McGill University
School of Computer Science
Sable Research Group

Oxford University
Computing Laboratory
Programming Tools Group

Measuring the Dynamic Behaviour of AspectJ Programs

Sable Technical Report No. 2003-8

Bruno Dufour, Christopher Goard, Laurie Hendren and Clark Verbrugge
McGill University
{bdufou1,cgoard,hendren,clump}@cs.mcgill.ca

Oege de Moor and Ganesh Sittampalam
Oxford University
{oege,ganesh}@comlab.ox.ac.uk

December 9, 2003

w w w . s a b l e . m c g i l l . c a

Contents

List of Figures

Abstract

In aspect-oriented programming, a base program is observed by a number of aspects; when a particular pattern of events occurs, the aspects inject new code into the execution of the base program. This dynamic weaving is extremely flexible, and it partly accounts for the success of AspectJ, an aspect-oriented extension to Java. The AspectJ compiler moves much of the dynamic weaving to compile-time.

This paper proposes and implements a methodology for studying the dynamic behaviour of AspectJ programs. As part of this methodology several new metrics specific to AspectJ programs are proposed and tools for collecting the relevant metrics are presented. The major tools consist of: (1) a modified version of the AspectJ compiler that tags bytecode instructions with an indication of the cause of their generation, such as a particular feature of AspectJ; and (2) a modified version of the *J dynamic metrics collection tool which is composed of a JVMPI-based trace generator and an analyser which propagates tags and computes the proposed metrics.

We present a set of benchmarks that exercise the dynamic nature of AspectJ, and the metrics that we measured on these benchmarks. The results provide guidance to AspectJ users on how to avoid efficiency pitfalls, to AspectJ implementors on promising areas for future optimisation, and to tool builders on ways to understand runtime behaviour of AspectJ.

1 Introduction

Aspect-oriented programming [10] offers new ways of modularising a program. An *aspect* is a feature that cross-cuts the traditional abstraction boundaries of classes and methods. Such aspects are woven into an existing base program. What sets aspect-oriented programming apart from earlier meta-programming systems is that such weaving happens (at least conceptually) at runtime. The aspect observes the behaviour of the base program, and when certain sequences of events happen, it injects new code (called *advice*) into the execution. Furthermore, this process is recursive, in the sense that advice execution itself can be observed by the aspect. The events that are of interest to aspects are called *join points*; patterns that trigger the execution of advice are called *pointcuts*. Each join point corresponds to a statement in the source program whose execution gave rise to the join point: this is called a *join point shadow*.

The most popular implementation of these ideas is AspectJ [11], an extension of Java. The AspectJ compiler moves much of the dynamic matching of pointcuts against join points to compile time. Often it is possible to determine statically that execution of a particular statement will always (or never) result in a join point that matches a given pointcut, and in such cases the weaving of advice can be done without the additional runtime overhead of a dynamic check. This can be understood as a form of partial evaluation, and [13] explains the behaviour of the AspectJ compiler in those terms.

AspectJ started out as a pioneering research effort, but in a very short time it has reached a level of maturity where it is being used for production programming. We believe, therefore, that the time is ripe for a systematic examination of the dynamic behaviour of AspectJ programs. In doing so, we hope to provide guidance to AspectJ users about the efficiency of certain idioms. We also aim to identify areas where compilers for AspectJ might be improved, for instance by making use of static analyses to pinpoint more accurately where a particular pointcut will match. Furthermore the results of our dynamic measurements can be fed back into a compiler to identify sections of code that are promising candidates for optimisation. Finally, we aim to provide data to underpin development tools that are specialised to AspectJ, which might range from performance measurement to aspect visualisation.

1.1 Static and Dynamic features of AspectJ

The features that AspectJ adds to Java can be classified into two groups. The first group consists of the dynamic features relating to join points. The second group are features for static cross-cutting, which change the static type structure of classes in the base program. These include the inter-type declarations, by which an aspect can add a new method to an existing class.

The main focus of this paper is the dynamic features relating to join points, because we believe these to be the distinguishing factor in the success of AspectJ. As mentioned earlier, although their semantics is dynamic, many examples can be statically woven. As a concrete example, consider an aspect that counts the calls to a library method $A.f()$. To do so, we define a pointcut that singles out these calls, and a piece of advice that increases the counter. If $f()$ is a static method, we can inject the counting advice at each call to $f()$ in the base program. However, when $f()$

is virtual, we have to dynamically check that the target is indeed of type *A*. In this example, virtual method resolution [23] would help to eliminate some of these dynamic checks. There are other features of the pointcut language where the distinction between static and dynamic is more subtle, for instance the **cflow** construct which checks whether a join point occurred within the dynamic scope of another. In many of the dynamic features, a complicating factor is the possibility of advice applying to itself or other advice, and as we shall see later, this sometimes necessitates the creation of costly closures.

While we focus mainly on pointcut matching and advice introduction, we have also investigated the static features of AspectJ. In principle these could lead to some overheads, for instance because (in the present implementation AspectJ) they can introduce a level of indirection in accessing newly introduced class members. Consequently our experimental results also include some examples that exhibit this phenomenon.

In both the dynamic and static features of AspectJ, the potential overheads are due to their integration in a general-purpose programming language. Of course in each particular example a metaprogramming approach could produce better results, but that would require more effort on the part of the programmer. Our aim is to provide a methodology and tools for discussing the performance price one may have to pay for the generality and convenience of AspectJ.

1.2 Overview

We have definined and implemented a methodology for collecting relevant dynamic measurements of AspectJ programs. This methodology is outlined in Section 2. A key part of our methodology is collecting dynamic metrics. We have accomplished this by modifying two existing tools, the AspectJ compiler [3], and the *J metric tool [6]. The idea is to augment the AspectJ compiler so that bytecodes get tagged with information about the kind of weaving-induced overhead they represent, or for non-overhead instructions, whether they originate from advice or base code. We have 29 kinds of tag to differentiate between different language constructs in AspectJ. The tagged bytecodes are then executed on a virtual machine that collects statistics for each kind of tag. The framework and the process of tagging are detailed in Section 3.

To conduct experiments with our tools, one needs a set of benchmarks, and as there is no commonly available set for AspectJ we constructed one of our own. Our complete benchmark set consists of a number of small applications from the introductory book by Laddad [12], as well as a set of programs that illustrate the use of design patterns from Hanneman and Kiczales [8], and a few from the AspectJ primer [3]. We also included some programs found on the web, as well as an example from previous work by one of the present authors [21]. In choosing these benchmarks, we aimed for code that exercises the dynamic nature of aspects in interesting ways. We also aimed for examples where a pure Java equivalent was available for comparison, so that there is a clear notion of what constitutes overheads in the aspect-oriented version.

We have chosen five relevant benchmarks, and in Section 4 we provide a description of the benchmarks and the results of running our tools on them. Powerful abstractions such as those offered by AspectJ necessarily come with a price, and indeed it turns out that many of our benchmarks show some overheads in terms of run times. Our metrics help identify where these overheads came from, and how they can be remedied. In some cases a simple adjustment to the relevant pointcuts can lead to a vast improvement in performance. We discuss such issues for a number of benchmarks in detail.

While we believe this is the first systematic study of the dynamic behaviour of AspectJ, there is naturally a wealth of related work on collecting dynamic metrics. We discuss these, and also existing efforts to improve the runtime behaviour of AspectJ programs, in Section 5. Finally we discuss our conclusions in Sections 6.

2 Measurements and Dynamic Metrics

In order to study the dynamic behaviour of AspectJ, it was necessary to develop a methodology and tools to collect measurements and dynamic metrics for AspectJ programs. Our approach uses the following three kinds of measurements.

2.1 Execution Time

The most coarse-grain measurement is just the execution time of a program. We use execution time as a first-order measurement of the overheads incurred by using aspects. In particular, we compare the execution time of an AspectJ version of a program and an equivalent Java program. All execution times in this paper were collected on a Athlon XP 1.8 GHz machine with 1 Gbyte of memory running Debian Linux and using Sun's Java™ 2 Runtime Environment, Standard Edition (build 1.4.1_05-b01).

2.2 Java-based dynamic metrics

One would also like more specific measurements of the dynamic behaviour of both the Java and AspectJ versions of benchmarks. Since both Java and AspectJ programs are compiled to Java bytecode, it was possible, using *J [6], an existing tool, to measure relevant dynamic metrics. The *J tool collects a wide variety of metrics, and we have found several metrics to be useful in our evaluation of AspectJ benchmarks.

For example, the base metrics can be used to measure: (1) the total number of bytecode instructions executed, a VM-neutral measure of execution time; (2) the total number of distinct bytecodes loaded and executed, which gives a measurement of total and live program size; and (3) the total number of bytes allocated, which measures how memory hungry the benchmarks are. We can also use more detailed metrics to measure specific behaviours. For example, we can look at the density of important (expensive) operations such as virtual method invocations, field read/writes and object allocations. Specific examples of these metrics are given in the discussion of our benchmarks in Section 4.

2.3 AspectJ-specific dynamic metrics

Although the previous two kinds (execution time and Java-based dynamic metrics) of measurements give a good overall idea of overheads incurred by the use of AspectJ, they do not help to locate the cause of such overheads, nor do they expose any behaviours that are specific to AspectJ programs. In order to study these it was necessary to define new metrics and extend existing tools in nontrivial ways to compute the new metrics. In Section 2.3.1 we describe the new AspectJ-specific metrics and in Section 3 we describe the tools we developed for computing them.

2.3.1 Tag Mix:

The purpose of the *tag mix* metric is to partition all executed bytecode instructions into 29 different bins, where each bin corresponds to a specific purpose, reported as a percent of total executed instructions.

The DEFAULT (0) bin represents all executed instructions that correspond to the base program (regular Java), whereas the ASPECT_CODE (28) bin corresponds to code that was executed as part of the aspect. This includes all non-overhead instructions corresponding to the body of an aspect and all non-overhead instructions in code called from the body. It also includes all non-overhead instructions in methods introduced by inter-type declarations.

Note that this is a dynamic classification and depends on the context in an executing program. For example, the same method may be called from the base program in one context and from an aspect body or a method introduced via inter-type declarations in another context. In the first case the instructions due to the method call called should be tagged DEFAULT (0), whereas in the second case the instructions should be tagged ASPECT_CODE (28). This means that we cannot just statically tag instructions with their bin number, but must dynamically propagate the correct tags while analysing the execution traces.

In making the distinction between base program and aspect, we err on the side of underestimating the effect of aspects, for instance by counting all instructions due to class loading and callbacks from native methods in the DEFAULT (0) bin.

The remaining 27 tags correspond to overhead and are used to account for instructions inserted by the AspectJ compiler to implement the weaving of aspects into the main body of Java code. For example, the tag ADVICE_EXECUTE (1) corresponds to the instructions woven into the base Java code to invoke the method containing the advice code. The tags CFLOW_EXIT (8) and CFLOW_ENTRY (9) signify instructions that perform bookkeeping for the matching of **cflow** pointcuts. A complete list of tags is given in Appendix I, and example measurements of these tags are given in Section 4. An important benefit of our tool set is that it is easy to extend the set of bins, thus giving fine-grained information to language designers and compiler writers about the code emanating from new language features.

2.3.2 Advice Execution:

In many cases the AspectJ compiler can determine statically if a piece of advice should be executed at all join points corresponding to a given join point shadow. In these cases no dynamic test is required to determine if the advice code should be executed or not. There are cases where static analysis cannot determine the applicability of the advice. For example, the **if** pointcut contains a boolean expression which is evaluated to determine join point membership; this expression may contain references to dynamic values, and so it may not be determinable statically whether it evaluates to true or false. The **cflow** pointcut also generally results in a dynamic test.

The *advice execution* metric reports on the outcome of those checks, categorising them into three bins, those that always succeed, those that always fail, and those that sometimes succeed and sometimes fail. Clearly those checks that sometimes succeed and sometimes fail are needed. However, those checks that always succeed or always fail are potential places where a stronger static analysis could eliminate the check, thus eliminating unnecessary overhead and improving performance.

2.3.3 Dead Code and Code Coverage:

In our study of AspectJ benchmarks we found that the AspectJ compiler sometimes includes code that is never executed; in particular, methods that are never called. This dead code can cause an unnecessary overhead on the class loaders, since the entire class must be loaded, even those methods not called, and it also causes unnecessary code bloat in class files. The *dead code* metric measures the number of bytecode instructions that are loaded, but never executed. The *code coverage* metric is computed as the ratio of *live code* over *loaded code*. Thus, a program that loads 10,000 bytecodes and has 2,000 dead bytecodes, has a code coverage of 0.80, that is (10,000 - 2,000)/10,000. It should be noted that the dead code metric is also dynamic and is reporting the code dead for a particular execution of the program. It may be the case that a different execution would touch different parts of the code. Also, in some cases, the dead code may never execute in any execution, but it is a necessary consequence of support for incremental compilation and weaving.

3 Tools for collecting Dynamic Metrics

An overview of the tools that we use for collecting the dynamic metrics is given in Figure 1. The darker shaded boxes correspond to new tools, and the more lightly shaded boxes correspond to components of existing tools that we modified.

Our main tools implement the two important components of our approach: (1) a static tagger which tags bytecode instructions with tags corresponding to their associated purpose; and (2) a dynamic metric analyser that propagates the bytecode tags across method calls, according to the context of the call, and computes the dynamic metrics.

It is not immediately clear why the tags cannot be completely assigned statically. However, there are several reasons why a dynamic tag propagator is needed. First, it may be impossible to determine the appropriate tag for an instruction statically, because the choice of tag may depend on calling context. For example, a method *foo* may be called from within regular Java code and also from within advice, and there are different tags to distinguish these two cases. Second, it alleviates the necessity of annotating all third-party libraries, including the Java and AspectJ runtime libraries since the propagation scheme will propagate the correct tag from the calling method in the user's code to the methods in the library code.

In addition to these main tools we have also developed two utilities. The *Retagger* utility allows us to modify the tags by hand interactively, so that we can experiment with new tagging approaches. The *TagReader* utility allows us to print a textual representation of the tagged bytecode so that we can check its correctness and view the details of which bytecode instructions are tagged.

In the following subsections we first provide an illustrative example, showing examples of static tagging and tag propagation (Section 3.1). We then provide more specific details on the implementation of the two main components

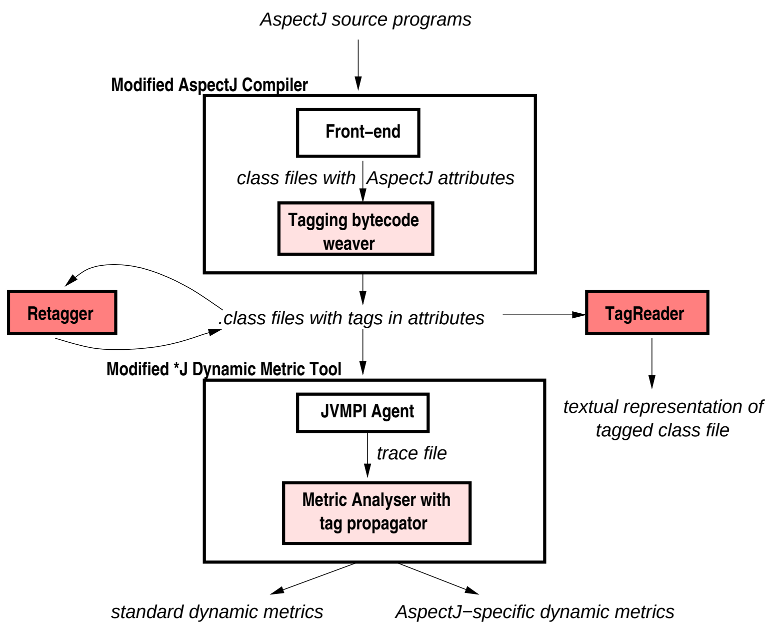
AspectJ source programs
Modified AspectJ Compiler
Front–end
class files with AspectJ attributes
Tagging bytecode weaver
Retagger
.class files with tags in attributes
TagReader
textual representation of tagged class file
Modified *J Dynamic Metric Tool
JVMPI Agent
trace file
Metric Analyser with tag propagator
standard dynamic metrics
AspectJ−specific dynamic metrics

Figure 1: Overview of Metric Collection Tools

of our approach, the static tagger, based on the AspectJ 1.1.1 compiler (Section 3.2), and the dynamic metric analyser, based on *J (Section 3.3) .

3.1 An example

To demonstrate our approach to static tagging and dynamic propagation, consider the small AspectJ program in Figure 2. The advice declared in *ExampleAspect* should execute before every call to *bar*() (selected by the first *call* pointcut) for which there is a call to *foo*() somewhere in the call stack (selected by the *cflow* pointcut).

```
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 {
    before(): call(void Example.bar()) && cflow(call(void Example.foo())) {
        System.out.println("foo->bar");
    }
}
```

Figure 2: Example AspectJ program

The listing in Figure 3 shows the bytecode instructions for each of the methods in *Example.class*, with the added instruction tags that were produced by our static tagger. Each line of bytecode corresponding to instructions introduced by the AspectJ compiler is annotated with the tag associated with it. Many bytecodes do not have a tag and these

```
public void <init>()
   0:  ( )  aload_0
   1:  ( )  invokespecial 9  //Method java/lang/Object.<init>:()V
   4:  ( )  return

public static void main(String[] args)
   0:  ( )  new 2  //class Example
   3:  ( )  dup
   4:  ( )  invokespecial 16  //Method Example.<init>:()V
   7:  ( )  astore_1
   8:  ( )  aload_1
   9:  (3)  getstatic 54  //Field ExampleAspect.ajc$cflowStack$0:Lorg/aspectj/runtime/internal/CFlowStack;
  12:  (3)  invokevirtual 60  //Method org/aspectj/runtime/internal/CFlowStack.isValid:()Z
  15:  (3)  ifeq -> 24
  18:  (2)  invokestatic 47  //Method ExampleAspect.aspectOf:()LExampleAspect;
  21:  (1)  invokevirtual 50  //Method ExampleAspect.ajc$before$ExampleAspect$148:()V
  24:  ( )  invokevirtual 19  //Method Example.bar:()V
  27:  ( )  aload_1
  28:  (9)  bipush 0
  30:  (9)  anewarray 4  //class java/lang/Object
  33:  (9)  astore_3
  34:  (9)  getstatic 54  //Field ExampleAspect.ajc$cflowStack$0:Lorg/aspectj/runtime/internal/CFlowStack;
  37:  (9)  aload_3
  38:  (9)  invokevirtual 64  //Method org/aspectj/runtime/internal/CFlowStack.push:([Ljava/lang/Object;)V
  41:  ( )  invokevirtual 22  //Method Example.foo:()V
  44:  (8)  goto -> 58
  47:  (8)  astore 4
  49:  (8)  getstatic 54  //Field ExampleAspect.ajc$cflowStack$0:Lorg/aspectj/runtime/internal/CFlowStack;
  52:  (8)  invokevirtual 67  //Method org/aspectj/runtime/internal/CFlowStack.pop:()V
  55:  (8)  aload 4
  57:  (8)  athrow
  58:  (8)  nop
  59:  (8)  getstatic 54  //Field ExampleAspect.ajc$cflowStack$0:Lorg/aspectj/runtime/internal/CFlowStack;
  62:  (8)  invokevirtual 67  //Method org/aspectj/runtime/internal/CFlowStack.pop:()V
  65:  ( )  nop
  66:  ( )  return
Exception Handlers:
from  to  target  type
41 44 47 java.lang.Throwable(73)

public void foo()
   0:  ( )  getstatic 31  //Field java/lang/System.out:Ljava/io/PrintStream;
   3:  ( )  ldc 32  //String foo
   5:  ( )  invokevirtual 38  //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   8:  ( )  aload_0
   9:  (3)  getstatic 54  //Field ExampleAspect.ajc$cflowStack$0:Lorg/aspectj/runtime/internal/CFlowStack;
  12:  (3)  invokevirtual 60  //Method org/aspectj/runtime/internal/CFlowStack.isValid:()Z
  15:  (3)  ifeq -> 24
  18:  (2)  invokestatic 47  //Method ExampleAspect.aspectOf:()LExampleAspect;
  21:  (1)  invokevirtual 50  //Method ExampleAspect.ajc$before$ExampleAspect$148:()V
  24:  ( )  invokevirtual 19  //Method Example.bar:()V
  27:  ( )  return

public void bar()
   0:  ( )  getstatic 31  //Field java/lang/System.out:Ljava/io/PrintStream;
   3:  ( )  ldc 39  //String bar
   5:  ( )  invokevirtual 38  //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   8:  ( )  return
```

Figure 3: Tagged class file for example AspectJ program

bytecodes will be assigned a tag during the subsequent dynamic analysis. Let us now examine the static tagging and dynamic tag propagation for our example.

Instructions 9–15 in both *main*(*String*[]) and *foo*() are tagged as ADVICE_TEST (3); these instructions perform the matching of the *cflow* pointcut, and test for the presence of a call to *foo*() in the call stack. If this test succeeds, the advice is executed.

Instructions 18–21 in both methods are advice execution overhead, tagged as ADVICE_ARG_SETUP (2) and ADVICE_EXECUTE (1). The distinction is made between these two tags because they propagate differently. Instruction 18 is a call to the *aspectOf*() method, which acquires the aspect instance. All of the untagged instructions in *aspectOf*() will inherit the tag of instruction 18 (ADVICE_ARG_SETUP (2)), as they also represent the same kind of overhead. Instruction 21, however, calls the advice body, which is not overhead, and so its tag is not propagated by the analyser. Instead, the ASPECT_CODE (28) tag is propagated to the advice body method.

Instructions 28–38 (CFLOW_ENTRY (9)) and 44–62 (CFLOW_EXIT (8)) manage the representation of the call stack, required by the *cflow* pointcut. This call stack representation is described in more detail in section 4. Before each call to *foo*(), a value is pushed onto the *CFlowStack* corresponding to the relevant *cflow* pointcut. On returning from that call, either normally or by thrown exception, the *CFlowStack* is popped. Both of these tags, CFLOW_ENTRY (9) and CFLOW_EXIT (8), propagate to the called methods since the *push* and *pop* methods represent the same kinds of overhead.

We have not provided the listing of the tagged aspect class in our figure. The only interesting tags for this class are in the static initializer, whose instructions are all tagged CLINIT (13), and the return instructions in the advice bodies, which are tagged ADVICE_EXECUTE (1).

3.2 Static Tagging: annotating class files using a modified AspectJ compiler

As shown in the top part of Figure 1, we have developed a modified version of the AspectJ 1.1.1 compiler where the class files produced are annotated using attributes in order to identify bytecode instructions that originated from one of the special features of AspectJ, and to distinguish between different kinds of such features. The annotations map instructions to tags.

3.2.1 Tagging during weaving:

The AspectJ compiler, since version 1.1, operates in two stages. The first is a compilation stage, using the Java compiler from the Eclipse project, and it produces class files with special attributes. These attributes contain information for the second stage, where aspects are woven into the existing bytecode. We have modified the bytecode weaver of version 1.1.1 to annotate the classes it produces.

In the AspectJ compiler, the major changes made to the classes being woven into are performed by two kinds of *munger*. The first is the *type munger*, which is responsible for changing the type structure of the program and implements inter-type declarations. The second is the *shadow munger*, and it is responsible for manipulating join point shadows, implementing, for example, the weaving in of advice. Consider the simple case of the **before** advice declared in the example in Figure 2. During the weaving stage this advice is represented by a shadow munger which operates on shadows for which a subset of associated join points are selected by the advice's pointcut. The body of the advice is compiled as a method on the aspect class during the compilation stage; the shadow munger inserts into the shadow the instructions necessary for calling this advice body method, and, if necessary, test instructions to determine at runtime if a join point matches the pointcut.

Our modified AspectJ weaver tags all the instructions according to their purpose. The first set of new instructions created by the weaver expose arguments to the advice and acquire the aspect instance. We add as attributes to each generated instruction object the ADVICE_ARG_SETUP (2) tag. Then the advice execution instruction is created, which is an invoke to the advice body method. We tag this ADVICE_EXECUTE (1) in the same way. Finally, if it hasn't been statically determined that this advice should always execute at this shadow, test instructions are generated, which we tag as ADVICE_TEST (3). This newly generated instruction list is then inserted into the shadow, which is a range of instructions on a method in the base program.

Our examples so far have demonstrated some of the most common tags. However, the AspectJ weaver introduces new instructions into the base program to implement many other features, both as a result of static cross-cutting and dynamic cross-cutting. All such overhead instructions have been captured by a comprehensive set of tags, which are described in detail in Appendix I.

3.2.2 Pretagging:

It is important to note that not all of the instructions we wish to annotate are generated by the weaver. There are some methods on the aspect classes, generated during the front-end AspectJ compilation, that we wish to tag. Therefore, before weaving into an aspect class, we tag these instructions by searching for bytecode patterns in methods matching naming conventions. An example case is that of an **around** advice body; we must tag the instructions corresponding to the **proceed**() call, leaving the other instructions untagged.

3.2.3 Generating attributed classfiles:

After all tagging and weaving has been performed on all classes, and as classes are being written, our modified AspectJ compiler converts the tag attributes on the instruction objects into a code attribute for each method which is stored in the generated class files. For those instructions with explicit tags we use that tag value, and for instructions without tags a placeholder tag is assigned, namely NO_TAG (-1). This will be replaced by a proper tag during the dynamic analysis phase.

3.3 Dynamic metric analysis with tag propagation: a modified *J tool

Dynamic metrics are computed using *J, a framework designed to allow new dynamic analyses, in particular dynamic metric computations, to be implemented quickly and easily. *J uses a trace collection agent which is based on the Java Virtual Machine Profiler Interface (JVMPI). This agent receives execution events from a regular Java Virtual Machine (JVM) and encodes the information in the form of an event trace. This trace can then be processed by the analyser, which internally consists of a sequence of operations organised as a pipe. Each analysis in the pipe receives events from the trace sequentially. *J provides a number of default analyses in its library, many of which provide services to subsequent analyses in the pipe. It also includes a full set of general-purpose dynamic metric computations.

3.3.1 Modifications to the *J analyser:

Since tags are stored in class file attributes which store both the bytecode offset and associated tag, they need not be recorded at the profiling level. *J records bytecode executions using offsets in the code arrays, and thus the mapping between bytecode instruction and associated tag can be done later, during analysis. Thus, the JVMPI agent of *J does not need to be modified: all modifications can be done in the trace analyser component.

The trace analyser component was modified to deal with collecting the tag metrics specific to AspectJ, and to correctly propagate tags. The most significant change is the propagation of tags. This propagation is accomplished by pushing bytecode tags along invocation edges in the dynamic call graph of the application under certain constraints. Every method execution thus has a tag obtained from the propagation scheme, and if there is no such tag it is assumed to be DEFAULT (0). However, tags which have been statically assigned by the modified AspectJ compiler correspond to overhead, and will never be overwritten, with the exception of ASPECT_CODE (28), whose case is further discussed below, and the special NO_TAG (-1) tag.

Since a given untagged bytecode instruction can be assigned different dynamic tags during the analysis, it must be possible to differentiate between dynamically and statically assigned tags. Moreover, not every tag can be propagated along call edges; it is often necessary to propagate a different tag than the current one for the invoke bytecode instruction. For example, a tag which indicates that an invocation is made to a piece of advice will not be propagated during the analysis; a tag which identifies bytecodes as being part of aspect code is propagated instead.

The propagation policy is defined via a table mapping tags to propagated tags. Of the 29 tags listed in Appendix I, 24 map to themselves. The exceptions are the following. ADVICE_EXECUTE (1) represents a call to an advice body method, and so maps to ASPECT_CODE (28). Both of INTER_METHOD (14) and INLINE_ACCESS_METHOD (27) also map to ASPECT_CODE (28). Two tags that relate to **around** map to DEFAULT 0, namely AROUND_CALLBACK (24) and AROUND_PROCEED (25), since they both represent calls to non-advice user-defined code.

Above we mentioned that statically assigned tags can never be overwritten, with ASPECT_CODE (28) as an exception. The reasoning behind this is as follows. An instance of an aspect can be accessed via the static method *aspectOf*() on the aspect class. This call can be made both from within the code inserted by the weaver to implement advice execution, and from within user-defined code. In the first case, the invoke is tagged ADVICE_ARG_SETUP (2), which we wish to propagate to the method. The method, however, is tagged ASPECT_CODE (28) in order to support the second case, in which the invoke is untagged, on account of being user-defined code. To deal correctly with this situation in the analyser, it is necessary that instances of the tag ASPECT_CODE (28) be overwritable during analysis by particular tags.

The entire tag propagation scheme is implemented as a separate *J analysis, so that subsequent AspectJ-related analyses can be implemented independently. This segregation of the tag propagation concern allows for an easy implementation of the analyses which compute the AspectJ-specific metrics.

3.3.2 Collecting the AspectJ-specific metrics:

With the tags correctly propagated, it is a simple addition to the *J analyser to collect the *tag mix* metric, since each instruction now has a tag. As it executes, the corresponding bin counter is incremented. In addition, the analyser also tracks all dynamic guards on advice, and for each such guard computes whether the guard always succeeds, always fails, or sometimes succeeds. In addition, a count of the number of times each guard is executed is maintained. The *dead code* and *code coverage* metrics are straightforward extensions of metrics already implemented in *J.

4 Benchmarks

In this section we provide the results and analysis for five benchmarks. Since there is no existing AspectJ benchmark set, we collected our benchmarks from a variety of sources. All benchmarks have an AspectJ version and an equivalent Java version so that direct comparisons can be made. We have configured the benchmarks so that they all run long enough to make meaningful runtime measurements, but short enough so that the traces generated by the *J tool remain a reasonable size. All benchmarks can be made to run longer by tuning some parameters.

The main point of this section is to to demonstrate our methodology on a variety of different kinds of AspectJ programs. More specifically, we show how to use the runtime measurements, the ordinary dynamic metrics and the AspectJ-specific metrics to understand the performance and overheads in the benchmarks. As a side-effect of our analysis we have also made some interesting observations about the overheads involved in the weaving approach taken in the AspectJ compiler and we point out where further improvements could be worthwhile.

4.1 Quicksort

This benchmark was adapted from [20], where it is presented in an aspect-oriented extension of Pascal. Its purpose is to instrument an implementation of Hoare's quicksort algorithm. The implementation of that algorithm proceeds in the usual way, namely by first partitioning the input array around a pivot, and then sorting each of the partitions recursively. It is a common exercise in introductory programming courses to write the *partition* method so that it makes as few *swap*s in the array as possible. The instrumentation suggested by [20] is therefore to count the number of calls to *partition* and *swap* in each invocation of *quicksort* that is not caused by *quicksort* itself.

In the pure Java version, this is done by introducing the counters, and directly manipulating them in the program. In the AspectJ version, we have striven for pointcuts that assume as little as possible about the program that is being instrumented. This is a common design goal when writing AspectJ programs, as pointcuts can easily become very brittle with respect to changes in the base program. In particular, we only wish to count calls to *swap* that occurred within the dynamic scope of *quicksort*, as there might be other places where *swap* is used as well. This is captured by the pointcut

	Java	AspectJ	**cflow** counter
WHOLE PROGRAM			
time (seconds)	4.60	9.25	5.10
instructions (million bytecodes)	1752	3137	2475
allocated (million bytes)	36	191	36
APPLICATION ONLY			
classes loaded	3	8	6
instructions loaded	169	688	457
instructions dead	6	236	122
code coverage	0.97	0.66	0.73
byte allocation density	10.4	39.0	6.2
field access density	53.4	66.3	78.3
invoke density	39.5	72.9	60.8
ASPECTJ TAG MIX (WHOLE PROGRAM) (%)			
DEFAULT (0)		52.6	
ASPECT_CODE (28)		5.9	
ADVICE_ARG_SETUP (2)		4.2	
ADVICE_TEST (3)		17.3	
CFLOW_EXIT (8)		10.1	
CFLOW_ENTRY (9)		8.9	
OTHER		0.9	
Total aspect overhead		41.5	
ADVICE EXECUTION (%)			
NEVER		50	
SOMETIMES		0	
ALWAYS		50	

Figure 4: Quicksort metrics

call (**private static void** swap(Object[],**int**,**int**)) &&
cflow (**call** (**public static void** QuickSort.quicksort(Comparable[],**int**,**int**)))

One can read this as a pattern that the call stack must satisfy. The first part specifies that *swap* appears at the top of the stack. The second part **cflow**$(\ldots)$ states that *quicksort* appears somewhere lower on the call stack. When a join point satisfies this pointcut, we apply **before** advice to increase the counter of swap operations.

Similarly, a robust way to pick out the non-recursive calls to *quicksort* is to specify that *quicksort* is itself at the top of the call stack, but there are no further occurrences below it. In AspectJ, this is conveniently expressed using **cflowbelow**. This pointcut has both **before** and **after** advice, to initialise the counters and to print the results, respectively.

We sorted an array of 1 million *Integer* values; the aspect-oriented version takes about twice as long as the pure Java version. In a first attempt to explain these overheads, we computed some of the metrics from [6], and these are displayed in the top half of Figure 4. The third column refers to an optimisation that the AspectJ compiler could have applied; it is further discussed below.

From the whole-program statistics, it is immediately clear that a very large proportion of the work goes into new allocations. This is further confirmed by the application-only statistics, which shows that the byte allocation density increases four fold. Allocation is not the only source of the overheads, however: field access and invoke density also increase significantly.

To gain further insight, it is illuminating to examine the AspectJ-specific metrics, which are displayed in the remainder of Figure 4. We first show what percentage of instructions can be traced back to certain features of AspectJ. Only about half of the instructions are part of the program proper, whereas 5.5 percent is spent in executing advice. The rest is overhead, and it consists of the time needed to obtain a reference to an aspect instance, to test dynamically whether advice is applicable, and the administration of the **cflow** matching. The final section of the table is concerned with tests in the program text that perform pointcut matching. Here it becomes apparent that half of them always succeed, while the other half never succeed.

To understand how these overheads come about, it is necessary to expand a little on the way **cflow**(P) is implemented by the AspectJ compiler. Essentially it matches the pointcut P against all prefixes of the call stack (read from top to bottom). To implement this efficiently, AspectJ creates an explicit stack, which is pushed and popped upon each method call that might influence the matching of P. The entries of this stack can be thought of as states in the matching automaton for P itself, and in particular any variable bindings that are made in P (for instance via **args**) are stored on the stack. A detailed discussion of this implementation can be found in [13].

In our example, three such stacks are created, one for each piece of advice that has a pointcut involving **cflow**(P). The entries of these are however extremely simple, because P just matches calls to *quicksort*. The AspectJ compiler represents all these entries by zero-length arrays of type *Object*. The representation as arrays is necessary to represent the states of a matching automaton for each instance when **percflow** is used, but in our example there is only one instance of the aspect. The use of arrays of objects thus leads to a great deal of unnecessary allocation.

An obvious optimisation is to represent these stacks of dummy matching states as integer counters. To test the effect of this transformation, we decompiled the output of AspectJ using Soot's Dava decompiler [15, 16, 19]. The decompiled source was then edited by hand, replacing the use of the *CFlowStack* type by a *CounterStack*, which maintains a counter that is incremented upon each push, and decremented upon each pop. This trivial change reduces the runtime to 5.1 seconds (a reduction of 45%) and it would clearly be beneficial to implement the transformation in AspectJ itself. The third column of the table in Figure 4 only shows the standard metrics, because the Dava decompiler produces Java and not AspectJ source.

Naturally it would be even better to eliminate the overheads of **cflow** altogether. The metrics in Figure 4 seem to suggest that this is possible, for all tests in the program text always succeed, or they always fail. Indeed, this elimination can be achieved by computing a superset of all possible call stacks at each join point shadow. For each of these sets, we can now determine whether a **cflow** pointcut will always match, never match, or match sometimes. The details of this analysis are worked out in [20], for a Pascal-like language. In particular, it is shown there how it automatically leads from the aspect-oriented version of quicksort to the efficient tangled version without aspects. That paper also proves that there are examples for which no such static weaving is possible, so the implementation via explicit stacks cannot be dispensed with completely.

4.2 Coding Standards — checking for return of null

Users of AspectJ have found many different kinds of applications for aspects. One potential use, as outlined in a short online article by Asberry, is to use aspects to enforce coding standards [2]. He suggests several applications, one of them being an aspect to detect when methods return null. According to Asberry, the justification for this aspect is that sometimes programmers use the *"on error condition, return null from method"* anti-pattern. This is considered to be a bad coding style, since throwing a meaningful exception would be much preferable. He suggests the following pointcut and **around** advice to detect all occurrences of returning *null* from a method.

```
// First primitive pointcut matches all calls, second avoids those with void return type.
pointcut methodsThatReturnObjects(): call(* *.*(..)) && !call(void *.*(..));

Object around(): methodsThatReturnObjects()
{ Object lRetVal = proceed();
  if (lRetVal == null)
    { System.err.println( "Detected null return value after calling " +
          thisJoinPoint.getSignature().toShortString() + " in file " +
          thisJoinPoint.getSourceLocation().getFileName() + " at line " +
          thisJoinPoint.getSourceLocation().getLine());
    }
   return lRetVal;
}
```

Since this seemed to be a very simple aspect that could be applied to any program, we applied it to a Java benchmark, *Certrevsim*, which is a discrete event simulator used to simulate the performance of various certificate revocation

	Orig. Java *(no null checks)*	Checked Java *(with null checks)*	Orig. AspectJ *(all non-void methods)*	Fixed AspectJ *(only Object+ methods)*	Pruned AspectJ *(not within aspect code)*
WHOLE PROGRAM					
time (seconds)	1.34	1.50	30.82	10.10	1.83
# instr. (million bytecodes)	856	906	4840	1869	1256
mem. alloc. (million bytes)	1.8	1.8	5612	1494	2.3
APPLICATION ONLY					
classes loaded	22	22	252	138	48
instructions loaded	2237	2421	13927	8483	7633
instructions dead	901	1051	6893	4959	4864
code coverage	.60	.57	0.51	0.42	0.36
ASPECTJ TAG MIX (WHOLE PROGRAM) (%)					
DEFAULT (0)			31.9	50.7	79.2
ASPECT_CODE (28)			3.6	2.7	5.0
ADVICE_EXECUTE (1)			1.8	1.3	2.0
ADVICE_ARG_SETUP (2)			23.6	19.3	6.9
AROUND_CONVERSION (23)			6.7	0.7	1.0
AROUND_CALLBACK (24)			16.1	13.3	0.0
AROUND_PROCEED (25)			7.2	5.3	5.9
CLOSURE_INIT (26)			9.0	6.7	0.0
Overhead (total)			64.4	46.6	14.8

Figure 5: Nullcheck metrics

schemes [1]. This seemed to be a suitable benchmark because it has non-trivial uses of data structures and it computes something useful.

Our first experiment was to analyse the dynamic behaviour of the original benchmark and compare it with the same benchmark, but with the suggested null check aspect applied to it. The data are given in Figure 5, where the data for the original Java benchmark is in the column labelled “Orig. Java” and the same benchmark with the suggested aspect applied is in the column labelled “Orig. AspectJ”. The results were very surprising, as the Java benchmark runs in 1.34 seconds, but the AspectJ benchmark runs in 30.82 seconds, a 23-fold slowdown. This was completely unexpected, because according to the description of the aspect, the only new useful code being inserted is a check of the return value of all non-void methods.[1] To verify that such checks should not account for such a slowdown we hand-wove the checks into the original program, and the dynamic measurements for this version are given in the column labelled “Checked Java”. As expected, the additional checks on return values only slows the benchmark down by 19%.

In order to understand what was happening, we decompiled the classes produced by the AspectJ compiler using Soot’s Dava decompiler and discovered that the **around** advice was being applied to **all** method calls returning values (including methods returning scalar types such as integers) instead of just those that returned values with some Object type (i.e. any type that is Object or a subclass of Object). Of course, looking back to the pointcut, erroneously named *methodsThatReturnObjects*, we can see that it does apply to all methods with non-void return type. Thus, we fixed the pointcut designator to be the following.

```
pointcut methodsThatReturnObjects(): call(Object+ *.*(..));
```

This fixed pointcut matches only those method calls which return Object types, as intended, and the dynamic measurements of applying this fixed pointcut to the simulator benchmark are given in Figure 5, in the column labelled “Fixed AspectJ”. Note that the runtime is still much larger than expected, 10.10 seconds, or 7.5 times slower than the original Java program which ran in only 1.34 seconds.

The WHOLE PROGRAM dynamic metrics give us some insight into this large performance difference. The fixed AspectJ version executes 1869 million instructions, whereas the original Java version executes only 856 million in-

[1]It turns out that the *Certrevsim* benchmark is well written and does not return null from methods, so the check against null never succeeds. Thus, the runtime overhead is simply the check against null and a branch.

structions. However, most surprising is that the fixed AspectJ benchmark allocates 1494 million bytes, whereas the original Java version only allocated 1.8 million bytes. This is a huge increase in memory consumption, considering the aspect body itself is very simple, the check against null never succeeds in this benchmark, and thus the aspect body does not explicitly allocate any objects at all.

When we look at the APPLICATION ONLY dynamic metrics we see that the original Java benchmark loaded only 22 application classes (2237 instructions), whereas the fixed AspectJ version loaded 138 classes (8483 instructions), another source of overhead for the class loader and JIT compiler.

By looking at the ASPECTJ TAG MIX metrics we can see there is a large amount of overhead, mostly coming from ADVICE_ARG_SETUP (2), AROUND_CALLBACK (24), AROUND_PROCEED (25) and CLOSURE_INIT (26). Given that all overhead was coming from **around** advice, we decompiled the class files and studied the code generated by the AspectJ compiler to implement the **around** advice. We found that, in this case, closures are created to handle the **around** advice. By studying the code produced we estimated that each method call with **around** advice has an overhead of 2 invokespecial calls, 5 invokestatic calls, 2 invokevirtual calls, 2 array allocations, 3 object allocations, 3 field read/write instructions, 4 cast/instanceof instructions, plus numerous simple load and store instructions. Clearly this use of closures is a very heavy weight solution, using many expensive bytecode instructions and considerable memory allocation, and it certainly accounts for the increase in runtime.

In order to understand why closures were being used to implement the **around** advice for such a simple case, we studied the AspectJ compiler and found that there are two strategies for implementing **around** advice, one uses closures and the other uses an inlining strategy. By default the compiler will try to inline; however there are two situations in which closures will be used: (1) the compiler flag -XnoInline has been set; or (2) the **around** body has **around** advice which applies to it. For our benchmark, the body of the **around** advice contains several method calls returning Object types (namely the string operations in the argument of *println*), so situation (2) applies and so the AspectJ compiler selects the closure strategy for *all* method calls which have this kind of **around** advice applied.

To study the performance of the inlining strategy, we changed the pointcut designator to eliminate those method calls that were in our aspect code as follows.

```
pointcut methodsThatReturnObjects(): call(Object+ *.*(..)) && !within(lib.aspects..*);
```

The dynamic measurements of this version are given in Figure 5 in the column labelled “Pruned AspectJ”. Clearly the inlining strategy for **around** advice is much more efficient than the closure strategy. However, it is somewhat alarming that such a minor change to the pointcut specification has such a large impact on the performance of the program. From the programmer’s point of view, the !**within** clause should not be necessary, but clearly it does have a very important impact on the ultimate performance. Furthermore, there is still a significant amount of overhead when we compare the hand-woven Java version (column labelled “Checked Java”) to the equivalent AspectJ version (column labelled “Pruned AspectJ”).

In terms of runtime performance, the hand-woven Java version executes in 1.5 seconds whereas the AspectJ version executes in 1.83 seconds, which is 22% slower. This overhead is also reflected in the number of instructions executed, 906 million for the Java version versus 1256 million for the AspectJ version. According to the ASPECTJ TAG MIX metrics, most of the overhead is due to ADVICE_ARG_SETUP (2) and AROUND_PROCEED (25).

Furthermore, the AspectJ program loads more application classes (48 vs. 22), because the AspectJ version must load many classes from the AspectJ runtime library, and the aspect class itself. The AspectJ version has more instructions (7633 vs. 2421), which is due to code from the AspectJ runtime library, the inlining of multiple copies of advice, and the fact that the inlining strategy introduces many overhead instructions, as demonstrated by the ASPECTJ TAG MIX metrics.

Finally, the AspectJ version has significantly more dead code (4864 vs 1051). The dead code comes from three sources: (1) methods in the AspectJ runtime library that are loaded, but never run, (2) the code in the never-taken branch of the advice which is inlined in many places, and (3) the presence of methods generated by the Aspect compiler which are never needed (for example, a method to deal with advice as closures is generated even if closures are not used). We believe AspectJ generates these dead methods for reasons of incremental compilation.

There are some important observations to be made with this benchmark. First, even though the pointcut in this example was very simple, it shows that it is very easy for a programmer to define a pointcut that applies to more places than absolutely necessary. Further, the decision of the AspectJ compiler to use closures or inlining for **around** advice can have a huge impact on runtime, due to the general, but heavy-weight, strategy used for closures. Programmers may

	Java	AspectJ
WHOLE PROGRAM		
time (seconds)	1.52	1.53
# instructions (million bytecodes)	345	356
memory allocated (million bytes)	199	199
APPLICATION ONLY		
classes loaded	7	11
instructions loaded	222	258
instructions dead	20	30
code coverage	0.90	0.88
ASPECTJ TAG MIX (WHOLE PROGRAM) (%)		
DEFAULT (0)		50.8
ASPECT_CODE (28)		46.9
INTERMETHOD (14)		2.2
Overhead (total)		2.2

Figure 6: Visitor metrics

unwittingly trigger the use of closures if they forget, or don't realize, the importance of avoiding pointcuts that apply in the aspect body. The inlining strategy for **around** advice is much more efficient than the closure-based strategy, but it can still lead to significant overheads, particularly if applied to method calls that execute frequently. Thus, we feel that this example shows that it would be worthwhile to further improve the approach to generating code for **around** advice.

4.3 Visitor

This benchmark was taken from Hannemann and Kiczales' study of the "Gang of Four" design patterns in AspectJ [8]. It consists of a Java and an AspectJ implementation of the Visitor design pattern [7]. We took their code and modified the "driver" class to increase the running times enough to gather useful data. In a loop, it constructs a tree of five nodes and two vistor objects: one that sums the values stored at all leaf nodes and one that prints out a string representation of the structure of the tree. The AspectJ implementation uses only static cross-cutting — the aspects use inter-type declarations to assign roles to the participant objects and introduce new methods on them. This static introduction of methods results in the introduction of dispatch methods.

In such a simple example, it is highly likely that the JIT will eliminate most of the overheads from the introduction of these methods, and indeed we see from Figure 6 that there is a negligible difference between the Java and the AspectJ versions in terms of running times and bytes allocated. Indeed, the overhead in terms of number of extra instructions executed is also small — less than 2 percent.

4.4 Strategy

This benchmark was also taken from Hannemann and Kiczales' study [8]. It consists of a Java and an AspectJ implementation of the Strategy design pattern [7]. Again, we modified the "driver" class to increase the running time, so that it now consists of a loop in which an array of numbers is sorted using one of two randomly assigned strategies: bubblesort and linearsort.

The idea behind the Strategy pattern is that a particular algorithm for carrying out a specified task (the *strategy*) is assigned to a *context*. Code that requires this task to be carried out will invoke the algorithm via the context, thus allowing the choice of algorithm to be logically separated from the point at which it is used.

The AspectJ version allows any object to be made into a context, using static cross-cutting, and it uses **around** advice to select the appropriate strategy when the appropriate method of such a context is invoked.

Our results for this benchmark are in Figure 7. The "AspectJ (hash table)" column contains the results from the AspectJ code provided by the authors of [8]. As with the Visitor benchmark, the number of instructions that were

	Java	AspectJ (hash table)	AspectJ (field)
WHOLE PROGRAM			
time (seconds)	1.36	1.64	1.40
instructions (million bytecodes)	1932	2166	2022
allocated (million bytes)	23	1	1
APPLICATION ONLY			
classes loaded	5	9	9
instructions loaded	219	362	351
instructions dead	30	98	91
code coverage	0.86	0.73	0.74
ASPECTJ TAG MIX (WHOLE PROGRAM) (%)			
DEFAULT (0)		10.9	7.2
ASPECT_CODE (28)		88.3	90.8
ADVICE_EXECUTE (1)		0.1	0.1
ADVICE_ARG_SETUP (2)		0.7	0.8
INTERFIELDGET (15)			0.3
INTERFIELDSET (16)			0.8
Overhead (total)		0.8	2.0
INSTRUCTION COUNT BREAKDOWN (WHOLE PROGRAM) (MILLION BYTECODES)			
Java	1932	235	146
Aspect body		1913	1837
"Real" code (total)	1932	2148	1983
Overhead (total)		18	40

Figure 7: Strategy metrics

overhead is small; however, we did notice that the time taken to execute this benchmark increased by about twenty percent between the Java and the AspectJ versions. Since the total overhead from the AspectJ compiler was small, it seemed unlikely that this was the cause of the slowdown, so we looked closely at the number of non-overhead instructions executed and noticed that this had also increased by about twenty percent — the relevant figures are in the final section of Figure 7. In other words, in the AspectJ version the combination of the program and the Strategy aspect was doing more work than the original program was.

On further investigation, we found that the Strategy aspect was using a global hash table to keep track of the current strategy for a particular context. It seemed to us that a better approach would be to simply store the strategy in the context, and indeed we found that a field for this very purpose was being added to every class designated as a context, but not being used. Changing the code to use this field rather than the hash table left us with the results in the third column of Figure 7, confirming that it was indeed the hash table that was largely responsible for the slowdown.

Of course, we do not intend any criticism of the authors of this code, which was produced merely as a proof of concept when they were investigating how well AspectJ could be used to modularise the application of design patterns. We include it simply because it provides a good illustration of how our methodology helps in investigating the causes of performance problems with AspectJ programs.

4.5 Bean

This is example is taken from the AspectJ primer on `aspectj.org`. Once again, we modified it slightly to increase the running time. It starts with a class named *Point* for representing pairs of *x* and *y* coordinates, and it adds the functionality of Java beans with bound properties to this class.

In order to do so, it injects a new private field into the *Point* class; this new field has type *PropertyChangeSupport*; all the associated methods are added as well, and the *Point* class is declared to be an implementation of *Serializable*. All these additions are accomplished via the static features of AspectJ. Furthermore, it also fires a property changer

	Java	AspectJ	AspectJ (alternative)
WHOLE PROGRAM			
time (seconds)	1.44	1.71	1.49
instructions (million bytecodes)	118	208	144
allocated (million bytes)	108	129	109
APPLICATION ONLY			
classes loaded	2	29	5
instructions loaded	272	1800	529
instructions dead	43	1123	160
code coverage	0.84	0.38	0.70
ASPECTJ TAG MIX (WHOLE PROGRAM) (%)			
DEFAULT (0)		9.1	10.5
ASPECT_CODE (28)		83.3	78.5
ADVICE_EXECUTE (1)		0.3	0.8
ADVICE_ARG_SETUP (2)		2.3	2.9
INTERMETHOD (14)		0.4	0.6
INTERFIELD_INIT (17)		1.0	1.4
AROUND_PROCEED (25)		1.6	2.4
INLINE_ACCESS_METHOD (27)		2.0	2.9
Total aspect overhead		7.6	11.0
INSTRUCTION COUNT BREAKDOWN (WHOLE PROGRAM) (MILLION BYTECODES)			
Java	118	19	15
Aspect body		174	113
"Real" code (total)		193	128
Overhead (total)		16	16

Figure 8: Bean metrics

whenever either the *x* or *y* coordinate is changed. This additional functionality is achieved with a pointcut and **around** advice.

For comparison, we wove the AspectJ version by hand to obtain a pure Java program. In doing so, we were able to eliminate a number of dynamic tests, because the **around** advice employs reflection to test whether it is *x* or *y* that is being changed. Of course that can be statically determined, and the same optimisation would have been obtained via constant propagation and branch elimination in an optimising compiler.

The results for these versions is shown in Figure 8. As with the *strategy* benchmark, there was a significant disparity between the Java and AspectJ versions which the instruction count breakdown showed was not caused by overhead. The changes we were able to make in constructing the Java version suggested a possibility for improving the AspectJ version — namely, specialising the **around** advice for the separate cases of *x* and *y* being changed. This produced the results shown in the third column of Figure 8, which show performance significantly closer to the hand-woven version.

4.6 Summary of Benchmarks

In this section we have provided the results and analysed five AspectJ benchmarks according to our methodology. We used the running times of Java and AspectJ versions of the benchmarks as a first-order measurement to determine if there existed any significant overhead in the AspectJ version. We then used the ordinary dynamic metrics to see the differences in instruction counts, memory allocation, class loading, live instructions and various density metrics. Finally, we used the new AspectJ-specific metrics to find the most relevant overheads and to measure the branching behaviour of dynamic advice guards (in the *quicksort* benchmark).

Our analysis led to several interesting observations, and of our five benchmarks, we found significant overheads in *quicksort*, *nullcheck* and *bean*. The *quicksort* benchmark demonstrates overheads due to **cflow**, the *nullcheck* benchmark demonstrates overheads due to **around** advice and highlights the large performance difference between the different strategies for weaving around advice. The *bean* benchmark shows that overhead can come from several sources, and when combined can be significant. We did not find any significant overhead in the *visitor* and *strategy* benchmarks. However, our metrics did help identify an inefficiency in the aspect code in the *strategy* benchmark.

Clearly a benchmark collection of five benchmarks does not provide for an exhaustive study of the performance of AspectJ. We are actively collecting a larger benchmark set and we would be very pleased to receive more contributions. Our benchmark set, including the benchmarks presented in this paper, will be made publicly available in conjunction with the preparation of a final version of this paper.

5 Related Work

Most work on dynamic metrics has centered on either addressing a specific optimisation problem such as memory use (e.g. [5, 22]), or more generally (and voluminously) on software engineering quality or complexity measures (eg [14, 24, 26]). More related work on analyzing programs through metrics is given in [6], along with a description of our overall approach.

The performance of AspectJ programs has also been discussed and investigated in the literature, and typically it is assumed or demonstrated to some degree that aspects do not impose unreasonable overhead. Kiczales et al's overview paper of AspectJ [11] for instance makes the pronouncement that (with respect to **before**/**after** advice) "...there should generally be no observable performance overhead from these additional method calls." Method calls inserted into code to support advice testing are assumed to be simple and strict enough that the Just-In-Time compiler in most Java Virtual Machine implementations will be able to inline the method call, and thus reduce any overhead to insignificance. The AspectJ FAQ reinforces that perception, claiming that most constructions have little overhead, which "could be optimised away by modern VM's." [25] (section 7.3).

There are a few studies that actually measured the performance impact of using aspects. Pace and Campo, for instance, analyzed regular and aspect-oriented versions of a temperature control benchmark [4]. Although they found one style of implementation to be over 3 times slower then the original, a different aspect-oriented approach had only about 1% runtime overhead. They attribute the former to the internal use of reflection, and conclude that the impact may depend on the problem under consideration. A more recent and larger study was done by Hilsdale and Hugunin [9], examining both runtime and compile-time performance issues. A naive implementation is shown to have quite poor performance (for a logging implementation they get a 2900% overhead versus a hand-coded implementation), but they improve that to an "unlikely to be noticeable" 22% runtime overhead for an optimised version. Again they attribute the former very poor performance largely to the use of reflection.

In the context of middleware, Zhang and Jacobsen [27] demonstrate that an aspect version of a CORBA/ORB benchmark has negligible runtime overhead. They argue that an AspectJ implementation should have no overhead since it is just specifying the same code in different ways (in the aspect versus in the program). In their case, however, an aspect-oriented approach significantly simplified the program design (overall code reduction of 9%, fewer methods per class on average, etc), so they are actually comparing an optimised design to an unoptimised design. The fact that the optimised design only achieves the same speed as the unoptimised is an argument that a significant overhead may well be present.

In their analysis, Zhang and Jacobsen also give data for a number of software engineering complexity metrics, and use that data to show that the aspect-oriented approach is quantitatively simpler. Complexity is also considered by Zhao, who proposes a specific complexity metric suite for aspect oriented programming [28]. We are focussing on performance and execution time costs, rather then complexity.

Clearly particularities of the implementation of aspects have a large impact on the overhead. Sereni and de Moor describe a better implementation of pointcut designators as well as a compiler flow analysis that can reduce the overhead by eliminating many instances of runtime matching [21]. Our experience with the overhead of aspects suggests that such optimisation techniques may be quite important in practice.

Performance analyses have also been done on dynamic weaving approaches where an aspect is applied to a running program. Dynamic weaving generally aims to enhance capabilities, allowing for instant "hot fixes" to be applied to running code [17, 18]. Popovici et al show an aspect-aware Java Virtual Machine that is only slightly slower (without applying any aspects) than a regular JVM, and a little slower then the equivalent statically-specified aspect application.

6 Conclusions

We have presented a tool set and a systematic method for analysing the dynamic behaviour of AspectJ programs. Our aim in doing so was to provide guidance for AspectJ developers, compiler writers and tool builders. Below we summarise our main conclusions for each of these constituencies, and then draw some more general conclusions.

Developers. AspectJ provides some powerful new features, and it is easy to introduce performance bugs by writing pointcuts that are too liberal. This was in particular illustrated by the null check benchmark, where the loose description of the initial pointcut implied **around** advice that could apply to the **around** body, thus necessitating the introduction of closures. This is a situation that developers in AspectJ should look out for, as it appears to be fairly common.

Compiler writers. The quicksort benchmark illustrated a number of optimisations that could be made to the implementation of **cflow**. In particular the important special case of stacks that consist of 0-length arrays of objects is a change that would not be difficult to implement.

Naturally it would be of interest to see how much of the progression of different versions of the null check benchmarks could be achieved by a compiler. It would appear that a more careful analysis of the necessity of introducing closures could lead to significant performance improvement in many examples that employ **around** advice.

The bean benchmark illustrates that the static features of AspectJ can also lead to some small overheads, but these are negligible compared to the costs associated with pointcut matching and advice execution.

Tool builders. Given the huge performance implications of the design of an aspect, and the pitfalls this presents to all but the most experienced users of AspectJ, it is desirable to have tools that make it easy to visualise the impact of a new pointcut. There already exists a plugin for Eclipse that indicates join point shadows where a pointcut might apply; it would be valuable to combine this with the results of dynamic metrics.

In particular, it would be interesting to find hot join point shadows, and alert the developer that further optimisation is required at this point. We have started to extend our tool set to collect this information, but it is too early to report on the results.

It would also be valuable to be able to view the results of the tagging in the output of the Dava decompiler — this would provide a quick way of explaining performance bottlenecks once they have been identified.

General. For all programmers with an interest in aspect-orientation, it is important to understand the implications of using aspects on the behaviour of their programs. The tools we have presented are an important step towards this goal, but perhaps even more important is the construction of a representative set of benchmarks that is accepted by the whole community. We hope that the benchmarks presented here provide a starting point, and that others will join us in extending and improving it.

Acknowledgements

This work was supported, in part, by NSERC (Canada), FQRNT (Quebec), EPSRC (UK), and IBM. Our thanks to Ondřej Lhoták, Ian Lynagh and Jennifer Lhoták for their comments on a draft of this paper.

References

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

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

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

[4] J. Andrés Díaz Pace and Marcelo R. Campo. Analyzing the role of aspects in software design. *Commun. ACM*, 44(10):66–73, 2001.

[5] Sylvia Dieckmann and Urs Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *Proceedings of ECOOP 1999, LNCS 1628*, pages 92–115, 1999.

[6] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for Java. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 149–168. ACM Press, 2003.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[8] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th Annual Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 161–173, 2002.

[9] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. Available from URL: `http://www.cs.indiana.edu/~ehilsdal/cv/index.html`, 2003.

[10] G. Kiczales, J. Lamping, A. Menhdekar, C. Maeda, C. Lopes, J.M. Loingties, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *European Conference on Object-oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[11] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *European Conference on Object-oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

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

[13] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Compilation semantics of aspect-oriented programs. In *Foundations of Aspect-Oriented Languages (FOAL), Workshop at AOSD 2002*, Technical Report TR #02-06, pages 17–26. Iowa State University, 2002.

[14] Jean Mayrand, Jean-Franois Patenaude, Ettore Merlo, Michel Dagenais, and Bruno Laguë. Software assessment using metrics: A comparison across large C++ and Java systems. *Ann. Softw. Eng.*, 9(1-4):117–141, 2000.

[15] Jerome Miecznikowski and Laurie Hendren. Decompiling Java using staged encapsulation. In *The Working Conference on Reverse Engineering (WCRE'01)*, pages 368–374, October 2001.

[16] Jerome Miecznikowski and Laurie Hendren. Decompiling Java bytecode: Problems, traps and pitfalls. In *Compiler Construction, 11th International Conference*, volume 2304 of *LNCS*, pages 111–127, April 2002.

[17] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-in-time aspects: efficient dynamic weaving for java. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 100–109. ACM Press, 2003.

[18] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147. ACM Press, 2002.

[19] McGill University Sable Research Group. Soot: a Java optimization framework, 1998-2003.

[20] Damien Sereni. A definitional interpreter for aspects. Available from URL: `http://www.comlab.ox.ac.uk/oucl/research/areas/progtools/aspects`, 2002.

[21] Damien Sereni and Oege de Moor. Static analysis of aspects. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 30–39, 2002.

[22] Yefim Shuf, Mauricio J. Serrano, Manish Gupta, and Jaswinder Pal Singh. Characterizing the memory behavior of Java workloads: a structured view and opportunities for optimizations. In *Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 194–205. ACM Press, 2001.

[23] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. *ACM SIGPLAN Notices*, 35(10):264–280, 2000.

[24] Michalis Xenos, D. Stavrinoudis, K. Zikouli, and D. Christodoulakis. Object-oriented metrics - a survey. In *Proceedings of the FESMA 2000, Federation of European Software Measurement Associations*, 2000.

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

[26] Sherif M. Yacoub, Hany H. Ammar, and Tom Robinson. Dynamic metrics for object oriented designs. In *Proceedings of the 6th International Symposium on Software Metrics*, page 50. IEEE Computer Society, 1999.

[27] Charles Zhang and Hans-Arno. Jacobsen. Quantifying aspects in middleware platforms. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 130–139. ACM Press, 2003.

[28] Jianjun Zhao. Towards a metrics suite for aspect-oriented software. Technical Report SE-136-25, Information Processing Society of Japan (IPSJ), March 2002. `http://citeseer.nj.nec.com/zhao02towards.html`.

Appendix I: Tags

NO_TAG (-1): This is a special tag inserted by the compiler which is meant to be overwritten by a propagated tag during analysis. An instruction with this tag is to be interpreted equivalently to an instruction with no tag at all. It is a necessary consequence of the way tags are encoded in a code attribute.

DEFAULT (0): This tag represents instructions that are not interpreted as AspectJ overhead or part of an advice body. They represent the base program that exists before weaving.

ADVICE_EXECUTE (1): This tag represents the overhead associated with executing the method implementing a piece of advice. Advice bodies are compiled as methods in the aspect class. When an aspect with advice is woven into a base class, an invoke instruction for the advice method is added to the relevant join point shadows.

ADVICE_ARG_SETUP (2): This tag represents the overhead associated with acquiring an aspect instance at a join point at which advice is to be executed, and exposing arguments to the advice body. At least one instruction of this kind will precede an advice execution instruction.

ADVICE_TEST (3): 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. The instructions corresponding to this test have this tag.

AFTER_THROWING_HANDLER (4): In order to support after and after throwing advice, exception handling code is inserted which catches any exception, executes any pertinent advice, and then rethrows the original exception. The instructions responsible for this have this tag.

EXCEPTION_SOFTENER (5): This tag represents the overhead involved in softening exceptions. The ‘declare soft’ declaration in an aspect results in exceptions of a given type, thrown from within join points selected by a given pointcut, being wrapped in the unchecked org.aspectj.SoftException, which is then thrown.

AFTER_RETURNING_EXPOSURE (6): This tag represents the overhead involved in exposing the value returned at a join point to the body of a piece of **after** advice.

PEROBJECT_ENTRY (7): By default, aspect instances are singletons. They can however be associated on a per-object basis, either with the execution or target objects at join points selected by a given pointcut. The instructions inserted at join point shadows matched by the pointcut to manage these instances have this tag. (See also PEROBJECT_GET (21) and PEROBJECT_SET (22).)

CFLOW_EXIT (8), CFLOW_ENTRY (9): The **cflow** and **cflowbelow** pointcuts require that a representation of the call stack be managed during the execution of the program. At every relevant join point shadow, this representation must be updated. Instructions for doing so receive one of these tags.

PRIV_METHOD (10), PRIV_FIELD_GET (11), PRIV_FIELD_SET (12): In order to support privileged aspects, public wrapper methods for the class’s private methods, and public accessor methods for the class’s private fields, are inserted during weaving. The instructions in these new methods have these tags.

CLINIT (13): The instructions in the static initializer of the aspect class have this tag. The static initializer may setup the default singleton instance of the aspect or setup the cflow stack, if necessary. Instructions woven into the static initializer of a base program class, such as for initializing the static join point information, also have this tag.

INTERMETHOD (14): An inter-type method declaration results in the body of the new method being compiled into a method on the aspect class, and dispatch method being added to the the target class. the instructions in this dispatch method have this tag.

INTERFIELDGET (15), INTERFIELDSET (16): Some inter-type field declarations result in accessor methods being woven into the target class. The instructions in these accessor methods have these tags.

INTERFIELD_INIT (17)**:** Inter-type field declarations result in intialization code being woven into either the target class's constructor, or its static initializer. These instructions invoke initialization methods on the aspect to handle variable initialization. This initialization code has this tag.

INTERCONSTRUCTOR_PRE (18)**,** INTERCONSTRUCTOR_POST (19)**:** When an aspect has an inter-type constructor declaration, two methods are created on the aspect: a *preInterConstructor* method and a *postInterConstructor* method. A new constructor method is added to the class, and it invokes both of these methods. The instructions that load these methods' arguments and invoke these methods have these tags.

INTERCONSTRUCTOR_CONVERSION (20)**:** This represents the overhead involved in calling methods on org.aspectj.-runtime.internal.Conversions from within a constructor added by an inter-type constructor declaration.

PEROBJECT_GET (21)**,** PERBOJECT_SET (22)**:** These accessor methods are added to a class to acquire instances of an aspect that is declared **pertarget** or **perthis**. (See also PEROBJECT_ENTRY (7).)

AROUND_CONVERSION (23)**:** This represents the conversion of arguments to and return values from a *proceed*() call within **around** advice. This conversion is done by making calls to methods on org.aspectj.runtime.internal.Conversions, which convert between primitive types and objects.

AROUND_CALLBACK (24)**,** AROUND_PROCEED (25)**:** Both of these tags represent an overhead involved in making a *proceed*() call from within **around** advice. One of these tags, AROUND_CALLBACK, is specific to the run method on closure classes.

CLOSURE_INIT (26)**:** **Around** advice may result in the creation of closure classes. When it does, the instructions in the constructors of these classes have this tag.

INLINE_ACCESS_METHOD (27)**:** This tag represents the overhead involved in calling a method defined on an aspect when there is a static dispatch method. The instructions of the static dispatch method have this tag.

ASPECT_CODE (28)**:** This tag represents the default for any instruction that is executed from an aspect, regardless of where it was originally defined. It is propagated, so that, for example, the body of a method call from advice will receive tag 28.