Overall data: AspectJ metrics*

7.1.2 Benchmarks with low overhead

From the overall data presented in Tables 7.1 and 7.2, a benchmark can be determined to have low runtime overhead in one of two ways: the dynamic metrics may indicate low aspect overhead (in terms of bytecode executions and object allocations), or, for those benchmarks with equivalent hand-woven Java versions, low overhead may be indicated by a small difference in execution time.

The *dcm-sim* and *prodline* benchmarks are shown to have relatively low runtime overheads by the dynamic metrics alone (4.82% and 0.73% AspectJ execution overhead, respectively), while the *bean* benchmark is shown to have relatively low runtime overhead by the dynamic metrics (14.24% AspectJ execution overhead) and execution time comparisons (5% slowdown vs. handcoded with JIT enabled). Each of these benchmarks will be considered individually below.

DCM-sim

One potential use for aspects is to instrument a base program in order to report on a facet of its dynamic behaviour—to confirm, for example, that the base program conforms to some policy. Hassoun, Johnson, and Counsell have proposed a dynamic coupling metric [HJC04a], and provided an AspectJ implementation [HJC04b] that can be applied to any base program.

As is also true for many subsequent benchmarks, the aspects in the *dcm-sim* benchmark can be applied to any base program. *Certrevsim* [Arn00] is a reasonably large and complex Java application that simulates and compares various certificate revocation schemes. This is the base program with which the *dcm-sim* aspects have been woven.

This benchmark makes extensive use of advice, both to compute the dynamic coupling metric and to report it. To compute the coupling metric, **around** and **after** advice apply to all method and constructor executions in the application space. This advice, which updates data records in a hash table and makes use of reflective information by means of the **thisJoinPoint** construct, is relatively expensive.

The results for this benchmark can be seen in the *dcm-sim* columns of Tables 7.1

and 7.2. Since the advice bodies defined in the aspects are relatively complex, and the pointcuts fairly broad, it is unsurprising that the execution is dominated by ASPECT_CODE (which is non-overhead) instructions. The AspectJ overhead is less than 5%.

Looking at the application-only metrics, however, which ignore the execution in the Java library, and thus the expensive hash table methods called in the advice bodies, we see the aspect overhead to be 17%, much of which is THISJOINPOINT and AROUND_CONVERSION instructions. The whole-program allocation metrics indicate that 54% of allocations are attributed to THISJOINPOINT and 19% to **around** advice. 5% of total execution time is spent in garbage collection, so these allocations are not without cost, although it is relatively low compared to that for the high-overhead benchmarks presented later. Nevertheless, it suggests that **around** advice and use of **thisJoinPoint** may be a source of significant overhead in other benchmarks.

ProdLine

A *product line* is a family of related programs. Lopez-Herrejon and Batory use AspectJ's static crosscutting features to implement a product line of related graph algorithms [LHB02].

This benchmark's base program consists of a number of empty classes representing graph primitives (e.g. `Edge`, `Vertex`, `Graph`). It provides a mere skeleton of common types—the functionality of each program in the product line is defined in the aspects. These aspects make heavy use of intertype declarations to add members to these basic types, and also make use of advice.

One might expect the static crosscutting features of AspectJ to incur no runtime overhead, and Table 7.2 does indicate the execution overhead to be less than 1%. An examination of the allocation metrics, however, indicates that the overhead is not non-existant: 40% of allocations are due to introduced constructors. This does not adversely affect the execution time of this benchmark, but does show that even static crosscutting features can result in some runtime overhead, and are a potential area for improved code generation.

Bean

The *bean* benchmark is an example taken from the AspectJ Programming Guide [Tea01]. The base program is a simple data structure with little functionality (a `Point` class) and the aspects encapsulate the implementation of the JavaBeans protocol.

The aspect makes use of static crosscutting features to introduce new fields and methods into the `Point` class, and to declare the `Point` class an implementor of the `Serializable` interface. **around** advice is used to fire property change events when the `Point`'s coordinates change.

A hand-woven version of this benchmark—that is, a version of equivalent functionality written in pure Java, without aspects—was produced for comparison.

Although *bean* displays some overhead in interpreted mode (as indicated by the aspect overhead metric and slowdown vs. hand-coded) the JIT with inlining enabled succeeds in mostly eliminating this overhead. This supports the claim in [Xer03] that the JIT and inliner should eliminate most overhead. However, this benchmark does not make many overhead allocations, and weaving of the advice results in no dynamic residues. Furthermore, with the inliner disabled, there is a 15% performance penalty. Since inliner behaviour can be difficult to predict, it is not yet clear that it can always be relied upon to eliminate this overhead.

7.1.3 Benchmarks with high overhead

Although some benchmarks in the previous section showed some overhead, these overheads were relatively small and generally supported the belief that AspectJ results in little runtime cost that cannot be eliminated by the JIT and inliner. Contrary to this conventional belief, however, some benchmarks show very significant runtime overheads.

This section examines the benchmarks that exhibit high runtime overheads. Each benchmark is considered below, and the dynamic metrics and execution times are used to identify the runtime overheads. For each of these benchmarks, the overheads are primarily due to two potentially expensive AspectJ features: **around**

	Original	Counters	No cflow	Original+Soot	Counters+Soot
Program Size (Application Only)					
Classes Loaded	63	63	62	63	63
Instructions Loaded	27187	32480	15948	23162	29415
Code Coverage (%)	58	59	55	57	60
Program Size (Whole Program)					
Classes Loaded	385	391	390	391	391
Instructions Loaded	120933	127809	111277	118491	124744
Execution Time Measurements (Whole Program)					
# instr. (million bytecodes)	4814	1487	128	4750	1458
Total time - client (sec)	136.44	11.09	1.29	20.68	8.04
JIT time - client (sec)	0.64	0.38	0.23	0.68	0.28
GC time - client (sec)	91.88	0.88	0.06	1.17	0.22
Time - client_noinline (sec)	97.93	13.15	1.25	26.74	10.75
Time - interpreter (sec)	201.10	38.01	3.02	152.01	34.47
Execution Space Measurements (Whole Program)					
Mem. Alloc. (million bytes)	1004	40	39	1005	40
Obj. Alloc. Density (per kbc)	7.30	0.25	2.90	7.40	0.26
#Garbage Collections	1104	42	42	1103	42

Table 7.3: *Law of Demeter: general metrics*

advice and **cflow** pointcuts. Some lesser overhead is a result of using **thisJoinPoint** and aspect instance binding.

Law of Demeter

The Law of Demeter (*lod-sim*) benchmark is similar in purpose to *dcm-sim*: it uses aspects to instrument a base program and report on its dynamic behaviour—in this case, checking if the program has correct Law of Demeter object form.

	Original	Counters	No cflow
ASPECTJ METRICS SUMMARIZING OVERHEAD			
AspectJ Overhead % (whole)	**97.69**	**92.52**	**22.02**
#overhead/#advice (whole)	49.25	14.41	0.29
#advice/#total (whole)	0.02	0.06	0.76
AspectJ Runtime Lib % (whole)	94.01	72.42	20.93
ASPECTJ TAG MIX FOR ALL INSTRUCTIONS (WHOLE PROG.) (%)			
BASE_CODE	0.32	1.06	2.46
ASPECT_CODE	1.98	6.42	75.52
ADVICE_EXECUTE	0.005	0.02	0.18
ADVICE_ARG_SETUP	0.15	0.49	5.64
ADVICE_TEST	0.35	1.70	5.00
THISJOINPOINT	0.03	0.10	1.18
AFTER_RETURNING_EXPOSURE	0.002	0.005	0.06
AFTER_THROWING_HANDLER	0.001	0.004	0.05
CFLOW_ENTRY	46.00	79.70	
CFLOW_EXIT	50.83	9.23	
PERCFLOW_ENTRY	0.14	0.45	5.14
PERCFLOW_EXIT	0.12	0.39	4.47
CLINIT	0.001	0.002	0.02
ASPECTJ TAG MIX FOR ALLOCATIONS ONLY (WHOLE PROG.) (%)			
BASE_CODE	0.03	2.74	2.72
ASPECT_CODE	0.27	26.04	26.05
AspectJ Overhead (total)	99.70	71.23	71.23
ADVICE_ARG_SETUP	0.02	2.09	2.09
THISJOINPOINT	0.24	22.96	22.96
CFLOW_ENTRY	98.97	0.009	
PERCFLOW_ENTRY	0.47	45.04	45.05
PEROBJECT_ENTRY	0.009	0.90	0.90
PEROBJECT_SET			
CLINIT	0.002	0.23	0.22

Table 7.4: *Law of Demeter: AspectJ metrics*

A program is said to have correct Law of Demeter [LLW03a] object form when, for each of its objects, the object can only send messages to: itself, its arguments, its instance variables, a locally constructed object, or a returned object from a message sent to itself.

Lieberherr, Lorenz, and Wu have implemented an AspectJ program [LLW03b] that can be applied to any base program to determine if it has correct Law of Demeter object form. This implementation makes use of **cflow** pointcuts for its advice, and **percflow** and **pertarget** aspect instance binding.

These aspects have been applied to the same discrete event simulator as *dcm-sim*.

Like *dcm-sim*, the advice bodies in this benchmark do a fair amount of work. One might therefore assume that, like *dcm-sim*, the overhead would be overwhelmed by the advice bodies and would, consequently, be relatively low. This turns out, however, not to be the case—this benchmark exhibits an enormous amount of runtime overhead.

An examination of the tag mix metrics identifies the source of this overhead: 97.7% of instructions executed, and 99.7% of allocations to the heap, are due to **cflow** bookkeeping instructions (CFLOW_ENTRY and CFLOW_EXIT). These instructions are those generated by the weaver to manage the representations of the call stack required by the implementation of **cflow** pointcuts.

The allocation instructions here are not only, in themselves, expensive instructions unlikely to be optimized away by the JIT, but also contribute to increased garbage collection activity: for this benchmark, garbage collection accounted for 67% of execution time.

Cflow Stacks

A closer examination of `ajc`'s implementation of **cflow** pointcuts further explains this overhead, and suggests some potential optimizations to reduce it.

Consider the following pointcuts:

```
pointcut P1(T t): call(* foo(T)) && args(t);
pointcut P2(T t): cflow(P1(t)) && call(* bar());
```

The **cflow** pointcut, *P2*, serves two purposes: (1) it selects all join points that are executed within the dynamic scope of join points matching pointcut *P1*; (2) it exposes the (most recent) value of *t* to advice associated with this pointcut. An implementation of **cflow** pointcuts must support both of these features.

The implementation in `ajc` does so by associating a stack, each element of which is a set of variable bindings, to each pointcut. (In actual fact, there is one such stack per pointcut per thread of execution.) On entering a join point matching *P1*, a binding for *t* is pushed onto the stack corresponding to *P2*. On leaving the join point, the stack is popped. A join point can be determined to match the **cflow** clause of *P2* by testing that the stack is not empty. The bound context variables can be retrieved from the top of the stack and passed to the advice body when it is invoked.

The overhead involved in such an implementation is two-fold: that responsible for maintaining the stack (CFLOW_ENTRY and CFLOW_EXIT instructions) and that responsible for testing it (ADVICE_TEST, CFLOW_TEST, and PEROBJECT_ENTRY_TEST instructions.) The join point shadows at which instructions for maintaining the stack are woven are called *update shadows*.

The *lod-sim* benchmark, however, does not bind any context variables with its **cflow** pointcut. (This kind of parameterless **cflow** usage turns out to be quite typical.) Examinations of the generated bytecode and the compiler source reveal that in this case, an empty `Object` array is constructed and pushed onto the **cflow** stack—this is clearly a heavyweight solution.

In the case of parameterless **cflow**, using a simple counter instead of a stack should result in significantly lower overhead. To test this hypothesis, a modified version of `ajc`, `cajc`, which uses counters instead of stacks when possible, was written. Results are presented in the "Counters" column of Table 7.3. Clearly, this is a vast improvement—execution time in both JIT and interpreted modes is almost an order of magnitude lower, the number of allocations and total bytes allocated are an order of magnitude lower, and garbage collection time now accounts for a much more reasonable 2.7% of total execution time.

Duplicated stacks

Even with this optimization, however, the tag mix metric still indicates a large amount of overhead due to **cflow** bookkeeping. Further examination of the generated code reveals that a large number of **cflow** stacks have been created and are being maintained at update shadows, as suggested by the tag mix metrics. In fact, they are all being maintained at the same shadows: at least 13 **cflow** stacks are generated and maintained for the same **cflow** pointcut.

The *lod-sim* source defines a named pointcut `scope()`, which is then referenced in many other pointcut definitions. As a consequence of its presumed inlining into each of these other pointcuts, a multitude of (identical) state objects (counters or stacks) is being maintained at each update shadow.

To establish that this was the cause of the overhead, a version of the benchmark was written without the **cflow** clause in the `scope()` pointcut. The results can be seen in the "No cflow" column of Tables 7.3 and 7.4—a significant improvement again.

This is clearly a case for potential optimization. (The `abc` compiler performs such an optimization, unifying and reusing **cflow** state objects when possible. Results are presented in the next section.)

Locals

The dynamic metrics for this variant without the **cflow** clause in the `scope` pointcut still suggest the presence of some overhead: 5.6% of executions are attributed to ADVICE_ARG_SETUP. This tag represents the code inserted before advice executions. Examination of the generated bytecode reveals a very large number of local variables in use—possibly a result of `ajc`'s weaving at the bytecode level, with the required additional complexity of having to sort through the Java stack. In order to test this hypothesis, the benchmark was post-processed with SOOT, which performs a local packing optimization. Examination of the bytecode revealed a large reduction in the number of locals, and the results, shown in the "No cflow" column of Table 7.3, show a significant improvement. The last column of Table 7.3 shows the results of combining this SOOT optimization with the use of **cflow** counters.

The allocation metrics for the no-cflow variant of this benchmark still indicate the presence of other overheads. 23% of allocations made by the no-cflow variant of this benchmark are attributed to THISJOINPOINT and 45% are attributed to PERCFLOW_ENTRY. Thus, the uses of **thisJoinPoint** and aspect instance binding also incur some not insignificant overhead, and are potential areas for future optimization.

Finally, the advice execution metric indicates that every advice guard always evaluates the same. This suggests that it might be possible to completely eliminate the **cflow** overheads with a static analysis. (See section 7.2.2).

Figure

The Figure benchmark illustrates the use of an aspect to update the display in a figure editor[KHH+01a]. The `DisplayUpdating` aspect observes all of the shapes the editor supports with the following pointcut:

	Original	Counters
PROGRAM SIZE (APPLICATION ONLY)		
Classes Loaded	15	14
Instructions Loaded	594	654
Code Coverage (%)	64	61
PROGRAM SIZE (WHOLE PROGRAM)		
Classes Loaded	301	300
Instructions Loaded	75258	75318
EXECUTION TIME MEASUREMENTS (WHOLE PROGRAM)		
# instr. (million bytecodes)	2871	1431
Total time - client (sec)	20.17	6.79
JIT time - client (sec)	0.08	0.04
GC time - client (sec)	0.10	0.00
Slowdown vs. handcoded(×)	**37.35**	**12.57**
Time - client_noinline (sec)	30.12	13.20
Slowdown vs. handcoded (×)	25.18	11.04
Time - interpreter (sec)	136.86	38.37
Slowdown vs. handcoded (×)	27.63	7.75
EXECUTION SPACE MEASUREMENTS (WHOLE PROGRAM)		
Mem. Alloc. (million bytes)	374	1
Obj. Alloc. Density (per kbc)	5.58	0.01
#Garbage Collections	489	0

Table 7.5: *Figure: general metrics*

	Original	Counters
ASPECTJ METRICS SUMMARIZING OVERHEAD		
AspectJ Overhead % (whole)	**95.78**	**91.54**
#overhead/#advice (whole)	229.17	109.17
#advice/#total (whole)	0.00	0.01
AspectJ Runtime Lib % (whole)	91.67	79.38
ASPECTJ TAG MIX FOR ALL INSTRUCTIONS (WHOLE PROG.) (%)		
BASE_CODE	3.80	7.62
ASPECT_CODE	0.42	0.84
ADVICE_EXECUTE	0.14	0.28
ADVICE_ARG_SETUP	0.35	0.70
ADVICE_TEST	20.62	48.08
CFLOW_ENTRY	35.39	38.01
CFLOW_EXIT	39.29	4.47
ASPECTJ TAG MIX FOR ALLOCATIONS ONLY (WHOLE PROG.) (%)		
BASE_CODE	0.02	99.83
AspectJ Overhead (total)	99.98	0.17
CFLOW_ENTRY	99.98	0.09
CLINIT		0.09
ASPECTJ METRICS FOR SHADOWS (WHOLE PROGRAM) (%)		
Advice Execution Const.(%)	100.00	100.00

Table 7.6: *Figure: AspectJ metrics*

```
pointcut move():
      call(void FigureElement.moveBy(int, int))
      || call(void Point.setX(int))
      || call(void Point.setY(int))
      || call(void Line.setP1(Point))
      || call(void Line.setP2(Point));
```

Whenever a shape is moved, the display should be updated. However, we wish to avoid unnecessary updates. Translating a line, for example, involves translating the points that comprise the line, but should result in only a single display update. Consequently, the advice to update the display is defined as follows:

```
after() returning: move() && !cflowbelow(move()) {
   Display.needsRepaint();
}
```

The negated **cflowbelow** pointcut here checks that a *move* operation is not part of a more complex *move* operation—that is, that no *move* exists above it on the call stack. This eliminates unnecessary display updates.

The core program in this benchmark performs no interesting computation. This benchmark's purpose is to isolate and examine the cost of using the **cflowbelow** pointcut. To this end, it is compared with a hand-woven version, in which all of the calls to `Display.needsRepaint()` are added by hand. Table 7.1 shows that the AspectJ version of the benchmark is 37 times slower. The execution space measurements indicate a large number of allocations, which, as can be seen in the tag mix metric, are due to the updating of **cflow** state objects. This benchmark, like *lod-sim*, has a significant amount of overhead due to the use of **cflow** pointcuts. The hand-woven version, however, suggests that there is room for a great deal of improvement.

The figure benchmark was compiled with `cajc`, as *lod-sim* was. The results are presented in the "Counters" column of Tables 7.5 and 7.6. As with *lod-sim*, the use of counters in place of stacks results in a significant reduction in overhead

and a significant increase in performance. Client mode execution time drops from 20.17s to 6.79s. The number of CFLOW_ENTRY instructions sees a small reduction, but the allocation overhead, almost entirely attributed to CFLOW_ENTRY, drops from 99.98% to 0.17%. Also like *lod-sim*, the advice execution metric suggests that static analysis might be able to completely eliminate the remaining **cflow** overhead.

NullCheck

Asberry [Asb02] has suggested an aspect (*nullcheck-sim*) to test for the anti-pattern "on error condition, return `null` from method." The idea behind this anti-pattern is that it is generally preferable, in Java, to throw a meaningful exception than to return `null`. Like *dcm-sim* and *lod-sim*, *nullcheck-sim* reports on the behaviour of any base program without altering its behaviour, and like both of these other benchmarks, the *nullcheck-sim* aspects have been applied to the *Certrevsim* discrete event simulator.

The *nullcheck-sim* aspects use **around** advice, applied to all methods that return objects, to check if the return value is `null`. If it is, a message is printed to the console that includes the signature and static location of the offending method.

For comparison, a hand-woven Java version was produced. As can be seen in Table 7.1, the AspectJ version is significantly slower than the Java version. The dynamic metrics provide some insight into this performance difference.

The metrics indicate the presence of significant overhead in several ways. First, the AspectJ version executes 1,938 million instructions to the Java version's 963 million. According to the AspectJ overhead metric, 50% of these (approximately the difference) are overhead instructions. Second, the AspectJ version loads 138 application classes to the Java version's 22. Third, and most significant, the AspectJ version makes 19.34 allocations per kbc (for a total of 1,535 MB) to the Java version's 0.04 (for a total of 2MB). This difference is reflected in the garbage collection behaviour: the AspectJ version performs 1,526 collections, accounting for 24% of execution time, while the Java version performs only 2 accounting for approximately 0% of execution time.

	Original	All non-void methods	notwithin	after advice	Hand-woven
PROGRAM SIZE (APPLICATION ONLY)					
Classes Loaded	138	252	48	48	22
Instructions Loaded	8577	14021	7727	3926	2421
Code Coverage (%)	41	50	36	47	57
PROGRAM SIZE (WHOLE PROGRAM)					
Classes Loaded	456	576	366	372	334
Instructions Loaded	101011	108038	100161	97943	93678
EXECUTION TIME MEASUREMENTS (WHOLE PROGRAM)					
# instr. (million bytecodes)	1938	5034	1313	1089	963
Total time - client (sec)	9.01	26.29	3.14	3.10	2.30
JIT time - client (sec)	0.11	0.30	0.09	0.08	0.07
GC time - client (sec)	2.17	7.68	0.02	0.01	0.01
Slowdown vs. handcoded(×)	**3.92**	**11.43**	**1.37**	**1.35**	**1.00**
Time - client_noinline (sec)	10.10	34.08	3.43	3.35	2.35
Slowdown vs. handcoded (×)	4.30	14.50	1.46	1.43	1.00
Time - interpreter (sec)	67.23	226.62	24.38	20.84	17.43
Slowdown vs. handcoded (×)	3.86	13.00	1.40	1.20	1.00
EXECUTION SPACE MEASUREMENTS (WHOLE PROGRAM)					
Mem. Alloc. (million bytes)	1535	5725	2	2	2
Obj. Alloc. Density (per kbc)	19.34	32.29	0.03	0.04	0.04
#Garbage Collections	1526	5818	3	2	2

Table 7.7: *Nullcheck: general metrics*

	Original	All non-void methods	notwithin	after advice
ASPECTJ METRICS SUMMARIZING OVERHEAD				
AspectJ Overhead % (whole)	**50.15**	**69.56**	**25.63**	**13.74**
#overhead/#advice (whole)	19.49	20.03	5.40	6.00
#advice/#total (whole)	0.03	0.03	0.05	0.02
AspectJ Runtime Lib % (whole)	3.88	21.39	0.00	0.00
ASPECTJ TAG MIX FOR ALL INSTRUCTIONS (WHOLE PROG.) (%)				
BASE_CODE	47.27	26.97	69.62	83.97
ASPECT_CODE	2.57	3.47	4.75	2.29
ADVICE_EXECUTE	1.29	1.74	1.90	3.44
ADVICE_ARG_SETUP	22.51	26.66	16.13	8.02
THISJOINPOINT				
AROUND_CONVERSION	0.64	8.31	0.95	
AROUND_CALLBACK	13.49	16.35		
AROUND_PROCEED	5.79	7.81	6.64	
CLOSURE_INIT	6.43	8.68		
AFTER_RETURNING_EXPOSURE				2.29
ASPECTJ TAG MIX FOR ALLOCATIONS ONLY (WHOLE PROG.) (%)				
BASE_CODE	0.10	19.25	99.24	99.75
AspectJ Overhead (total)	99.90	80.75	0.76	0.25
ADVICE_ARG_SETUP	66.62	53.85		
THISJOINPOINT				
AROUND_PROCEED	33.28	26.90		
CLINIT	0.001		0.76	0.25

Table 7.8: *Nullcheck: AspectJ metrics*

The tag mix metrics, for executions and allocations, point to the implementation of **around** advice as the source of the overhead. In particular, the allocation tag mix indicates that 99.9% of allocations are performed by ADVICE_ARG_SETUP and AROUND_PROCEED instructions, combined.

An examination of the instructions in the generated bytecode revealed the use of heavyweight closures to implement **around** advice for this benchmark. Examination of the `ajc` source-code revealed that there are two strategies used for implementing **around** advice: this heavyweight closure strategy and an "inlining" strategy. The compiler prefers the inlining strategy, but falls back on the use of closures if the **around** advice body has **around** advice that applies to it. Since the **around** advice in *nullcheck-sim* makes several method calls returning objects, the use of closures for all applications of this advice is triggered.

Since it is presumably not of much value for the *nullcheck-sim* aspect to test itself, and in order to compare the closure strategy to the inlining strategy, an alternate version of the benchmark was written with the following pointcut:

```
call(Object+ *.*(..)) && !notwithin(codingstandards.*)
```

The **notwithin** clause deselects all join points within the advice body, thus allowing for the inlining strategy to be used. Results for this variant are presented in Tables 7.7 and 7.8.

(The use of closures can be forced in `ajc` with the `-Xnoinline` command line option. In order to confirm that the performance differences reported in Table 7.7 were due to the use of the inlining strategy, and not to the slight change in behaviour incurred by the use of the `notwithin` clause, the `notwithin` variant was compiled with this option turned on: results were no different than those for the original version that triggered the use of closures.)

It is possible to rewrite the benchmark using **after returning** advice instead of **around** advice, achieving the same functionality. This was done, and results for the **after returning** version are given in the last column of Tables 7.7 and 7.8. The metrics indicate even lower overhead for this version, suggesting that even the

inlining implementation of **around** advice weaving could be improved.

It is clear from these results that `ajc`'s inlining strategy for **around** advice is much more efficient than its closure strategy. However, the very large change observed in performance behaviour was due to a very small change in pointcut definition. Ideally, a programmer should not have to be concerned with the performance impacts of such minor changes.

This last point is emphasised by the "All non-void" variant of the benchmark. This is the original implementation of the aspect, which although functioned correctly, contained a coding error that drastically affected performance: where the documentation claimed the aspect checked the return value of methods that could return objects, the implementation checked the return values of all non-void methods. The error has been fixed for the primary version of the benchmark, but results for this version are shown to illustrate the performance consequence a small change to pointcut definition can have.

7.2 abc Results

Where `ajc` has been designed with fast incremental compilation and integration with the Eclipse toolset as primary design goals, `abc` has been designed with extensibility and optimization as primary goals. `ajc`'s emphasis on fast incremental compilation comes at the cost of optimization: little intraprocedural and no whole-program analysis is done. The design of `abc`, in contrast, facilitates and incorporates both intraprocedural and interprocedural analyses and optimizations, at the cost of incremental compilation.

The observation of significant runtime overheads in AspectJ programs prompted, in part, the development of `abc`. In order to facilitate the design of novel optimizations, `abc`'s back-end is based on the SOOT analysis and optimization framework, which not only makes it easy to develop new analyses and optimizations for AspectJ programs, but provides a library of stock analyses and optimizations that can be leveraged immediately in `abc`. (The local packing optimization mentioned in

the previous section, for example.)

One major difference between `ajc` and `abc` is that where weaving in `ajc` is performed on Java bytecode, it is performed on Jimple in `abc`. Jimple is a typed 3-address intermediate representation used by SOOT. The use of Jimple simplifies the weaving process and facilitates analysis.

The AspectJ-specific optimizations present in `abc` address each of the major overheads identified in section 7.1.3. They are described in detail in [ACH+05b]. In brief, they are:

1. An improved implementation of **around** weaving, reducing the use of expensive closures.

2. Intraprocedural optimizations to reduce the cost associated with **cflow** pointcuts.

3. An interprocedural optimization to eliminate the cost of **cflow** pointcuts in many cases.

And again, because `abc` is based on SOOT, it makes use of the standard SOOT optimizations.

The following sections will briefly explain the optimizations, and present results for the relevant benchmarks compiled using them.

7.2.1 Intraprocedural cflow optimizations

As explained in section 7.1.3, `ajc`'s implementation results in significant overheads of several kinds. `abc` incorporates several intraprocedural optimizations to reduce these overheads.

1. It is often the case that **cflow** pointcuts expose no context. This turns out to be true for all of the benchmarks used in this work. In this case, as demonstrated by the `cajc` modifications to `ajc` introduced in section 7.1.3, using counters instead of stacks is preferable—`abc` does so when appropriate.

2. `abc` attempts to unify **cflow** pointcuts. Often, pointcuts which are similar enough to share states don't, resulting in redundant updating. By unifying pointcuts, `abc` can reduce the cost of updating pointcut states and also reduce code bloat.

3. caching of **cflow** state objects to locals.

The results in Tables 7.9 and 7.10 are for the benchmarks that showed **cflow**-related overheads, compiled with `abc`, with intraprocedural optimizations enabled. For the sake of comparison, results for the same benchmarks, as compiled with `ajc` and `cajc`, and presented in section 7.1, are also listed. These results show significant improvements. *lod-sim*'s execution time, in client mode, is 1.1% what it is when compiled with `ajc` (1.46s vs. 136.44s), and *figure*'s is 6% (1.19s vs. 20.17s). The metrics show the expected and corresponding reduction in overhead: CFLOW_ENTRY and CFLOW_EXIT instructions account for much less of the execution, and, in interpreted mode, the number of executed bytecodes differs by an order of magnitude (147 million to 4,814 million for *lod-sim*, and 310 million to 2,871 million for *figure*.)

7.2.2 Interprocedural cflow optimizations

The results in Tables 7.9 and 7.10 still indicate the presence of noticeable **cflow**-related overhead, though significantly reduced by `abc`'s intraprocedural optimizations. The slowdown vs. handcoded metric for *figure*, for instance, suggests additional room for improvement.

The advice execution metric indicates, for a given run of a benchmark, what percentage of advice guards always evaluate to true, always evaluate to false, and sometimes evaluate to true and sometimes to false. A guard that sometimes evaluates to true and sometimes to false clearly cannot be statically eliminated. One that always evaluates the same, for a given run, may always evaluate the same for every run, and thus suggests the possibility that a more sophisticated static analysis could

	lod-sim (abc)	*lod-sim* (cajc)	*lod-sim* (ajc)	*figure* (abc)	*figure* (cajc)	*figure* (ajc)
PROGRAM SIZE (APPLICATION ONLY)						
Classes Loaded	59	63	63	13	14	15
Instructions Loaded	22441	32480	27187	583	654	594
Code Coverage (%)	54	59	58	72	61	64
PROGRAM SIZE (WHOLE PROGRAM)						
Classes Loaded	389	391	385	301	300	301
Instructions Loaded	117784	127809	120933	75261	75318	75258
EXECUTION TIME MEASUREMENTS (WHOLE PROGRAM)						
# instr. (million bytecodes)	147	1487	4814	310	1431	2871
Total time - client (sec)	1.46	11.09	136.44	1.19	6.79	20.17
JIT time - client (sec)	0.19	0.38	0.64	0.03	0.04	0.08
GC time - client (sec)	0.06	0.88	91.88	0.00	0.00	0.10
Slowdown vs. handcoded(×)				**3.72**	**12.57**	**37.35**
Time - client_noinline (sec)	1.39	13.15	97.93	1.57	13.20	30.12
Slowdown vs. handcoded (×)				2.18	11.04	25.18
Time - interpreter (sec)	3.51	38.01	201.10	7.00	38.37	136.86
Slowdown vs. handcoded (×)				2.58	7.75	27.63
EXECUTION SPACE MEASUREMENTS (WHOLE PROGRAM)						
Mem. Alloc. (million bytes)	44	40	1004	1	1	374
Obj. Alloc. Density (per kbc)	3.09	0.25	7.30	0.02	0.01	5.58
#Garbage Collections	47	42	1104	0	0	489

Table 7.9: *abc with intraprocedural optimization: general metrics*

	lod-sim (abc)	*lod-sim* (cajc)	*lod-sim* (ajc)	*figure* (abc)	*figure* (cajc)	*figure* (ajc)
ASPECTJ METRICS SUMMARIZING OVERHEAD						
AspectJ Overhead % (whole)	**30.29**	**92.52**	**97.69**	**65.80**	**91.54**	**95.78**
#overhead/#advice (whole)	0.45	14.41	49.25	17.00	109.17	229.17
#advice/#total (whole)	0.67	0.06	0.02	0.04	0.01	0.00
ASPECTJ TAG MIX FOR ALL INSTRUCTIONS (WHOLE PROG.) (%)						
BASE_CODE	2.21	1.06	0.32	34.20	7.62	3.80
ASPECT_CODE	67.50	6.42	1.98		0.84	0.42
INLINED_ADVICE			3.87			
ADVICE_EXECUTE	0.22	0.02	0.005		0.28	0.14
ADVICE_ARG_SETUP	2.83	0.49	0.15	5.16	0.70	0.35
ADVICE_TEST	2.07	1.70	0.35	7.74	48.08	20.62
THISJOINPOINT	1.06	0.10	0.03			
CFLOW_ENTRY	9.49	79.70	46.00	27.42	38.01	35.39
CFLOW_EXIT	7.92	9.23	50.83	20.64	4.47	39.29
PERCFLOW_ENTRY	3.11	0.45	0.14			
PERCFLOW_EXIT	2.50	0.39	0.12			
GET_CFLOW_LOCAL	0.31			0.97		
ASPECTJ TAG MIX FOR ALLOCATIONS ONLY (WHOLE PROG.) (%)						
AspectJ Overhead (total)	**60.26**	**71.23**	**99.70**	**0.09**	**0.17**	**99.98**
BASE_CODE	2.17	2.74	0.03	99.91	99.83	0.02
ASPECT_CODE	37.57	26.04	0.27			
THISJOINPOINT	20.10	22.96	0.24			
CFLOW_ENTRY			98.97		0.09	99.98
PERCFLOW_ENTRY	37.70	45.04	0.47			
PEROBJECT_ENTRY	0.74	0.90	0.009			
CLINIT	0.002	0.23	0.002	0.09	0.09	
ASPECTJ METRICS FOR SHADOWS (WHOLE PROGRAM) (%)						
Hot Shadows (for 90%)	27.62	27.62	27.62	100.00	100.00	100.00
Hot Sources (for 90%)	66.67	66.67	66.67	100.00	100.00	100.00
Advice Execution Const.(%)	100.00	100.00	100.00	100.00	100.00	100.00

Table 7.10: *abc with intraprocedural optimization: AspectJ metrics*

eliminate the guard entirely. The value of the metric for these benchmarks (100%) suggests that guard elimination may be possible.

`abc` has an interprocedural analysis for eliminating these guards. Since performing this sort of analysis before weaving can be very complicated, (for one thing, the analysis must be AspectJ-aware,) `abc` performs the analysis after weaving and feeds the results back into the weaver for reweaving. Weaving, therefore, occurs as a two stage process in `abc`: the first stage is a naive weaving, the results of which are subject to an interprocedural analysis. This analysis informs the second stage, which repeats the weaving operation with this additional information. This is illustrated in Figure 4.2 and explained in more detail in [ACH+05b]. The result of this optimization is the elimination of unnecessary **cflow** advice guards and of unnecessary updates to **cflow** state objects.

Table 7.11 shows the results of compiling the benchmarks with this optimization enabled. Using static analysis to eliminate **cflow** checks and updating is clearly desirable. The execution time of *figure* compiled with `abc` with interprocedural **cflow** elimination is 1% that of the same benchmark compiled with `ajc` 1.2, (that is, it is 100× faster,) and it is only 1% slower than the hand-woven version. The execution time of *lod-sim* is likewise 1% what it is when compiled with `ajc` 1.2. Both are improvements over the performance gained with intraprocedural **cflow** optimization alone.

7.2.3 around optimizations

around advice is the second AspectJ feature that results in high overhead in benchmarks compiled with `ajc`. Consequently, `abc` optimizes the weaving of around advice.

As shown in section 7.1.3, it is the creation of heavyweight closures that causes the significant overheads observed with **around** advice.

`abc`'s implementation of **around** weaving, which [ACH+05b] describes in detail, never performs significantly worse than `ajc`'s, and often performs significantly better. When `ajc` uses closures, `abc` produces much faster code. When advice is

	lod-sim (inter)	*lod-sim* (intra)	*figure* (inter)	*figure* (intra)
PROGRAM SIZE (APPLICATION ONLY)				
Classes Loaded	54	59	8	13
Instructions Loaded	16092	22441	245	583
Code Coverage (%)	57	54	75	72
PROGRAM SIZE (WHOLE PROGRAM)				
Classes Loaded	382	389	294	301
Instructions Loaded	111421	117784	74909	75261
EXECUTION TIME MEASUREMENTS (WHOLE PROGRAM)				
# instr. (million bytecodes)	114	147	121	310
Total time - client (sec)	1.24	1.46	0.35	1.19
JIT time - client (sec)	0.15	0.19	0.03	0.03
GC time - client (sec)	0.06	0.06	0.00	0.00
Slowdown vs. handcoded(×)			**1.03**	**3.72**
Time - client_noinline (sec)	1.23	1.39	0.80	1.57
Slowdown vs. handcoded (×)			1.07	2.18
Time - interpreter (sec)	2.87	3.51	2.98	7.00
Slowdown vs. handcoded (×)			1.08	2.58
EXECUTION SPACE MEASUREMENTS (WHOLE PROGRAM)				
Mem. Alloc. (million bytes)	44	44	1	1
Obj. Alloc. Density (per kbc)	3.96	3.09	0.06	0.02
#Garbage Collections	47	47	0	0

Table 7.11: *abc with interprocedural optimization*

large and applies frequently, `abc` produces code with much less bloat. In the case of circular advice application, `abc` produces fewer closures and is much faster.

Table 7.13 shows the results of compiling **around**-heavy benchmarks with `abc`. The enormous performance difference observed between *nullcheck-sim* and *nullcheck-norec*, when compiled with `ajc`, has been eliminated. The small change in pointcut definition no longer incurs a huge change in performance. Likewise, the "All non-void" variant is no longer as costly a coding error—it is now only 3.76 times slower than the hand-coded version. (Although it is still 3 times slower than the corrected version.)

	Original	All non-void methods	notwithin	after advice	Hand-woven
PROGRAM SIZE (APPLICATION ONLY)					
Classes Loaded	43	43	43	43	22
Instructions Loaded	6333	10306	6010	3828	2539
Code Coverage (%)	50	56	51	49	55
PROGRAM SIZE (WHOLE PROGRAM)					
Classes Loaded	367	367	367	367	345
Instructions Loaded	100350	104323	100027	97845	96549
EXECUTION TIME MEASUREMENTS (WHOLE PROGRAM)					
# instr. (million bytecodes)	1426	3269	1426	1089	901
Total time - client (sec)	2.99	8.80	2.95	2.85	2.33
JIT time - client (sec)	0.12	0.20	0.09	0.08	0.06
GC time - client (sec)	0.01	1.31	0.01	0.01	0.01
Slowdown vs. handcoded(×)	**1.28**	**3.78**	**1.27**	**1.22**	**1.00**
Time - client_noinline (sec)	3.24	13.05	3.31	3.22	2.40
Slowdown vs. handcoded (×)	1.35	5.44	1.38	1.34	1.00
Time - interpreter (sec)	26.52	67.53	27.82	21.79	16.00
Slowdown vs. handcoded (×)	1.66	4.22	1.74	1.36	1.00
EXECUTION SPACE MEASUREMENTS (WHOLE PROGRAM)					
Mem. Alloc. (million bytes)	2	973	2	2	2
Obj. Alloc. Density (per kbc)	0.03	9.57	0.03	0.04	0.04
#Garbage Collections	2	956	2	2	2

Table 7.12: *abc with around optimization (NullCheck): general metrics*

	Original	All non-void methods	notwithin	after advice
ASPECTJ METRICS SUMMARIZING OVERHEAD				
AspectJ Overhead % (whole)	**28.00**	**43.54**	**28.00**	**13.76**
#overhead/#advice (whole)	4.57	4.65	4.57	4.00
#advice/#total (whole)	0.06	0.09	0.06	0.03
AspectJ Runtime Lib % (whole)	0.00	0.00	0.00	0.00
ASPECTJ TAG MIX FOR ALL INSTRUCTIONS (WHOLE PROG.) (%)				
BASE_CODE	65.87	47.09	65.87	82.80
ASPECT_CODE	6.13	9.36	6.13	3.44
INLINED_ADVICE				
ADVICE_EXECUTE	7.88	14.71	7.88	3.44
ADVICE_ARG_SETUP	7.88	12.04	7.88	10.32
THISJOINPOINT	0.001	0.001	0.001	
AROUND_CONVERSION				
AROUND_CALLBACK				
AROUND_PROCEED	12.23	16.79	12.23	
CLOSURE_INIT				
AFTER_RETURNING_EXPOSURE				
ASPECTJ TAG MIX FOR ALLOCATIONS ONLY (WHOLE PROG.) (%)				
AspectJ Overhead (total)	**1.06**	**0.002**	**0.95**	**0.26**
BASE_CODE	98.94	100.00	99.05	99.74
ADVICE_ARG_SETUP				
THISJOINPOINT	1.06	0.002	0.94	0.26
AROUND_PROCEED				
CLINIT	0.003		0.003	0.003

Table 7.13: *abc with around optimization (NullCheck): AspectJ metrics*

7.3 Results for the latest ajc and abc

A number of these optimizations have recently been incorporated into `ajc`. This section provides execution time comparisons of high-overhead benchmarks in `ajc` 1.2.1 and the most recent development version, as of August 2005, of `abc`. The results are presented in Table 7.14. (Since the tagging code has not yet been ported to these versions, only general metrics and execution times are presented.) As can be seen, `ajc` 1.2.1 has a significantly improved **cflow** implementation, but still lags `abc`, especially when `abc`'s interprocedural optimizations are enabled. Client mode execution time for *figure* is 0.35s when compiled with the latest `abc` and 7.40s when compiled with `ajc` 1.2.1; since *figure*'s overhead is almost entirely due to **cflow**, `abc`'s interprocedural **cflow** optimization results in a large performance difference. *lod-sim*, however, has additional overhead due to aspect instance binding and use of THISJOINPOINT. Consequently, it sees a similar but lesser performance difference: 1.41s for `abc` compared to 2.26s for `ajc`. **around** advice, however, can still be extremely expensive in `ajc` 1.2.1 when the closure strategy is used, while `abc` continues to improve its implementation; the performance difference between `ajc` and `abc` is a factor of three.

(*lod-sim* compiled with `abc` allocates slightly more memory than *lod-sim* compiled with `ajc`—44MB compared to 40MB, 3.96 allocations per kbc to 1.20—this is due not to compilation strategies but to the different runtime library implementations.)

7.4 Summary

While some of the benchmarks analyzed here seem to confirm the general belief that AspectJ incurs little runtime overhead, others have shown runtime overheads increasing execution time by an order of magnitude. The *lod-sim* benchmark shows very high runtime overheads due to its use of **cflow** pointcuts and the *nullcheck-sim*

	lod-sim (ajc)	*lod-sim* (abc)	*figure* (ajc)	*figure* (abc)	*nullcheck-sim* (ajc)	*nullcheck-sim* (abc)
PROGRAM SIZE (APPLICATION ONLY)						
Classes Loaded	66	54	15	8	138	43
Instructions Loaded	20363	16092	484	245	8695	6393
Code Coverage (%)	54	57	74	75	41	50
PROGRAM SIZE (WHOLE PROGRAM)						
Classes Loaded	395	382	302	294	462	361
Instructions Loaded	115699	111421	75155	74909	102712	98827
EXECUTION TIME MEASUREMENTS (WHOLE PROGRAM)						
# instr. (million bytecodes)	346	114	1805	121	1939	1650
Total time - client (sec)	2.26	1.41	7.40	0.35	8.83	2.81
JIT time - client (sec)	0.26	0.15	0.04	0.03	0.12	0.13
GC time - client (sec)	0.06	0.06	0.00	0.00	2.04	0.01
Slowdown vs. handcoded(×)			**21.76**	**1.03**	**3.74**	**1.20**
Time - client_noinline (sec)	2.75	1.33	11.32	0.80	11.44	3.31
Slowdown vs. handcoded (×)			14.89	1.07	4.70	1.38
Time - interpreter (sec)	9.02	2.96	46.66	2.98	75.81	22.75
Slowdown vs. handcoded (×)			15.88	1.08	4.24	1.42
EXECUTION SPACE MEASUREMENTS (WHOLE PROGRAM)						
Mem. Alloc. (million bytes)	41	44	1	1	1523	2
Obj. Alloc. Density (per kbc)	1.20	3.96	0.00	0.06	19.33	0.02
#Garbage Collections	43	47	0	0	1525	2

Table 7.14: *General metrics for the latest compilers*

benchmark shows very high runtime overheads due to its use of **around** advice. Additional lesser overheads, associated with **thisJoinPoint** and aspect instance binding, are also found. In the case of *nullcheck-sim*, careful programming can avoid most of these overheads. For example, writing "tighter" pointcuts or using **after returning** advice instead of **around** advice can have a significant improvement on performance. As the `abc` compiler demonstrates, however, there is significant room for optimization in the implementation of these features, and the programmer should not necessarily have to "hand-optimize" their aspects in this fashion.

This chapter has presented some key benchmarks illustrating the main overheads found in AspectJ programs. Execution time comparisons between `abc` and `ajc` for some additional benchmarks affected by the same overheads identified here can be found in [ACH+05b].

Chapter 8
Related Work

Work on analyzing the performance of AspectJ programs and the efficiency of AspectJ compilers is limited. Some research on this, and on closely related subjects, however, is briefly surveyed below.

A fair amount of research has been done on the subject of dynamic metrics, much of which, however, has been related to software engineering complexity and quality measures.

Yacoub *et al*. [YAR99] present a suite of dynamic metrics for assessing the design quality of object-oriented systems, including metrics for dynamic complexity and object cohesion. The metrics are applied to a sample application, and the dynamic measurements made with these metrics are compared to measurements made with corresponding static metrics.

Dufour *et al*. [DDHV03] have defined a set of general dynamic metrics for characterizing Java programs and describe a framework for collecting them. This work is the basis for the dynamic metrics collection in this thesis. It is further described in Bruno Dufour's masters thesis [Duf04].

Previous work on analyzing the behaviour of Java programs at the bytecode level has been done by Daly *et al*. In [DHPW01], they claim that

> Even though the majority of Java code executed may now be using some form of JIT compiler, dynamic analysis of interpreted bytecode usage

> ...can provide valuable information for profiling programs and for the design and implementation of virtual machines.

The authors use this form of analysis to do a comparative study of Java Grande Forum [EPC] benchmarks across a variety of platforms. They establish, by means of this analysis, that compiler choice is not the main explanation of observed execution speed differences.

Brown *et al*. similarly use bytecode-level dynamic analysis in [BAMP05] to compare certain static and dynamic metrics using coverage criteria.

The `abc` group has performed a lot of recent work optimizing the implementation of AspectJ to reduce the impact of runtime overhead. The general design of the compiler is described in [ACH+04] while a closer presentation of optimizations to reduce **cflow** and **around** advice overheads is described in [ACH+05b]. Sascha Kuzins' masters thesis [Kuz04] is a detailed description of `abc`'s strategy for weaving **around** advice. Sereni and de Moor present a theoretical alternate implementation of pointcut designators and an analysis for the static elimination of runtime matching in [SdM03].

Hilsdale and Hugunin describe the implementation of advice weaving in `ajc` in [HH04]. They conclude with a limited performance study, comparing AspectJ implementations of a logging aspect using **before** advice, applied to the Xalan XML parser, with a hand-woven equivalent. They benchmark the Xalan library with the XSLTMark benchmark. In order to isolate the overhead associated with executing the advice, logging functionality is disabled. A naive implementation of the aspect is found to incur 2900% overhead. They find 22% overhead in an optimized version, however, which they claim to be "an upper bound on the performance overhead for well-written advice." The work presented in this thesis is a more comprehensive performance study, and identifies some very significant overheads due to other AspectJ features, not examined by the authors.

Pace and Campo [PC01] compare several different approaches to aspect oriented programming (including AspectJ). A temperature control benchmark is implemented in these various approaches, and several quality factors are studied, one

of which is performance. The AspectJ version is shown to be insignificantly slower than the standard (Java) version.

The performance of aspect weaving is of particular concern when it is performed dynamically at runtime. As such, a number of studies have addressed this issue, including that by Sato *et al*. [SCT03] and Popovic *et al*. [PAG03]. Performance of the aspect weaver itself, however, in either a static or dynamic weaving context, is orthogonal to the work in this thesis.

Chapter 9
Conclusions and Future Work

The features AspectJ provides for the modularization of crosscutting concerns show much promise for improving source code quality in complex systems. Until now, however, little work has been done to establish the runtime cost of these features, and it has been taken mostly as an article of faith that this cost is negligible.

This thesis has provided a means for evaluating the code generation strategies of AspectJ compilers. This has included:

- Defining some new AspectJ-specific dynamic metrics and implementing these metrics in the **J* dynamic analysis framework.
- Defining a taxonomy of AspectJ overheads.
- Modifying the existing AspectJ compilers, `ajc` and `abc`, to annotate the class-files they generate with metadata required for the computation of the AspectJ-specific dynamic metrics.
- Collecting a set of AspectJ benchmarks.

By these means, it has identified some significant runtime overheads in programs compiled with `ajc` 1.2, and attributed them to particular uses of **cflow** pointcuts and **around** advice. It has suggested some possible improvements, which have been implemented, among others, in `abc`, to which `ajc` is here compared. Some of these optimizations have since been integrated in `ajc` 1.2.1.

9.1 Future Work

The work presented in this thesis can be extended upon in a number of ways. Some possible directions for future work are presented in this section.

9.1.1 Accurate measurement of overhead time

In order for the **J* agent to produce a useful execution trace for a program, the JVM must execute the program in interpreted mode. The AspectJ dynamic metrics, therefore, describe the program's behaviour when run in interpreted mode. It is quite possible for runtime overhead in interpreted mode to disappear when the JIT is enabled. At first, this may seem like a severe limitation of these metrics. In practice, however, this does not invalidate their use, as has been shown in this work. In many cases, the JIT cannot reduce runtime overhead to insignificant levels, and the dynamic metrics remain useful for identifying and locating these overheads. Overheads identified by execution time comparisons performed with the JIT enabled, for example, can often be explained by metrics calculated with the JIT disabled. Furthermore, the allocation tag mix metric, for instance, counts expensive instructions that remain costly even when the JIT is enabled, and thus tends to identify overheads that are not eliminated by the JIT.

Furthermore, not all bytecode instructions are of equal cost. A large number of executed overhead instructions does not necessarily indicate a large runtime overhead, even in interpreted mode. Metrics that count an expensive subset of instructions (such as the allocation metrics) are a partial solution to this, but obtaining an accurate measure of actual overhead execution time would be preferable to counting instructions in interpreted mode.

One way to achieve this might be to implement the metric calculations in the JVM. For example, the classloader could demarcate regions of overhead code with special accounting bytecode instructions. Another could be to postprocess the tagged classfiles and insert calls to high-precision native timing routines. However it is

achieved, obtaining accurate and precise measures of overhead execution time, especially with the JIT enabled, would probably be a valuable extension.

9.1.2 Measurement efficiency

Although dynamic metrics are a useful tool, the current technology for measuring them has some performance limitations. The **J* agent can produce many gigabytes of trace data, which must be output and then read by the **J* analyzer. I/O is a major bottleneck. Metric calculation for even simple benchmarks can take many hours. This limits the size of the programs that can be analyzed and also limits the audience: some metrics would provide useful information to regular programmers, but not when it takes a day to compute them. This is a limitation of **J* more than it is of the metrics themselves, and implementing the metrics directly in the JVM has the potential to eliminate this I/O bottleneck and significantly improve performance.

9.1.3 Additional metrics

Implementing the metric calculations in the JVM opens some additional avenues for extension. The current metrics are limited by the information made available to **J* by the JVMPI. By implementing the metrics in the JVM, a great deal of additional runtime information might be made available. For example, the current allocation metrics are a little crude: the number of bytes allocated, and the number of allocations made. A program that allocates and frees 100K 10,000 times for a total of 1GB over the course of its run has significantly different behaviour than one that allocates 900MB at once and 10K 10,000 times, but this difference is not captured by the current metrics. Having access to the garbage collector potentially allows for some much more sophisticated and interesting allocation metrics.

Other additional metrics, not requiring implementation within the JVM, could also be defined. Further subdividing instruction kinds, for example, may prove useful for the study of particular features and optimizations. Likewise may counting them for different subsets of the execution.

9.1.4 New optimizations

Work has been done in the `abc` compiler to optimize the implementations of the two most significant sources of overhead identified in this thesis: **around** advice and **cflow** pointcuts. Some similar improvements have been made to recent versions of `ajc`. Overheads due to other features have been found, and although they seem generally to be of lower impact, may prove to be an avenue of fruitful further research. In particular, the implementations of **thisJoinPoint** and of advice instance binding incur overheads that might be reduced by static analysis and improved code generation.

Bibliography

[ACH+04] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Building the *abc* AspectJ compiler with Polyglot and Soot. Technical Report abc-2004-4, The abc Group, 2004. Available from: http://aspectbench.org/techreports#abc-2004-4.

[ACH+05a] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. *abc*: An extensible AspectJ compiler. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 87–98. ACM Press, 2005. doi:10.1145/1052898.1052906.

[ACH+05b] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Optimising AspectJ. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 117–128. ACM Press, 2005. doi:10.1145/1065010.1065026.

[aG] The abc Group. The AspectBench compiler for AspectJ. Home page with downloads, FAQ, documentation, support mailing lists, and bug database. Available from: http://aspectbench.org/.

[Arn00] André Arnes. Certificate revocations performance simulation project, 2000. Available from: http://www.pvv.ntnu.no/~andrearn/certrev/sim.html.

[Asb02] R. Dale Asberry. Aspect oriented programming (AOP): Using AspectJ to implement and enforce coding standards, 2002. Available from: http://www.daleasberry.com/newsletters/200210/20021002.shtml.

[Asp] AspectJ Eclipse Home. The AspectJ home page. Available from: http://eclipse.org/aspectj/.

[BAMP05] Stephen Brown, Áine Mitchell, and James Power. A coverage analysis of Java benchmark suites. In *The IASTED International Conference on Software Engineering*, pages 144–150. ACTA Press, 2005. Available from: http://www.cs.may.ie/~jpower/Research/Papers/2005/.

[DDHV03] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 149–168. ACM Press, 2003. doi:10.1145/949305.949320.

[DGH+04] Bruno Dufour, Christopher Goard, Laurie Hendren, Oege de Moor, Ganesh Sittampalam, and Clark Verbrugge. Measuring the dynamic behaviour of AspectJ programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 150–169. ACM Press, 2004. doi:10.1145/1028976.1028990.

[DHPW01] Charles Daly, Jane Horgan, James Power, and John Waldron. Platform independent dynamic Java virtual machine analysis: the Java Grande Forum benchmark suite. In *Proceedings of the joint ACM-ISCOPE conference on Java Grande (JGI)*, pages 106–115. ACM Press, 2001. doi:10.1145/376656.376826.

[Duf04] Bruno Dufour. Objective quantification of program behaviour using dynamic metrics. Master's thesis, McGill University, Montréal, Québec, Canada, 2004. Available from: `http://www.sable.mcgill.ca/publications/thesis/#brunoMastersThesis`.

[EPC] EPCC. The Java Grande Forum benchmark suite. Available from: `http://www.epcc.ed.ac.uk/javagrande/javag.html`.

[Fou] Apache Software Foundation. The byte code engineering library. Available from: `http://jakarta.apache.org/bcel/`.

[HG05] Bruno Harbulot and John Gurd. A join point for loops in AspectJ. In *Proceedings of the Foundations of Aspect-Oriented Languages Workshop at AOSD (FOAL)*, pages 11–20, 2005. Available from: `http://www.cs.iastate.edu/~leavens/FOAL/`.

[HH04] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 26–35. ACM Press, 2004. doi:10.1145/976270.976276.

[HJC04a] Youssef Hassoun, Roger Johnson, and Steve Counsell. A dynamic runtime coupling metric for meta-level architectures. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 339–346. IEEE Computer Society Press, 2004. doi:10.1109/CSMR.2004.1281436.

[HJC04b] Youssef Hassoun, Roger Johnson, and Steve Counsell. Empirical validation of a dynamic coupling metric. Technical Report BBKCS-04-03,

Birbeck College, London, 2004. Available from: http://www.dcs.bbk.ac.uk/link/research/techreps/2004/.

[HK02] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 161–173. ACM Press, 2002. Available from: http://www.cs.ubc.ca/~jan/AODPs/.

[HOT02] William H. Harrison, Harold L. Ossher, and Peri L. Tarr. Asymmetrically vs. symmetrically organized paradigms for software composition. Research Report RC22685, IBM Thomas J. Watson Research Center, 2002.

[KHH+01a] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001. doi:10.1145/383845.383858.

[KHH+01b] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001. doi:10.1145/646158.680006.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[Kuz04] Sascha Kuzins. Efficient implementation of around-advice for the aspectbench compiler. Master's thesis, Oxford University, 2004. Available from: http://aspectbench.org/theses.

[Lad03] Ramnivas Laddad. *AspectJ in Action*. Manning, 2003.

[LHB02] Roberto E. Lopez-Herrejon and Don Batory. Using AspectJ to implement product-lines: A case study. Technical report, University of Texas at Austin, 2002. Available from: http://www.cs.utexas.edu/users/rlopez/Publications.html.

[LLW03a] Karl Lieberherr, David H. Lorenz, and Pengcheng Wu. A case for statically executable advice: checking the law of Demeter with AspectJ. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 40–49. ACM Press, 2003. doi:10.1145/643603.643608.

[LLW03b] Karl Lieberherr, David H. Lorenz, and Pengcheng Wu. A case for statically executable advice: Checking the law of Demeter with AspectJ. Code available from URL: http://www.ccs.neu.edu/home/lorenz/papers/aosd2003lod/, 2003.

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification (2nd Edition)*. Addison-Wesley Professional, 1999. Available from: http://java.sun.com/docs/books/vmspec/.

[MKD03] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Proceedings of the International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2003.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of the International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2003.

[PAG03] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-in-time aspects: efficient dynamic weaving for Java. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 100–109. ACM Press, 2003. doi:10.1145/643603.643614.

[PC01] J. Andrés Díaz Pace and Marcelo R. Campo. Analyzing the role of aspects in software design. *Communications of the ACM*, 44(10):66–73, 2001. doi:10.1145/383845.383859.

[SCT03] Yoshiki Sato, Shigeru Chiba, and Michiaki Tatsubori. A selective, just-in-time aspect weaver. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 189–208. Springer, 2003.

[SdM03] Damien Sereni and Oege de Moor. Static analysis of aspects. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 30–39. ACM Press, 2003. doi:10.1145/643603.643607.

[Sun] Sun Microsystems Inc. Java authentication and authorization service. Available from: `http://java.sun.com/products/jaas/`.

[Tea01] AspectJ Team. The AspectJ programming guide, 2001. Available from: `http://aspectj.org/doc/dist/progguide/index.html`.

[VRGH+00] 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 *Proceedings of the International Conference on Compiler Construction*, pages 18–34. ACM Press, 2000. Available from: `http://www.sable.mcgill.ca/publications/#CC2000`.

[Xer03] Xerox Corporation. Frequently asked questions about AspectJ, revision 1.8, 2003. Available from: `http://dev.eclipse.org/viewcvs/indextech.cgi/aspectj-home/doc/faq.html`.

[YAR99] Sherif M. Yacoub, Hany H. Ammar, and Tom Robinson. Dynamic metrics for object oriented designs. In *Proceedings of the International Symposium on Software Metrics*, pages 50–61. IEEE Computer Society, 1999. doi:10.1109/METRIC.1999.809725.