McGill University
School of Computer Science
Sable Research Group

Oxford University
Computing Laboratory
Programming Tools Group

Measuring the Dynamic Behaviour of AspectJ Programs

(Revised - replaces Sable Technical Report 2003-8)

Sable Technical Report No. 2004-2

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

March 22, 2004

Contents

1 Introduction **3**

2 A brief introduction to AspectJ **4**

2.1 Join points, pointcut and advice . . . 5

2.2 Intertype declarations . . . 6

3 Measurements and Dynamic Metrics **6**

3.1 Execution Time . . . 6

3.2 Java-based dynamic metrics . . . 6

3.2.1 Dead Code and Code Coverage . . . 7

3.3 AspectJ-specific dynamic metrics . . . 7

3.3.1 Tag Mix . . . 7

3.3.2 Aspect Overhead . . . 9

3.3.3 AspectJ Runtime . . . 9

3.3.4 Advice Execution . . . 9

3.3.5 Hot Shadows . . . 9

4 Tools for collecting Dynamic Metrics **10**

4.1 An example . . . 10

4.2 Static Tagging: annotating class files using a modified AspectJ compiler . . . 12

4.2.1 Tagging during weaving . . . 12

4.2.2 Pretagging . . . 13

4.2.3 Generating attributed class files . . . 13

4.3 Dynamic metric analysis with tag propagation using *J . . . 13

4.3.1 Modifications to the *J analyzer: . . . 13

4.3.2 Collecting the AspectJ-specific metrics . . . 14

5 Benchmarks **15**

5.1 Overall Data . . . 15

5.2 Benchmarks with low runtime overhead . . . 17

5.2.1 DCM . . . 17

5.2.2 ProdLine . . . 17

5.2.3 Tetris . . . 18

5.2.4 Bean . . . 18

5.3 Benchmarks with high overheads . . . 19

5.3.1 NullCheck . . . 19

5.3.2 Figure . . . 23

5.3.3 LoD . . . 25

5.4 Benchmark for performance improvement . . . 26

5.4.1 *J Pool . . . 28

6 Related Work **28**

7 Conclusions **30**

List of Figures

1 Example AspectJ program 5
2 Overview of Metric Collection Tools 10
3 Tagged class file for example AspectJ program 11
4 Dynamic Propagation Table 14
5 Benchmark Measurements 16
6 Nullcheck metrics 21
7 Figure Benchmark Measurements 24
8 Law of Demeter Benchmark Measurements 27
9 *J Pool Benchmark Measurements 29

Abstract

This paper proposes and implements a rigorous method 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 analyzer which propagates tags and computes the proposed metrics. This dynamic propagation is essential, and thus this paper contributes not only new metrics, but also non-trivial ways of computing them.

We furthermore present a set of benchmarks that exercise a wide range of AspectJ's features, 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 optimization, and to tool builders on ways to understand runtime behaviour of AspectJ.

1 Introduction

Aspect-oriented programming [17] is a new technique for modularizing a program. An *aspect* is a feature that "cross-cuts" the traditional abstraction boundaries of classes and methods; the most common examples of aspects are ones used for tracing or logging the execution of an existing program, but aspect-oriented design techniques have also been used successfully for more closely coupled functionality improvements, such as connection pooling.

The most popular implementation of these ideas is AspectJ [16], an extension of Java. The textbook by Laddad [19] provides a nice introduction, both to the language and its potential applications. AspectJ started out as a pioneering research effort, but has quickly reached a level of maturity where it is on the verge of being used for production programming, and we therefore believe that the time is right for the research community to pay more attention to the performance of AspectJ programs.

The conceptual model behind AspectJ execution is one in which aspects dynamically "observe" the execution of a base Java program. At certain points during this execution, known as *join points* and specified (in aspects) by *pointcuts*, an aspect inserts or substitutes its own code, known as *advice*. Of course, this conceptual model would be extremely expensive to implement literally; instead, AspectJ is implemented as a compiler which statically *weaves* advice code into the base code. In many cases, whether or not advice would apply at runtime (in the conceptual model) is statically determinable, and so this can be done without introducing runtime overhead. However, it is not always possible to decide this at compile time, and so a runtime test has to be inserted, particularly when the more complex features of pointcuts are being used. However, it is a stated goal of the AspectJ compiler to minimize these overheads; indeed, the AspectJ FAQ [35] states:

> "*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.*"

It appears to be generally believed in the AspectJ community that the compiler does not introduce overheads, and indeed we have confirmed that in many situations it is the case that equivalent Java and AspectJ programs have essentially the same performance. However, we have also identified a number of examples in which the AspectJ compiler does impose a significant overhead, contradicting this belief.

The FAQ goes on to say:

> "*There is currently no benchmark suite for AOP languages in general or for AspectJ in particular. It is probably too early to develop such a suite because AspectJ needs more maturation of the language and the coding styles first. Coding styles really drive the development of the benchmark suites since they suggest what is important to measure.*"

We contend that the development of a benchmark set which shows good as well as bad uses of AspectJ language features will help to inform the development both of the AspectJ language and compiler, and of coding styles; and that it is better to view the situation as a two-way process where benchmarking both drives and is driven by such development. The overheads that we have found using our benchmark set confirm this.

In detail, the contributions of this paper are as follows:

- We provide a new set of dynamic metrics and tools for measuring the performance of AspectJ programs and attributing elements of this performance between the original Java code, the introduced aspect code and the compilation overhead of individual AspectJ language features.
- We have collected the first benchmark set of AspectJ programs, from a variety of public sources. Despite the growing popularity of AspectJ, it has proved rather difficult to find publicly available programs. We hope that it will form the basis of a generally accepted suite of benchmarks and we welcome further contributions from the AspectJ community.
- We explain the "conventional wisdom" that the AspectJ compiler introduces no runtime overhead, by showing a series of benchmarks in which this overhead is indeed negligible.
- We show that in other benchmarks, there is a significant overhead. We identify the language features and patterns of usage that lead to this overhead.
- Using the Dava decompiler [25] from the Soot toolkit [32], we investigate the output of the AspectJ compiler where our tools pinpointed a significant performance impact, and demonstrate various ways in which improvements could be made. This measure-identify-decompile-fix cycle is very economic in the AspectJ situation, where a new language paradigm calls for novel analyses and optimizations: it would be immensely labour-intensive to obtain the same results through direct experiments with different versions of the compiler.

These contributions will be of benefit to three groups of people:

- *AspectJ users:* Our results provide guidance on which AspectJ idioms are cheap to use and which impose a performance penalty. For example, we found that directions for advice placement (before/after/around) can have a significant impact on performance, and our experiments explain why.
- *AspectJ compiler implementors:* We identify areas in which compilers could be improved, for example by using more sophisticated static analyses to eliminate runtime checks for pointcut matching. Some of these suggestions are very easy to implement, and indeed we report one such optimization which we applied in a modified version of the compiler.
- *AspectJ tool developers:* The power of AspectJ makes it very easy to write a seemingly innocuous piece of advice that turns out to have dramatic consequences for performance. Our results point the way towards interactive tools that warn the programmer of such situations, and help to remedy the problem when it arises.

The remainder of this paper is structured as follows. In Section 2 we provide a brief overview of the AspectJ language, and in Section 3 we provide an overview of the statistics we collect for our set of benchmarks.

In Section 4 we give the full details of our toolset, which consists of a modified version of the AspectJ compiler [3] that "tags" bytecode instructions according to their provenance (the base Java program, aspect code, or compiler overhead from particular language features), along with a modified version of the *J metric tool [8] which collects statistics for each of these tags. The tagging is performed both statically and dynamically to allow some tags to be context-dependent; this is vital since in some cases code that is compiler overhead may make calls that should also be attributed to this overhead, but in other cases it may call aspect code that should also be attributed correctly. Developing this tag "propagation" scheme has been a major part of our work.

The benchmarks themselves are presented in Section 5. We split them into two categories: those that do not demonstrate significant compiler overhead and those that do. In the case of the latter category, we investigate the reasons for this overhead in detail and suggest possible improvements.

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 6. Finally we discuss our conclusions in Section 7.

2 A brief introduction to AspectJ

AspectJ is an extension of Java; it provides novel features for modularization, in particular when adding new functionality to an existing 'base program'. The novel features can be classified into two groups. The first group allows

one to influence the dynamic behaviour of the base program by injecting new code when certain events occur in its execution. We discuss these dynamic features in Section 2.1. The second group of features allows one to statically add new members to classes. These features are reviewed in section 2.2. This introductory section only covers the very basics, and readers who are new to AspectJ may wish to consult one of the textbooks [10, 18, 19] for a more comprehensive introduction.

2.1 Join points, pointcut and advice

When adding tracing functionality to an existing program, it is often undesirable to modify the program itself: the implementation of tracing is scattered over the design, and hence it obscures the existing code, and it is difficult to maintain itself. It would be preferable if we could describe the execution events that we wish to trace, and the action to take upon each such event. AspectJ allows us to do this by specifying such execution events. The events are called *join points*, the pattern that specifies a set of join points is named a *pointcut* and the additional code that gets run is called *advice*. The join points that can be be selected via pointcuts can be thought of as nodes in the dynamic call tree of the program. Besides nodes for method calls, this call tree also includes nodes for the execution of a method body, exception handlers, and so on.

```
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 1: Example AspectJ program

To illustrate these abstract definitions, let us examine a tiny example, shown in Figure 1. It consists of a base program (the class *Example*) and an aspect (named *ExampleAspect*). The base program consists of two methods called *foo* and *bar*. The purpose of the aspect is to signal any calls that are made to *bar* within the dynamic scope of *foo*. In terms of the call tree, this means that we are interested in *bar* nodes that occur below a call to *foo*. In the aspect, this is expressed as follows. It says that **before** entering any join point selected by the pointcut

```
call(void Example.bar()) && cflow(call(void Example.foo()))
```

the message "`foo -> bar`" should be displayed on the standard output. The pointcut itself consists of two parts: it says we want a **call** to *bar*, and furthermore we must be dynamically within the control flow (**cflow**) of *foo*.

While the matching of join points to pointcuts is conceptually a dynamic process that happens entirely at runtime, the AspectJ compiler shifts a lot of the work to compile-time. In the above example, it will identify the calls to *bar* in the program text, effectively matching the first part of the pointcut. The second part of the pointcut (involving **cflow**) is however matched dynamically, and to this end some extra code is inserted, which checks whether we are in the

dynamic scope of *foo*. To make that check, it is in turn necessary to do a little administration at each call to *foo*. As we shall discuss in more detail later, the AspectJ compiler mimics the call stack by recording each entry to *foo*, and each exit.

There is some terminology to ease discussion of these issues. The place in a program text that gives rise to a particular join point at runtime is called a *shadow*. As we have just explained, the compiler matches pointcuts against such shadows, possibly leaving a *dynamic residue* for the tests that could not be resolved completely. The process of producing combined code for the base program and its aspects is called *weaving*.

Our own understanding of join points and advice was mostly shaped by [33], which gives a definitional interpreter for join points and advice. Our discussion of the weaving process has been greatly influenced by [23], which explains it in terms of partial evaluation of the interpreter in [33]. The definitive account of the way the AspectJ compiler works can be found in [14].

We shall introduce further features relating to join points and advice as we discuss specific benchmarks later on in this paper. In particular, we shall examine different placements of advice (**after** and **around**) in addition to **before**.

Finally, we should remark that the example in this section does not require the use of **cflow**. AspectJ has another kind of pointcut, namely **withincode** that would be preferable to use for such a simple application, because it is more efficient. One aim of the present paper is to elucidate such issues.

2.2 Intertype declarations

While advice is a powerful mechanism to modularize designs where the traditional abstractions of Java fail, it is not always enough on its own. Sometimes it is necessary to make a static change to an existing class, for example to add a new method. AspectJ allows such *intertype declarations*. For example, the aspect in Figure 1 could enhance the *Example* class with a new method called *goo* by including the line

```
public void Example.goo() { System.out.println("goo"); foo(); }
```

Client code of *Example* (introduced by the aspect) can now refer to *goo* in the same way as it references *foo* or *bar*.

Similar ideas can be found in other extensions to Java, in particular MultiJava [5] and RMJ [26]. These designs are in fact more disciplined than AspectJ, and they allow for modular type checking, which AspectJ does not; furthermore they include multimethods, a feature that AspectJ lacks at present.

3 Measurements and Dynamic Metrics

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

3.1 Execution Time

The most coarse-grained measurement is the execution time of a program, which we use 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 1.4.2_02-b03).

3.2 Java-based dynamic metrics

As well as execution time, 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 [8], 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 5.

It is often useful to differentiate between application code and library code, especially in the case of AspectJ programs. We define application code as the set of class files that are directly generated by the AspectJ compiler. In addition to generating "WHOLE PROGRAM" versions of the metrics, *J is also able to generate "APPLICATION ONLY" versions of them by only taking into consideration the contributions made by the application code.

3.2.1 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. Since the entire class must be loaded, this causes unnecessary time to be spent in class loading and verification.

Thus, we found that it was useful to add two new metrics to our standard Java metrics. 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 given run, but is a necessary consequence of support for incremental compilation and weaving, since a change to the base program might cause the code to become required and we would not want to have to recompile the aspect in that case.

3.3 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 identify 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 non-trivial ways to compute them. These extensions are described in more detail in Section 4, but mainly consist of associating a tag to every executed bytecode instruction indicating its purpose. In the following subsections we describe the new metrics that were designed specifically for analyzing AspectJ programs.

3.3.1 Tag Mix

The *tag mix* metric partitions all executed bytecode instructions into 29 different bins, where each bin corresponds to a specific purpose. Bins are reported as a percentage of total executed instructions. This breakdown of executed bytecodes is useful in determining which particular features of AspectJ are used in a given benchmark.

Individual tags can be grouped into categories according to the AspectJ language feature that they relate to. We define 10 categories of tags, 9 of which correspond to overhead code introduced by the AspectJ compiler. A detailed list of tags and categories is given in Appendix I, and example measurements of these tags are given in Section 5. A short description of all categories, along with their most important tags, is presented next.

Readers who are unfamiliar with AspectJ may wish to skim this section first time through, and then return to it after seeing some example programs in Section 5.

General tags This category contains tags which are associated with user-defined code. For analysis purposes, we distinguish between regular code and advice code. The BASE_CODE bin represents all executed instructions that correspond to the base program (regular Java), whereas the ASPECT_CODE bin corresponds to code that was executed as part of the aspect. This includes all non-overhead instructions corresponding to the body of an advice and all

non-overhead instructions in code called from the body. It also includes all non-overhead instructions in methods introduced by intertype declarations.

In making the distinction between base program and aspect, we err on the side of underestimating the effect of aspects, for example by making all instructions due to callbacks from native methods contribute to the BASE_CODE bin.

Advice-related tags This category contains tags that are common to **before**, **after** and **around** advice. The ADVICE_EXECUTE tag identifies overhead associated with executing an advice body. The ADVICE_ARG_SETUP tag identifies overhead associated with exposing parameters to the advice body. The ADVICE_TEST tag is associated with dynamic guards inserted by the compiler in cases where it could not determine whether a particular advice body should always be executed for a given join point.

Tags specific to around advice Unlike **before** and **after**, **around** advice replaces existing code with the advice body. The original code can still be invoked through the special **proceed**() statement, though implementation of this feature implies additional overhead. The AROUND_PROCEED tag identifies instructions which are inserted to make a call to **proceed**() from within an advice body. Under some circumstances, it is possible that the call to **proceed**() is not implemented using the inlining strategy, but implemented using a more general technique, a *closure*. We therefore define the tag AROUND_CALLBACK which serves the same purpose as AROUND_PROCEED, but which additionally identifies the tagged instructions as part of the closure implementation. The CLOSURE_INIT tag is used to identify instructions which initialize the closure objects that are created.

Tags specific to after advice There are two distinct kinds of overhead that are associated with the use of **after** advice. As with **around** advice, exposing the return value of a method to the advice body requires support from the compiler, leading to the addition of some overhead code. Also, because **after** advice must execute regardless of whether the method terminated normally or not, the compiler adds exception handlers to the original code in order to address this issue. The AFTER_RETURNING_EXPOSURE and AFTER_THROWING_HANDLER tags are associated with these two situations, respectively.

Intertype declaration tags Intertype declarations in AspectJ can lead to several forms of overhead being introduced: additional method invocations, accessor methods for introduced fields, variable initialization, etc. Several tags are defined to identify each kind of overhead.

perthis and pertarget-specific tags Normally aspects are singletons; however, they can also be defined on a per-object basis. This category contains instructions which are used to manipulate aspect instances when there are multiple instantiations rather than a single one.

Cflow-specific tags Because **cflow** pointcuts and **percflow** aspects (as with **perthis** and **pertarget**, **percflow** is defined on a per-object basis) require some knowledge of the dynamic control flow of the application, the compiler inserts overhead code in order to create and maintain a representation of this information. There are two tags, CFLOW_ENTRY and CFLOW_EXIT, to identify instructions which keep this data structure updated.

Exception softening tags This category contains a single tag, EXCEPTION_SOFTENER, which identifies instructions which are used to wrap instances of checked exceptions into an unchecked *org.aspectj.SoftException* instance.

Tags specific to privileged aspects Privileged aspects have access to private methods and fields of classes. The compiler makes it possible by adding public wrappers to the appropriate classes. This category contains tags that identify instructions which are part of the inserted wrapper methods.

Miscellaneous aspect tags This category contains two kinds of tags. The first kind of tag, CLINIT, is associated with instructions which are found in static initializers of aspect classes. The second kind, INLINE_ACCESS_METHOD, identifies the overhead involved in calling a method defined on an aspect when there is a static dispatch method.

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.

3.3.2 Aspect Overhead

Once every executed bytecode has been tagged appropriately, it is possible to compute the percentage of executed instructions which fall into the "overhead" category. We define overhead as all instructions executions which do not fall within the "general" tag category (BASE_CODE or ASPECT_CODE). This closely corresponds to the instruction executions that would not be found in a hand-woven implementation of the same application.

The aspect overhead metric can also be expressed as the product of two other ratios. The "overhead to advice" ratio indicates the relative amount of overhead per introduced advice. It is measured as the number of executed overhead bytecode instructions divided by the number of executed advice instructions. The "advice to total ratio" measures the proportion of the executed code that belongs to advice bodies, and is computed as the number of executed advice instructions divided by the total number of executed instructions.

3.3.3 AspectJ Runtime

In order to truly measure the proportion of the code that can be attributed to the use of AspectJ, it is necessary to keep track of the calling context. The "AspectJ runtime library" metric measures the percentage of the code that is executed as part of the AspectJ library, or on its behalf.

3.3.4 Advice Execution

In many cases, the AspectJ compiler can statically determine 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 for which 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 statically determinable 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, categorizing 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 (in one particular run) are potential places where a stronger static analysis might be able to eliminate the check, thus eliminating unnecessary overhead and improving performance. Of course it may be the case that some checks that are measured as always going one way actually could go the other way in a different run of the program, so it is not necessarily the case that all of those which are identified could really be removed.

3.3.5 Hot Shadows

Recall that a *shadow* is the static location in a program text that gives rise to a particular join point at runtime. The hot shadows metric measures the percentage of all shadows that account for 90% of the total advice body invocations. This gives an indication of whether runtime advice execution is mostly concentrated on a few shadows, or whether it is thinly spread; this metric thus helps us to understand whether we might obtain a performance gain by concentrating on just a few locations (and for example inlining advice bodies at those locations). Note that if there are overlapping pointcuts, it is possible for one shadow to invoke multiple advice bodies.

4 Tools for collecting Dynamic Metrics

An overview of the tools that we use for collecting the dynamic metrics is given in Figure 2. 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 analyzer, which propagates the bytecode tags across method calls, according to the context of the call, and computes the dynamic metrics.

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.

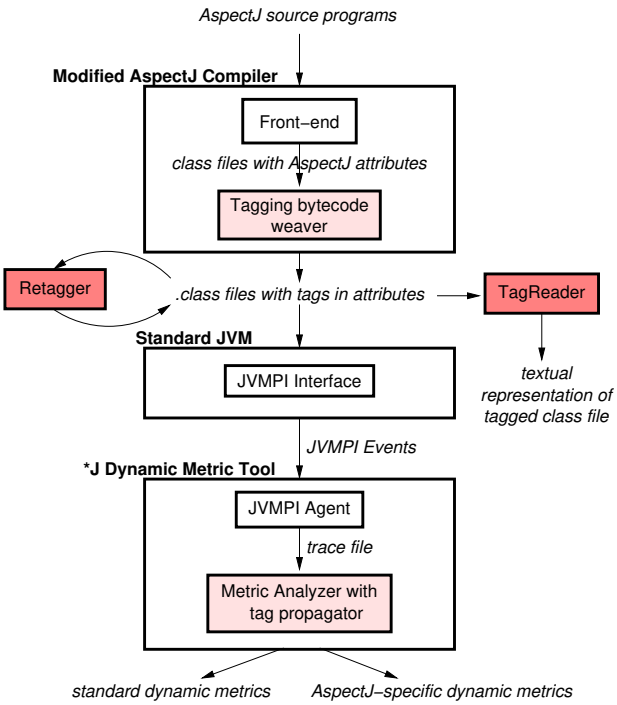

Figure 2: Overview of Metric Collection Tools

In the following subsections we first provide an illustrative example, showing examples of static tagging and tag propagation (Section 4.1). We then provide more specific details on the implementation of the two main components of our approach, the static tagger, based on the AspectJ 1.1.1 compiler (Section 4.2), and the dynamic metric analyzer, based on *J (Section 4.3) .

4.1 An example

To demonstrate our approach to static tagging and dynamic propagation, consider the small AspectJ program in Figure 1. 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).

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 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.

Tag	Shadow	
		public Example()
		0: `aload_0`
		1: `invokespecial` **Object()**
		4: `return`
		public static void main(String[] args)
		0: `new` **Example**
		3: `dup`
		4: `invokespecial` **Example()**
		7: `astore_1`
		8: `aload_1`
ADVICE_TEST	**12**	**9:** `getstatic` **CFlowStack ExampleAspect.ajc$cflowStack$0**
ADVICE_TEST	**12**	**12:** `invokevirtual` **boolean CFlowStack.isValid()**
ADVICE_TEST	**12**	**15:** `ifeq` → 24
ADVICE_ARG_SETUP	**12**	**18:** `invokestatic` **ExampleAspect ExampleAspect.aspectOf()**
ADVICE_EXECUTE	**12**	**21:** `invokevirtual` **void ExampleAspect.ajc$before$ExampleAspect$148()**
		24: `invokevirtual` **void Example.bar()**
		27: `aload_1`
CFLOW_ENTER	**13**	**28:** `bipush` **0**
CFLOW_ENTER	**13**	**30:** `anewarray` **Object[]**
CFLOW_ENTER	**13**	**33:** `astore_3`
CFLOW_ENTER	**13**	**34:** `getstatic` **CFlowStack ExampleAspect.ajc$cflowStack$0**
CFLOW_ENTER	**13**	**37:** `aload_3`
CFLOW_ENTER	**13**	**38:** `invokevirtual` **void CFlowStack.push(Object[])**
		41: `invokevirtual` **void Example.foo()**
CFLOW_EXIT	**13**	**44:** `goto` → 24
CFLOW_EXIT	**13**	**47:** `astore` **4**
CFLOW_EXIT	**13**	**49:** `getstatic` **CFlowStack ExampleAspect.ajc$cflowStack$0**
CFLOW_EXIT	**13**	**52:** `invokevirtual` **void CFlowStack.pop()**
CFLOW_EXIT	**13**	**55:** `aload` **4**
CFLOW_EXIT	**13**	**57:** `athrow**
CFLOW_EXIT	**13**	**58:** `nop`
CFLOW_EXIT	**13**	**59:** `getstatic` **CFlowStack ExampleAspect.ajc$cflowStack$0**
CFLOW_EXIT	**13**	**62:** `invokevirtual` **void CFlowStack.pop()**
		65: `nop`
		66: `return`
		public void foo()
		0: `getstatic` **PrintStream System.out**
		3: `ldc` **"foo"**
		5: `invokevirtual` **void PrintStream.println(String)**
		8: `aload_0`
ADVICE_TEST	**17**	**9:** `getstatic` **CFlowStack ExampleAspect.ajc$cflowStack$0**
ADVICE_TEST	**17**	**12:** `invokevirtual` **boolean CFlowStack.isValid()**
ADVICE_TEST	**17**	**15:** `ifeq` → 24
ADVICE_ARG_SETUP	**17**	**18:** `invokestatic` **ExampleAspect ExampleAspect.aspectOf()**
ADVICE_EXECUTE	**17**	**21:** `invokevirtual` **void ExampleAspect.ajc$before$ExampleAspect$148()**
		24: `invokevirtual` **void Example.bar()**
		27: `return`
		public void bar()
		0: `getstatic` **PrintStream System.out**
		3: `ldc` **"bar"**
		5: `invokevirtual` **void PrintStream.println(String)**
		8: `return`

Figure 3: Tagged class file for example AspectJ program

Instructions 9–15 in both *main*(*String*[]) and *foo*() are tagged ADVICE_TEST; 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 ADVICE_ARG_SETUP and ADVICE_EXECUTE. 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), 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 analyzer. Instead, the ASPECT_CODE tag is propagated to the advice body method.

Instructions 28–38 (CFLOW_ENTER) and 44–62 (CFLOW_EXIT) manage the representation of the call stack, required by the **cflow** pointcut. This call stack representation is described in more detail in section 5.3.2. 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 and CFLOW_EXIT, propagate to the called methods since the *push* and *pop* methods represent the same kinds of overhead.

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

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, which produces class files with special attributes. These attributes contain information for the second stage, where aspects are woven into the bytecode of a base program.

We have modified the bytecode weaver of version 1.1.1 of the AspectJ compiler to annotate the classes it produces. A first set of annotations assigns *tags* to certain bytecode instructions. These tags aim at identifying the role of the instruction in the generated code, such as dynamically guarding a given piece of advice, invoking an advice body, etc. The tag annotations are focused on studying the use of the different language features that AspectJ supports; 27 out of the 29 possible tags represent overhead instructions (the other two are for base and aspect code respectively).

A second set of annotations identify the join point shadows into which instructions have been inserted during weaving. Each added instruction is tagged with a shadow ID corresponding to a single join point shadow. For example, the single advice declaration listed in Figure 1 results in instructions being added to multiple join point shadows in the base program. These added instructions have shadow ID tags as shown in Figure 3. The three join point shadows, each corresponding to a method call, have IDs 12, 13, and 17. The weaver additionally stores a table mapping each shadow ID to its shadow kind (*e.g.* method-call) and its signature (*e.g.* void Example.bar() for shadow 12 in the example.)

4.2.1 Tagging during weaving

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 intertype declarations. The second is the *shadow munger*, which 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 1. 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 tag. Then the advice execution instruction is created, which is an invoke to the advice body method. We tag this ADVICE_EXECUTE 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. This newly generated instruction list is then inserted into the shadow, which is a range of instructions in 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.

4.2.2 Pretagging

Not all of the instructions we wish to annotate during the weaving stage are generated during the weaving stage. Existing instructions in aspect classes, generated during the front-end AspectJ compilation, may also represent overhead. 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, however, since AspectJ 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 this stage, we search for existing overhead instructions within aspect classes and tag them. Since the AspectJ compiler automatically generates names for advice bodies and other methods on the aspect class, this is accomplished by searching for bytecode patterns in methods whose names match the naming conventions. An example case is that of an **around** advice body. The body of this around advice is implemented as a method on the aspect class. For this method, we isolate the instructions implementing the **proceed**() call, and tag them appropriately.

4.2.3 Generating attributed class files

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. This will be replaced by a proper tag during the dynamic analysis phase.

4.3 Dynamic metric analysis with tag propagation using *J

**J* is an framework designed to perform dynamic analyses of Java programs. While it was primarily designed for computing dynamic metrics, it can be easily extended to include various other kinds of analyses. The **J* framework 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 analyzer, which internally consists of a sequence of operations organized as a pipeline structure. Each analysis in the pipeline 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 pipeline. It also includes a full set of general-purpose dynamic metric computation modules.

4.3.1 Modifications to the **J* analyzer:

Static tagging identifies bytecode that is added to support AspectJ constructs. Because only the application classes are compiled with the modified AspectJ compiler, using only the static instruction tags in an analysis results in a significant underestimate of the overhead code. For example, it is possible for parts of the Java standard library to be called in AspectJ overhead code as well as from the original application. It is thus necessary to propagate the statically-assigned tags dynamically based on the control flow of the application in order to obtain a correct measurement of overhead.

Several additions were made to **J* in order to make it recognize and use the bytecode tags. The **J* class file reader was extended to enable reading of the encoded information, and association of tags with each loaded bytecode instruction. For untagged bytecodes, a default tag value, NO_TAG, serves as a placeholder.

The most significant addition to **J* consists of the tag propagation analysis. This analysis is responsible for dynamically assigning tags to executed bytecodes by pushing tags along invocation edges in the dynamic call graph of the application. For example, in Figure 1, the *invokestatic* bytecode at offset 18 in *main*(*String*[]) has a static ADVICE_ARG_SETUP tag. This instruction invokes the *aspectOf*() method on the *ExampleAspect* aspect class. At runtime, the ADVICE_ARG_SETUP tag will be propagated to all bytecodes in the *aspectOf*() method, and all bytecodes in methods that it calls, etc. If an instruction has no static tag, and no tag has been propagated to it, it is assigned the

default tag, BASE_CODE. This guarantees that all bytecodes executed during "normal" control flow receive a dynamic tag every time they are executed.

The exception to this is when program code is entered from places the *J agent cannot observe. This can happen in the case of callbacks from JNI code, or the execution of the class loader, for example. In the cases described, where this task is especially difficult, we always opt for the conservative solution, ensuring that our analysis will never overestimate the overhead.

The meaning associated with some tags precludes their propagation. For example, the ADVICE_EXECUTE tag is used for calls to methods corresponding to advice code; the call (and subsequent return statement) are overhead, but the body of the advice is not and should be tagged ASPECT_CODE. In this and similar cases, particular tags trigger propagation of different tags. Therefore, we define a *propagation table*. This table provides a mapping from each tag to another tag which is to be used in its stead when propagating. Most tags are propagated as themselves; the exceptions are listed in Figure 4.

Current	Propagated
ADVICE_EXECUTE	ASPECT_CODE
INTERMETHOD	ASPECT_CODE
INLINE_ACCESS_METHOD	ASPECT_CODE
AROUND_CALLBACK	BASE_CODE or ASPECT_CODE
AROUND_PROCEED	BASE_CODE or ASPECT_CODE

Figure 4: Dynamic Propagation Table

The INTERMETHOD and INLINE_ACCESS_METHOD tags, like ADVICE_EXECUTE, both identify call sites which invoke user-defined aspect code, and thus have the same propagation behaviour. The AROUND_CALLBACK and AROUND_PROCEED tags identify call sites which implement the **proceed**() construct. The tag to be associated with the code called by **proceed**() depends on the calling context. The call sites will therefore propagate either BASE_CODE or ASPECT_CODE depending on the context of the advised join point. Keeping track of the depth of nested aspect code is therefore required.

The propagation algorithm is further complicated by tags which are to be propagated to bytecode instructions which already possess a tag. In such cases, it is sometimes necessary to allow the new tag to temporarily override the previous one. While tags identifying overhead code should not be overridden, it must be possible to override the tags which correspond to base or aspect code.

This is best illustrated by a simple example. An instance of an aspect can be accessed via the static method *aspectOf*() on the aspect class. This call can originate from within user-defined code, as well as from within the code inserted by the weaver to implement advice execution. In order to support the first case, the method is statically tagged ASPECT_CODE. In the second case, the invoke is tagged ADVICE_ARG_SETUP (as illustrated in Figure 1), which we wish to propagate to the method. To correctly handle all similar situations, it is necessary that instances of the ASPECT_CODE and BASE_CODE tags can temporarily be overridden by other tags during analysis. Note that in order to support the first case, an instruction tagged BASE_CODE must not be allowed to override a statically assigned ASPECT_CODE tag. In cases where it would, an ASPECT_CODE tag is propagated instead.

4.3.2 Collecting the AspectJ-specific metrics

The entire tag propagation scheme is implemented as a separate *J analysis, so that subsequent AspectJ-specific analyses can be implemented independently and easily.

Since each instruction execution now has an associated dynamically computed tag, it is a simple addition to the *J analyzer to collect the *tag mix* metric, which counts the number of instructions executed for each tag. We can also apportion other existing metrics, such as allocation counts, between the different tags.

In addition, the analyzer also tracks all dynamic guards on advice, and for each such guard computes whether the guard always succeeds, always fails, or sometimes succeeds. A count of the number of times each guard is executed is also maintained.

5 Benchmarks

In this section we provide the results and analysis for eight benchmarks which span a wide variety of uses of AspectJ. Although AspectJ is becoming quite popular there is no existing AspectJ benchmark set, thus our first challenge was to collect benchmarks that were representative of many different applications of AspectJ. All of our benchmarks were collected from public sources on the web, and we are currently working on contacting all authors so we can release the benchmark set in conjunction with the final version of this paper. We believe that providing an interesting and diverse benchmark set is an important contribution in itself.

Four of our benchmarks have equivalent Java versions, while the other four are too large and/or complex to easily produce Java equivalents. The benchmarks with Java versions are particularly valuable because we can compare runtime overheads shown by direct timing comparisons with overheads shown by our dynamic metric analysis; the timings tell us where there is observable overhead, and the metric analysis helps us understand the reasons for that overhead.

When analyzing the benchmarks we did not know what to expect *a priori*. The general belief in the AspectJ community seems to be that overheads are low. Thus, an important part of our study was to find out if and why this is true. The first four benchmarks, presented in Section 5.2, are examples where we found low overall overheads. However, somewhat to our surprise we found three benchmarks which had extremely high overheads, and for those benchmarks we have made a detailed examination of the source of the overheads, as presented in Section 5.3. Finally, we investigated one benchmark for which the aspect is intended to improve performance. We discuss it in Section 5.4.

5.1 Overall Data

Figure 5 gives an overview of the key data for all eight benchmarks. Each heading of related rows contains references to those sections of the paper that discuss the relevant metrics in detail.

At the top of the table we give the metrics that measure program size. Note that six of the benchmarks are quite large, and are composed of between 24 and 252 application classes (classes that are not part of the standard Java library). Two benchmarks, *Bean* and *Figure* are smaller, but have been selected to illustrate some standard uses of AspectJ. Also, note that as with all Java programs, the size of the programs, when the Java libraries are included, are very large, even for the small applications.

The region of the table labelled "EXECUTION TIME MEASUREMENTS" gives measurements for execution time, including both real execution times and metrics. For real execution times we consider three different configurations of the Java VM (Java HotSpot™ Client VM (build 1.4.2_02-b03, mixed mode)). In the first configuration we use the default mode which enables the client VM. For this configuration we also provide the amount of time spent in the JIT compiler, and total GC time. Since the `ajc` compiler's code generation strategy assumes a VM with a JIT that inlines, we also provide the performance for the client VM when inlining is disabled. Finally, in order to see performance of the bytecode directly, without the effect of JIT compilation, we provide the time for the interpreter configuration.

Another important aspect of performance is space usage. In the section of the chart labelled "EXECUTION SPACE MEASUREMENTS", a key metric is the *Object Allocation Density* which measures the number of objects allocated per 1000 bytecode instructions executed (*kilobytecode* or *kbc*). If the allocation density is high, then it is important to examine the "ASPECTJ TAG MIX FOR ALLOCATIONS" section at the bottom of the table to determine if significant space is used for AspectJ overhead.

In the section labelled "ASPECTJ METRICS SUMMARIZING OVERHEAD" we provide those measurements that summarize overheads. Benchmarks with high AspectJ Overhead are those most likely to have performance problems.

The sections for "ASPECTJ TAG MIX" provide a more detailed breakdown of the overheads, first considering all instructions, and then the tag mix for allocations only.

Finally, in the section labelled "ASPECTJ METRICS FOR SHADOWS", we give two metrics. The first one refers to the hot shadow metric as defined in Section 3.3.5. The second one, called "Shadow Guards Runtime Const.", is computed using the advice execution metrics defined in Section 3.3.4, and is simply the percentage of all shadow guards that always evaluate to true or always evaluate to false (*i.e.* those guards that are runtime constants and perhaps could be optimized away using a compiler analysis).

A detailed individual analysis of all benchmarks is given in the next three subsections. For each benchmark we

	DCM	ProdLine	Tetris	Bean	NullCheck	Figure	LoD	*J Pool
PROGRAM SIZE (APPLICATION ONLY) (5.1, 3.2.1)								
Classes Loaded	55	24	60	5	252	12	60	221
Instructions Loaded	17124	3108	5516	529	13927	607	27809	48220
Instructions Dead	9268	1241	2038	160	6893	241	11829	28621
Code Coverage (%)	46	60	63	70	51	60	57	41
PROGRAM SIZE WITH JAVA LIBRARIES (WHOLE PROGRAM) (5.1)								
Classes Loaded	386	320	1016	379	568	296	383	893
Instructions Loaded	108407	80931	328543	98999	103124	72304	118453	180138
EXECUTION TIME MEASUREMENTS (WHOLE PROGRAM) (5.1, 3.1)								
# instr. (million bytecodes)	3283	2396	59	145	4841	1461	2722	2314
Total time - client (sec)	7.53	1.46	125.47	1.37	30.64	4.70	90.52	8.39
JIT time - client (sec)	0.21	0.09	0.23	0.06	0.19	0.04	0.55	0.57
GC time - client (sec)	0.46	0.03	0.05	0.04	8.93	0.12	73.52	1.87
Slowdown vs. handcoded(×)				**1.00**	**20.99**	**23.50**		**1.04**
Time - client_noinline (sec)	7.50	1.52	125.46	1.42	31.98	4.94	91.77	8.48
Slowdown vs. handcoded (×)				1.02	20.77	23.52		1.05
Time - interpreter (sec)	51.72	14.55	125.49	4.54	172.57	39.57	151.32	42.60
Slowdown vs. handcoded (×)				1.16	12.32	19.98		1.04
EXECUTION SPACE MEASUREMENTS (WHOLE PROGRAM) (5.1, 3.2)								
Mem. Alloc. (million bytes)	333	28	10	109	5626	370	975	132
Obj. Allocation Density (per kbc)	2.37	0.30	2.10	23.58	33.57	10.96	12.92	0.66
#Garbage Collections	373	36	9	144	5818	488	1103	38
ASPECTJ METRICS SUMMARIZING OVERHEAD (3.3.2, 3.3.3)								
AspectJ Overhead % (whole)	**2.94**	**0.62**	**0.73**	**14.11**	**68.60**	**92.97**	**95.95**	**2.73**
#overhead/#advice (whole)	0.03	0.01	0.32	0.18	18.99	113.17	24.24	2.10
#advice/#total (whole)	0.93	0.99	0.02	0.78	0.04	0.008	0.04	0.01
AspectJ Runtime Lib % (whole)	2.53	0.00	0.06	0.00	20.31	84.89	89.34	0.00
Aspect Overhead % (app)	8.91	11.41	10.16	33.17	72.97	83.91	97.86	3.59
#overhead/#advice (app)	0.11	0.13	1.27	0.65	18.99	44.33	47.61	2.10
ASPECTJ TAG MIX FOR ALL INSTRUCTIONS (WHOLE PROGRAM) (%) (3.3.1, appendix)								
BASE_CODE	3.79	0.04	96.97	7.46	27.79	6.21	0.09	95.96
ASPECT_CODE	93.27	99.34	2.30	78.43	3.61	0.82	3.96	1.31
AspectJ Overhead (total)	2.94	0.62	0.73	14.11	68.60	92.97	95.95	2.73
INTER_METHOD		0.21		0.55				
INTER_FIELD_INIT		0.08		1.38				
INTER_CONSTR_PRE		0.06						
INTER_CONSTR_POST		0.21						
INTER_CONSTR_CONV		0.03						
ADVICE_EXECUTE	0.32	0.001	0.09	0.83	1.81	0.27	0.009	0.12
ADVICE_ARG_SETUP	1.14	0.01	0.46	5.67	27.73	0.69	0.21	1.12
ADVICE_TEST						10.41	0.18	0.87
AROUND_CONVERSION	1.15		0.002		6.72			
AROUND_CALLBACK			0.002		16.10			
AROUND_PROCEED	0.34	0.005	0.17	2.77	7.22			
CLOSURE_INIT			0.004		9.03			
AFTER_RET_EXPOSURE							0.003	
AFTER_THROWING							0.002	
CFLOW_ENTRY						38.34	45.13	
CFLOW_EXIT						43.27	50.41	
PER_OBJECT_ENTRY							0.004	0.62
ASPECT_CLASS_INIT			0.001				0.001	
INLINE_ACCESS_METHOD				2.90				
ASPECTJ TAG MIX FOR ALLOCATIONS ONLY (WHOLE PROGRAM) (%) (3.3.1, appendix)								
BASE_CODE	54.73	0.43	97.37	3.70	19.25	0.01	0.02	100.00
ASPECT_CODE	26.27	57.10	1.80	92.74			0.27	
AspectJ Overhead (total)	19.00	42.47	0.83	3.57	80.75	99.99	99.71	0.005
INTER_FIELD_INIT				3.57				
INTER_CONSTR_PRE		19.98						
INTER_CONSTR_POST		22.48						
INTER_CONSTR_CONV								
ADVICE_ARG_SETUP			0.62		53.85		0.27	
AROUND_CONVERSION	19.00		0.09					
AROUND_PROCEED			0.09		26.90			
CFLOW_ENTRY						99.99	99.43	
PER_OBJECT_ENTRY							0.009	0.005
ASPECT_CLASS_INIT	0.005	0.002	0.02				0.002	
ASPECTJ METRICS FOR SHADOWS (WHOLE PROGRAM) (%) (3.3.5, 3.3.4)								
Hot Shadows (for 90%)	3.12	33.33	4.00	100.00	2.93	33.33	12.94	66.67
Shadow Guards Runtime Const.(%)						75.00	99.64	100.00

Figure 5: Benchmark Measurements

give the source of the benchmark, a brief description of the aspects involved, and a discussion of our performance measurements.

5.2 Benchmarks with low runtime overhead

In this section we present four benchmarks which seem to confirm the general opinion that AspectJ programs have low overhead when compared to equivalent hand-woven Java programs.

As shown in the bold entries in Figure 5, the first four benchmarks either show a low amount of total overhead as computed by our metrics (*DCM*, *Prodline* and *Tetris*), or show little or no slowdown when compared to an equivalent Java program (*Bean*).

The overall conclusion is that total overhead is not a problem when: (1) each advice body represents a large amount of work, so the overhead per advice application is low; (2) the application spends most of its time in the Java library, which usually does not have advice applied to it; (3) the application spends very little time in the part of the code which has advice applied to it (so even if the overhead per advice instruction is high, the overall overhead is low); or (4) there is some noticeable overhead in the code produced by `ajc`, but a good inlining JIT compiler removes the overhead. In the following sections we expand upon these conclusions and examine each of the low-overhead benchmarks in more detail.

5.2.1 DCM

One important use of AspectJ is to provide a convenient way of instrumenting a base Java program. In this case the base program doesn't change, but aspects are used to inject instrumentation code to measure some sort of dynamic behaviour. Hassoun, Johnson and Counsell have suggested a new dynamic coupling metric (DCM) [12] and a validation of that metric using AspectJ [13]. We have implemented a more efficient version of their aspects (using a hash table with one entry per class, instead of one entry per object) which computes their proposed dynamic coupling metric. The aspects use **around** and **after** advice. The basic idea is that each constructor call and each method call is instrumented so as to increment a time step counter and to compute a dynamic coupling metric as a function of the value of the metric at the previous time step, the number of currently live objects, and the static coupling metric values. Computing this function is quite expensive as it requires iterating through the entries in a hash table, where there is one entry for each class in the application.

Since this aspect can be applied to any program, we applied it to a reasonably large Java benchmark, *Certrevsim*, which is a discrete event simulator used to simulate the performance of various certificate revocation schemes [1]. This seemed to be a suitable benchmark because it has non-trivial uses of objects and it computes something useful.

The performance measurements for the DCM aspects applied to the *Certrevsim* program are given in the column labelled "DCM" in Figure 5. As shown by the bold entry, the AspectJ Overhead is only 2.94%. Furthermore, as expected, the ASPECTJ TAG MIX metrics shows that over 93% of the instructions executed are in the aspect code. This is completely reasonable, since the advice bodies are very expensive, and they involve calls to relatively expensive hash table routines in the Java library.

A more detailed analysis does show that the overhead when looking at just the application code (Aspect Overhead (app)) is higher, at 8.91%. Furthermore, in the TAG MIX metrics for allocations, 20% of all allocations are due to AROUND_CONVERSION. These overheads do not matter for this particular benchmark, but for a benchmark with smaller advice bodies, it could be a problem, and may be worth further investigation and possible improvements to the compiler.

5.2.2 ProdLine

Intertype declarations in AspectJ allow one to define new fields, constructors and methods for existing Java classes. Lopez-Herrejon and Batory use this idea to experiment with using AspectJ to implement product lines, where a *product line* is a family of related software applications [22]. Their application experiments with a product line for related graph algorithms. This application is interesting because it heavily uses intertype declarations. The base program is effectively just a collection of empty classes (for example *Edge*, *Vertex* and *Graph*) and various aspects that use

intertype declarations to insert fields, constructors and methods into those classes (for example, *Directed*, *Undirected*, *DFS*), plus some uses of advice to splice in some method calls. The underlying implementations of the graph data structures and algorithms make heavy use of the *LinkedList* implementation in the standard Java library. We used the original benchmark as provided by the authors, but added our own module to generate random graphs, and run larger tests suitable for timing.

The performance numbers are given in the column labelled "ProdLine" in Figure 5. The overall AspectJ overhead is very low at 0.62% and almost all of the overhead comes from the intertype tags. However, note that the AspectJ overhead for the application only is much higher at 11.41%. This indicates the benchmark spends a majority of its time in the Java library. Also, a potentially important overhead is found in the ASPECTJ TAG MIX for allocations. It appears that the heavy use of intertype constructors in this benchmark leads to considerable space overhead, with about 40% of the total space used due to objects allocated in the pre and post processing of constructors that have been introduced using intertype declarations. This may be another area where a better compilation strategy can avoid some of that overhead.

5.2.3 Tetris

Graphical, interactive applications pose difficulties for analysis in that they both require human intervention and may have large variations in execution time thereby. However, they certainly form a large class of applications, and the performance and overhead of aspects in such a context is quite relevant in terms of program response times, or the cost of background computations.

We have analyzed an AspectJ version of the arcade game Tetris, available on the web [9]. In order to get reproducible results, we have modified the program to use a seeded random number generator, and to (non-interactively) replay a previously-recorded interactive session. The code to accomplish this naturally changes the program; however, the core program logic is unaltered, and the use of aspects remains the same as the original program.

Aspects in this situation were used to augment the base game with new functionality. A number of aspects were applied, though most of them apply to situations that did not happen or which happened only a few times during our sample game play. The remaining aspects (NEW BLOCKS and NEXT BLOCK in [9]) are applied to code that is exercised every few game moves, roughly in (a reduced) proportion to the number of game events, or sequences of active code execution.

Overall aspect overhead in Tetris is low, accounting for less than 1% of executed bytecodes (see the Tetris column in figure 5). This is further demonstrated by the limited use of aspects with respect to the overall program—advice constitutes only 2% of the program.

In fact, the WHOLE PROGRAM metrics are dominated by costs external to the application (startup, GUI library code). This can be seen in the relative size (INSTRUCTIONS LOADED) of the application versus the whole program, but is also apparent in the APPLICATION ONLY version of the aspect metrics. Overhead rises to over 10%, and is now greater than the cost of the aspect code itself (overhead to advice ratio is 1.27).

Program design in this case limits any apparent overhead. Of course variations in Java library/startup design may change the relative weight of application code, and thus the visibility of this overhead.

5.2.4 Bean

This is example is taken from the AspectJ primer on the website `aspectj.org`.[1] 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 whenever either the x or y coordinate is changed. This additional functionality is achieved with a pointcut and **around** advice, for each of x and y separately.

[1] An earlier version on that webpage was flawed; we are using the revision suggested in an early draft of this paper and also on the `aspectj-dev` list by Gregor Kiczales on January 14, 2004.

For comparison, we wove the AspectJ version by hand to obtain a pure Java program. Both the AspectJ and the pure Java program were compiled with the JIT inliner turned on and off. The results for these versions are shown in the Bean column of figure 5.

From the tag mix, it is apparent that this benchmark spends most of its time in aspect code, which consists of library calls introduced via intertype declarations, but there is also some **around** advice.

The overhead in terms of bytecodes executed is quite significant (14.11%). This is reflected in the execution time when run through the interpreter. However, it appears that the JIT compiler is able to eliminate most of the overhead. Without inlining turned on, there is still a discernible price in execution time of about 2%. With inlining, the JIT compiler completely eliminates the cost of the overhead instructions inserted by the AspectJ compiler.

In the context of this small benchmark, these numbers appear to justify an assumption of the AspectJ implementors, stated in [14], that the inliner eliminates most overhead of intertype declarations, and also of advice declarations where there is no dynamic residue of pointcut matching. It is however notoriously difficult to predict the effect of inlining strategies, so further benchmarking is necessary to justify the assumption in general.

5.3 Benchmarks with high overheads

Contrary to the belief that there are no significant overheads for AspectJ we did find extremely large overheads in three benchmarks. In this section we present these benchmarks, examine where the overheads come from and suggest some solutions for both the programmer (what to avoid using in AspectJ) and for compiler writers (what can be improved and some ideas on how to make those improvements).

5.3.1 NullCheck

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 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 is another case of an aspect that can be applied to any Java program, we applied it to the same Java benchmark, *Certrevsim*, that we used for the *DCM* example in Section 5.2.1. Our first experiment was to analyze the dynamic behaviour of the original *Certrevsim* benchmark and compare it with the same benchmark, but with the suggested null check aspect applied to it. Results given in Figure 6 in the column labelled "Orig. AspectJ". The results were very surprising, as the original Java benchmark runs in 1.37 seconds, but the AspectJ benchmark runs in 30.64 seconds, a 21-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.[2] To verify

[2] 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.

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 last column labelled "Hand-woven Java". The runtime for this hand-woven version is 1.46 seconds, which is only 6.6% slower than the benchmark without the checks. Thus, there is a huge gap between the performance of the AspectJ program (2136% slower) and the hand-woven program (6.6% slower). The hand-woven version does of course not admit the collection of the AspectJ metrics, and therefore that part of the table has been omitted in the relevant column.

Our metrics indicate the source of the problem. First, there is a lot of AROUND overhead — this is to be expected. However, AROUND_CONVERSION should be significant only when non-object types have to be boxed (and unboxed) upon invocation of the **around** advice body. Here we did not expect that to happen, as we only wish to process method results that are objects in the first place. However, 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*).[3] Of course, looking back to the pointcut *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 6, in the column labelled "Fixed AspectJ". Note that the runtime is still much larger than expected, 9.86 seconds, or about 7 times slower than the handwoven Java program.

The WHOLE PROGRAM dynamic metrics give us some insight into this large performance difference. The fixed AspectJ version executes 1870 million instructions, whereas the hand-woven Java version executes only 907 million instructions. However, most surprising is that even the fixed AspectJ benchmark allocates 1500 million bytes, whereas the original Java version only allocated 1.9 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 hand-woven Java benchmark loaded only 22 application classes (2421 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 attributed to the tags ADVICE_ARG_SETUP, AROUND_CALLBACK, AROUND_PROCEED and CLOSURE_INIT. Furthermore, by concentrating on the ASPECTJ TAG MIX FOR ALLOCATIONS ONLY metrics, it is clear that the **around** advice tags ADVICE_ARG_SETUP and AROUND_PROCEED account for almost 100% of the allocations in the program. 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 thus 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.

[3] This could also be observed using the Eclipse plugin for AspectJ.

	Orig. AspectJ *(all non-void methods)*	Fixed AspectJ *(only Object+ methods)*	Pruned AspectJ *(not within aspect code)*	Best AspectJ *(after returning)*	Hand-woven Java *(with null checks)*
PROGRAM SIZE (APPLICATION ONLY)					
Classes Loaded	252	138	48	48	22
Instructions Loaded	13927	8483	7633	3832	2421
Instructions Dead	6893	4959	4864	2011	1051
Code Coverage (%)	51	42	36	48	57
PROGRAM SIZE WITH JAVA LIBRARIES (WHOLE PROGRAM)					
Classes Loaded	568	456	366	366	328
Instructions Loaded	103124	97718	96868	93067	89599
EXECUTION TIME MEASUREMENTS (WHOLE PROGRAM)					
# instr. (million bytecodes)	4841	1870	1256	1032	907
Total time - client (sec)	30.64	9.86	1.70	1.67	1.46
JIT time - client (sec)	0.19	0.08	0.06	0.05	0.04
GC time - client (sec)	8.93	2.42	0.01	0.01	0.02
Slowdown vs. handcoded(×)	**20.99**	**6.75**	**1.16**	**1.14**	**1.00**
Time - client_noinline (sec)	31.98	10.22	2.01	1.86	1.54
Slowdown vs. handcoded (×)	20.77	6.64	1.31	1.21	1.00
Time - interpreter (sec)	172.57	52.93	21.21	16.70	14.01
Slowdown vs. handcoded (×)	12.32	3.78	1.51	1.19	1.00
EXECUTION SPACE MEASUREMENTS (WHOLE PROGRAM)					
Mem. Alloc. (million bytes)	5626	1500	2	2	2
Obj. Allocation Density (per kbc)	33.57	20.05	0.03	0.04	0.04
#Garbage Collections	5818	1525	2	2	2
ASPECTJ METRICS SUMMARIZING OVERHEAD					
AspectJ Overhead % (whole)	**68.60**	**50.66**	**26.78**	**14.50**	
#overhead/#advice (whole)	18.99	18.99	5.40	6.00	
#advice/#total (whole)	0.04	0.03	0.05	0.02	
AspectJ Runtime Lib % (whole)	20.31	4.03	0.00	0.00	
ASPECTJ TAG MIX FOR ALL INSTRUCTIONS (WHOLE PROGRAM) (%)					
BASE_CODE	27.79	46.67	68.25	83.08	
ASPECT_CODE	3.61	2.67	4.96	2.42	
AspectJ Overhead (total)	68.60	50.66	26.78	14.50	
ADVICE_EXECUTE	1.81	1.33	1.98	3.62	
ADVICE_ARG_SETUP	27.73	23.33	16.86	8.46	
AROUND_CONVERSION	6.72	0.67	0.99		
AROUND_CALLBACK	16.10	13.32			
AROUND_PROCEED	7.22	5.34	6.94		
CLOSURE_INIT	9.03	6.67			
AFTER_RET_EXPOSURE				2.42	
ASPECTJ TAG MIX FOR ALLOCATIONS ONLY (WHOLE PROGRAM) (%)					
BASE_CODE	19.25	0.10	99.22	99.73	
AspectJ Overhead (total)	80.75	99.90	0.78	0.27	
ADVICE_ARG_SETUP	53.85	66.61			
AROUND_PROCEED	26.90	33.28			
ASPECT_CLASS_INIT		0.001	0.78	0.27	

Figure 6: Nullcheck metrics

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

The dynamic measurements of this version are given in Figure 6 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 1.46 seconds whereas the Pruned AspectJ version executes in 1.70 seconds, which is 16% 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 (16.86%) and AROUND_PROCEED (6.94%).

Furthermore, the Pruned 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 Pruned 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 AspectJ 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.

After studying the null check aspect further, one can notice that the pruned version can be further improved by using **after returning** advice instead of **around** advice, as follows.

```
after() returning(Object lRetVal): methodsThatReturnObjects()
  { 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());
       }
  }
```

The measurements for this final version are given in the column labelled "Best AspectJ". As indicated by the ASPECTJ TAG MIX metrics, the overhead due to **around** in the Pruned version (0.99% for AROUND_CONVERSION and 6.94% for AROUND_PROCEED) is replaced by a smaller overhead due to after **after returning** (2.42 % for AFTER_RET_EXPOSURE).

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 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. Finally, programmers should be aware of situations where **after** advice could be used instead of **around** advice, since the overheads for **after** advice are lower.

5.3.2 Figure

The Figure benchmark illustrates the use of aspect-oriented programming in a figure editor [15]. Here we have selected just one aspect from that example, namely to update the display whenever one of the figure elements has been altered.

There is an interface called *FigureElement*, and all shapes that the editor support implement that interface, for example the *Point* and *Line* classes. To capture any alterations to figure elements, we define a named pointcut:

```
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));
```

The first use of **call** captures the *moveBy* operations on any of the implementations of *FigureElement*; the other disjuncts deal with alterations to individual classes.

Now when do we want to update the display? Clearly whenever a move has occurred, but not when the move is part of a more complex operation that is itself a move. Furthermore we only want to update the display when the relevant move has been successfully completed, not when it throws an exception. These considerations lead [15] to declare the following pointcut and advice:

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

The primitive *cflowbelow* checks that there is a *move* somewhere strictly below the top of the call stack. One might argue that it is not necessary to use this primitive: it would be possible to explicitly write out all the composite operations. In that case, however, the pointcut depends on intimate implementation detail, and is not robust to changes in that detail.

In the present paper, the purpose of the *Figure* benchmark is to examine the cost of using **cflowbelow**. We have thus disabled the other aspects introduced in [15], using only the core figure editor and the above advice. The core program is only a skeleton, and it does no interesting computation on its own. It is therefore to be expected that there is a very high overhead as a proportion of the total computation time. This expectation is confirmed by the first column of Figure 7: the slowdown is about a factor of 23 compared to an equivalent, hand-coded version (where all the necessary calls to *needsRepaint* are inserted by hand into the core).

To understand this huge performance penalty, it is worthwhile to examine the numbers in more detail. It appears that there is a great deal of allocation, as indicated by the EXECUTION SPACE MEASUREMENTS. Furthermore the tag mix reveals that the relevant overheads lie in the administration of CFLOW_ENTRY and CFLOW_EXIT, as well as ADVICE_TEST. The dynamic tests for **cflowbelow** are thus at the root of the problem. However, from the last row in our table we can conclude that all the dynamic tests are in fact runtime constants — so there is likely to be a significant saving possible.

As described in [23, 14], the AspectJ compiler generates code to maintain a stack to keep track of each $\mathbf{cflow}(P)$ pointcut. When a join point that matches P is encountered, a new entry is pushed onto the stack; and when such a join point terminates, the stack is popped. We examined the generated code using the Dava decompiler to gain further insight.

In this example, the entries of the stack are zero length arrays of *Object*. In general these arrays are used to store variable bindings. A pointcut can bind variables through a number of primitives such as $\mathbf{args}(x)$, which assigns the value of a join point parameter to x. If the pointcut P in $\mathbf{cflow}(P)$ binds variables, we need to keep track of them in the stack. In this benchmark program, the arrays have zero length because there are no arguments to bind.

Such a stack of zero length arrays could be more efficiently implemented using a counter; and the only check we need to make is that the argument of **cflow** does not have variable binders in it. This optimization was implemented by modifying the AspectJ compiler, and the results are displayed in the second column of figure 7. The results are a lot better, but there are still significant overheads. The slowdown compared to the hand-woven version is a factor of 7.6.

	Orig.	Counters	Opt. Counters	Single Thread	Hand-woven
PROGRAM SIZE (APPLICATION ONLY)					
Classes Loaded	12	10	10	8	6
Instructions Loaded	607	546	633	347	189
Instructions Dead	241	218	280	89	37
Code Coverage (%)	60	60	56	71	80
PROGRAM SIZE WITH JAVA LIBRARIES (WHOLE PROGRAM)					
Classes Loaded	296	295	262	259	257
Instructions Loaded	72304	72346	64503	63936	61853
EXECUTION TIME MEASUREMENTS (WHOLE PROGRAM)					
# instr. (million bytecodes)	1461	491	274	210	95
Total time - client (sec)	4.70	1.52	0.38	0.26	0.20
JIT time - client (sec)	0.04	0.03	0.02	0.02	0.01
GC time - client (sec)	0.12	0.00	0.00	0.00	0.00
Slowdown vs. handcoded(×)	**23.50**	**7.60**	**1.90**	**1.30**	**1.00**
Time - client_noinline (sec)	4.94	1.64	0.44	0.26	0.21
Slowdown vs. handcoded (×)	23.52	7.81	2.10	1.24	1.00
Time - interpreter (sec)	39.57	14.39	4.89	3.28	1.98
Slowdown vs. handcoded (×)	19.98	7.26	2.47	1.66	1.00
EXECUTION SPACE MEASUREMENTS (WHOLE PROGRAM)					
Mem. Alloc. (million bytes)	370	1	1	1	1
Obj. Allocation Density (per kbc)	10.96	0.01	0.02	0.02	0.05
#Garbage Collections	488	0	0	0	0
ASPECTJ METRICS SUMMARIZING OVERHEAD					
AspectJ Overhead % (whole)	**92.97**	**79.08**			
#overhead/#advice (whole)	113.17	32.33			
#advice/#total (whole)	0.008	0.02			
AspectJ Runtime Lib % (whole)	84.89	61.55			
ASPECTJ TAG MIX FOR ALL INSTRUCTIONS (WHOLE PROGRAM) (%)					
BASE_CODE	6.21	18.48			
ASPECT_CODE	0.82	2.45			
AspectJ Overhead (total)	92.97	79.08			
ADVICE_EXECUTE	0.27	0.81			
ADVICE_ARG_SETUP	0.69	2.04			
ADVICE_TEST	10.41	24.05			
CFLOW_ENTRY	38.34	24.46			
CFLOW_EXIT	43.27	27.72			
ASPECTJ TAG MIX FOR ALLOCATIONS ONLY (WHOLE PROGRAM) (%)					
BASE_CODE	0.01	99.65			
AspectJ Overhead (total)	99.99	0.35			
CFLOW_ENTRY	99.99	0.15			
ASPECT_CLASS_INIT		0.20			
ASPECTJ METRICS FOR SHADOWS (Whole Program) (%)					
Shadow guards runtime const.	100.00	100.00			

Figure 7: Figure Benchmark Measurements

The overheads of this counter-based implementation are due to the fact that it is necessary to maintain a counter for each **cflow** in each thread. To this end the implementation keeps a mapping from threads to counters: upon each push, pop or is-empty operation, one first needs to retrieve the relevant counter for the current thread.

To improve upon this bookkeeping, note that the thread can be assumed to be the same throughout a method body. It is therefore possible to retrieve the relevant counter once when the first **cflow** operation is done, store it in a *final* local variable, and then use the same counter throughout the method. To measure the impact of this optimization, we decompiled the output of our modified compiler, and applied the transformation by hand. The results are displayed in the third column of Figure 7: the slowdown has now been brought down to a factor of 1.90. From the difference with the interpreted version, it appears that the change enables the JIT to do a much better job.

Of course for this very simple benchmark, we know that there is only a single thread, and thus the thread-counter mapping is wholly unnecessary. The result of eliminating it from our code (again by editing the decompiled source) is shown in the penultimate column of Figure 7. It further reduces the slowdown to a factor of 1.30. It would not be too difficult to implement this optimization, with a conservative whole-program analysis to determine whether the application is single-threaded.

In [30], it is argued that by building an accurate call graph that accounts for advice as well as ordinary method calls, one may often completely eliminate the dynamic tests for **cflow**. That paper makes a lot of simplifying assumptions, however, and in fact the language under consideration is a simple aspect-oriented variant of Pascal. We expect, however, that the same techniques can be applied in the more general setting of Java, and we are working towards an implementation using the Soot analysis framework [29]. Such an implementation would truly be on a par with the hand-woven version.

5.3.3 LoD

A very interesting application of AspectJ for checking the Law of Demeter was proposed by Lieberherr, Lorenz and Wu [20] and the code to accompany the paper is also available [21]. In the paper they suggest two checkers, one for *object form* and another for *class form*. We have used the object form checker as our benchmark. The basic idea is that a program has correct *Law of Demeter object form* when an 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. To achieve this check Lieberherr et. al. have written a concise, but advanced collection of aspects which includes relatively complex pointcuts, and the use of **percflow**, **pertarget** and **cflow**.

The basic idea behind the checking code is that each calling context is associated with a hash table (through the use of **percflow**) and all valid (preferred) objects for that context are inserted into the hash table for that context. Then, at each method call, the checker verifies that the method call uses only preferred objects, otherwise it is a violation of the Law of Demeter object form.

In order to generate an interesting application of the checker, we applied it to the same simulator base code as used in Sections 5.3.1 and 5.2.1. We slightly modified the Law of Demeter code so that each error would be reported only once (in the original code an error was reported once for each dynamic instance of the error, which led to large, difficult to read, output files). After applying the Law of Demeter checker code (AspectJ code) to the simulator code base (Java code), and executing the resulting woven code, the following three object form violations were reported.

```
!! LoD Object Violation !!
   call(double certrevsim.RevocationInfo.
          getNextUpdate())
   at EndEntity.java:26
!! LoD Object Violation !!
   call(double certrevsim.RevocationInfo.
          getFirstDeltaUpdate())
   at EndEntity.java:29
!! LoD Object Violation !!
   call(RevocationInfo certrevsim.Repository.
          requestRevocationInfo())
   at Simulator.java:248
```

At first glance one might expect that the AspectJ overhead for this benchmark should be small in relation to the

amount of work done in each advice body (which includes inserting and testing for membership in hash tables). However, as shown in Figure 8 this was not the case. As demonstrated by the column labelled "Orig.", the original benchmark code has almost 96% overhead, which is entirely unexpected. By examining the ASPECTJ TAG MIX metrics it is immediately obvious that **cflow** is the problem, with 95% of the instructions and over 99% of the object allocations coming from CFLOW_ENTRY and CFLOW_EXIT. The effect of all these allocations has a huge impact on execution time, with garbage collection taking 73.52 seconds, out of a total of 90.52 seconds.

In order to examine this problem in more depth, we created a second version of the benchmark using our modified `ajc` which implements **cflow** with counters instead of stacks ("Counter" column in Figure 8). As we saw in the *figure* benchmark, the counters do improve performance substantially, reducing total running time to 4.81 seconds and garbage collection time to 0.20 seconds. However, there remains over 80% overhead due to CFLOW_ENTRY and CFLOW_EXIT, which is still higher than expected.

We examined the benchmark and found that **cflow** is used in two places, first in the definition of a pointcut, and second in a **percflow** clause. The pointcut definition is as follows.

```
public pointcut scope(): !within(lawOfDemeter..*)
  && !cflow(withincode(* lawOfDemeter..*(..))) ;

public pointcut StaticInitialization(): scope() && staticinitialization(*);

public pointcut MethodCallSite(): scope() && call(* *(..));

// ... followed by many other uses of scope()
```

Note that the definition of the *scope*() pointcut contains a **cflow** and then *scope*() is used within the definition of many other pointcuts. By examining the decompiled output of `ajc` we determined that at least 13 **cflow** stacks are created for the same **cflow**, presumably due to the inlining of the *scope*() pointcut inside the other pointcuts. Since all 13 stacks are updated on method entry and exit of some key methods, this leads to enormous overheads. Since the states of all of these stacks are the same, there is clearly room for improvement in the `ajc` code generation strategy, and further work will be needed to avoid the creation of unneeded duplicate stacks.

To show that most of the overhead is due to this use of **cflow** and not the **percflow**, we created a version of the benchmark that eliminated the **cflow** clause in the definition of the *scope*() pointcut. This is safe for our benchmark because we know for our case it is not needed. The performance measurements for this version are given in the column labelled "No cflow", and it is clear that we have removed the majority of the **cflow** overheads.

Clearly programmers like to include **cflow** pointcuts for ease of specification and for safety, so it seems important to work on efficient implementations for them. By eliminating the multiple copies of stacks, and applying the efficient counter schemes presented in the previous section, it should be possible to greatly reduce the overheads due to **cflow**.

Even after dealing with the **cflow** overheads, there still remains about 14% overhead which is due mostly to the **percflow** and **pertarget** aspects. The **pertarget** overhead shows up in two ways. First, there are some significant overheads for ADVICE_ARG_SETUP (4.93%) and ADVICE_TEST (1.99%). These overheads are larger than normal because the **pertarget** advice leads to extra code to be generated that checks if the aspect instance corresponding to the target already exists, and to allocate a new aspect instance if one does not exist. Also, the space requirements for **percflow** and **pertarget** are significant. The BASE_CODE only accounts for 2.20% of the total allocations, whereas the **percflow** accounts for 44.90% (shown in the bin for CFLOW_ENTRY), and the **pertarget** accounts for 25.83% (shown in the bin for ADVICE_ARG_SETUP, since this is where new aspect instances are created in the case of **pertarget** aspects). We expect that at least some of these space overheads could be reduced.

5.4 Benchmark for performance improvement

The final benchmark in our set is somewhat different from the others in that the aspects used for this benchmark were intended to improve upon the performance of an existing Java program.

	Orig.	Counters	No cflow
PROGRAM SIZE (APPLICATION ONLY)			
Classes Loaded	60	59	59
Instructions Loaded	27809	25871	16570
Instructions Dead	11829	11809	7491
Code Coverage (%)	57	54	55
PROGRAM SIZE WITH JAVA LIBRARIES (WHOLE PROGRAM)			
Classes Loaded	383	381	381
Instructions Loaded	118453	116412	107111
EXECUTION TIME MEASUREMENTS (WHOLE PROGRAM)			
# instr. (million bytecodes)	2722	676	113
Total time - client (sec)	90.52	4.81	0.92
JIT time - client (sec)	0.55	1.77	0.16
GC time - client (sec)	73.52	0.20	0.06
Time - client_noinline (sec)	91.77	4.31	0.86
Time - interpreter (sec)	151.32	19.27	1.94
EXECUTION SPACE MEASUREMENTS (WHOLE PROGRAM)			
Mem. Alloc. (million bytes)	975	38	38
Obj. Allocation Density (per kbc)	12.92	0.55	3.32
#Garbage Collections	1103	42	42
ASPECTJ METRICS SUMMARIZING OVERHEAD			
AspectJ Overhead % (whole)	**95.95**	**84.33**	**13.97**
#overhead/#advice (whole)	24.24	5.51	0.17
#advice/#total (whole)	0.04	0.15	0.84
AspectJ Runtime Lib % (whole)	89.34	68.92	10.59
ASPECTJ TAG MIX FOR ALL INSTRUCTIONS (WHOLE PROG.) (%)			
BASE_CODE	0.09	0.38	2.25
ASPECT_CODE	3.96	15.30	83.78
AspectJ Overhead (total)	95.95	84.33	13.97
ADVICE_EXECUTE	0.009	0.03	0.21
ADVICE_ARG_SETUP	0.21	0.83	4.93
ADVICE_TEST	0.18	0.62	1.99
AFTER_RET_EXPOSURE	0.003	0.01	0.07
AFTER_THROWING	0.002	0.009	0.06
CFLOW_ENTRY	45.13	39.31	3.78
CFLOW_EXIT	50.41	43.49	2.84
PER_OBJECT_ENTRY	0.004	0.02	0.07
ASPECT_CLASS_INIT	0.001	0.004	0.02
ASPECTJ TAG MIX FOR ALLOCATIONS ONLY (WHOLE PROG.) (%)			
BASE_CODE	0.02	2.20	2.20
ASPECT_CODE	0.27	25.95	25.95
AspectJ Overhead (total)	99.71	71.85	71.85
ADVICE_ARG_SETUP	0.27	25.82	25.83
CFLOW_ENTRY	99.43	44.90	44.90
PER_OBJECT_ENTRY	0.009	0.90	0.90
ASPECT_CLASS_INIT	0.002	0.23	0.22

Figure 8: Law of Demeter Benchmark Measurements

5.4.1 *J Pool

This benchmark is drawn from our own tool set, namely the *J tool itself. The *J analyzer reads events one-by-one from a trace file (as described in Section 4.3). Each time it reads a new event, a new object is allocated to hold this event; since there are potentially millions of events in a trace file this places significant stress on the memory manager. However, it is a property of the implementation that for any given trace file, no more than the last two events will ever be in use at any one time, which makes manual memory management of these objects possible (by reusing previously allocated ones that are guaranteed to no longer be in use, rather than allocating new ones).

This optimization is implemented by maintaining two pools of events, each pool containing one object of each of the various possible event type. At any one moment, one pool is "active" and the other is "inactive"; each time a new event would have been allocated, the appropriate type of event from the active pool is reused instead, and the active and inactive pools are swapped over. This guarantees that the last two events are always allocated from different pools, which ensures that events in use can never collide with each other.

We wrote this optimization as a piece of **around** advice; in the original program a single method (*newEvent*) is used to allocate new event objects, so this advice simply replaces calls to *newEvent* with code to reuse an object from the appropriate pool as described above. Of course, this could be implemented relatively simply by just replacing the body of *newEvent*, but this would make it harder to disable the optimization easily if required. Multiple trace files can be read simultaneously by creating multiple objects of the appropriate class; therefore the advice is implemented in a **pertarget** aspect, to ensure that different pools are used for each trace file (the current implementation actually reads just one file at a time, so the aspect could be implemented without using **pertarget**, but this would be rather more fragile).

The results of this optimization are detailed in Figure 9. The first column, "Aspect" shows it implemented as a **pertarget** aspect as described above; the second column ("Hand-woven") is for a manual implementation. Finally the third column ("No pooling") shows the unoptimized version for comparison. In each case, the *J analyzer was run on a trace generated from a short run of a program to calculate the Fast Fourier Transform.

We have provided comparisons of running time with the unoptimized version; these show that introducing the aspect provides a speedup of about 3%. In fact, there is some overhead from weaving, since the version that applies pooling directly shows a speedup of about 8%. The amount of memory allocated drops by nearly a factor of 2, and the number of garbage collections also goes down significantly (although the total time spent in garbage collection does not; we do not know why this is). In fact, the improvement in running time is more marked for runs of *J involving longer traces, but we were not able to collect the tag mix information for this due to time constraints.

6 Related Work

Most work on dynamic metrics has focused on either addressing a specific optimization problem such as memory use (*e.g.* [7, 31]), or more generally (and voluminously) on software engineering quality or complexity measures (*e.g.* [24, 34, 36]). More related work on analyzing programs through metrics is given in [8], 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 [16] 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 optimized away by modern VM's." [35] (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 [6]. 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

	Aspect	Hand-woven	No pooling
PROGRAM SIZE (APPLICATION ONLY)			
Classes Loaded	221	218	218
Instructions Loaded	48220	46869	46305
Instructions Dead	28621	27418	27843
Code Coverage (%)	41	42	40
PROGRAM SIZE WITH JAVA LIBRARIES (WHOLE PROGRAM)			
Classes Loaded	893	890	890
Instructions Loaded	180138	178787	178326
EXECUTION TIME MEASUREMENTS (WHOLE PROGRAM)			
# instr. (million bytecodes)	2314	2243	2311
Total time - client (sec)	8.39	8.03	8.68
JIT time - client (sec)	0.57	0.56	0.57
GC time - client (sec)	1.87	1.82	1.83
Speedup vs. no pooling($\times$)	**1.03**	**1.08**	**1.00**
Time - client_noinline (sec)	8.48	8.04	8.40
Speedup vs. no pooling ($\times$)	0.99	1.04	1.00
Time - interpreter (sec)	42.60	41.07	42.51
Speedup vs. no pooling ($\times$)	1.00	1.04	1.00
EXECUTION SPACE MEASUREMENTS (WHOLE PROGRAM)			
Mem. Alloc. (million bytes)	132	138	242
Obj. Allocation Density (per kbc)	0.66	0.69	1.29
#Garbage Collections	38	37	55
ASPECTJ METRICS SUMMARIZING OVERHEAD			
AspectJ Overhead % (whole)	**2.73**		
#overhead/#advice (whole)	2.10		
#advice/#total (whole)	0.01		
AspectJ Runtime Lib % (whole)	0.00		
ASPECTJ TAG MIX FOR ALL INSTRUCTIONS (WHOLE PROGRAM) (%)			
BASE_CODE	95.96		
ASPECT_CODE	1.31		
AspectJ Overhead (total)	2.73		
ADVICE_EXECUTE	0.12		
ADVICE_ARG_SETUP	1.12		
ADVICE_TEST	0.87		
PER_OBJECT_ENTRY	0.62		

Figure 9: *J Pool Benchmark Measurements

[14], 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 optimized version. Again they attribute the former very poor performance largely to the use of reflection.

In the context of middleware, Zhang and Jacobsen [37] 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 optimized design to an unoptimized design. The fact that the optimized design only achieves the same speed as the unoptimized 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 [38]. We are focusing 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 [30]. That paper is mostly a theoretical study, dealing with a small toy language, and wholly without performance experiments. The results presented here suggest that such optimization 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 [27, 28]. Popovici *et al* show an aspect-aware Java Virtual Machine that imposes relatively little overhead when aspects are inactive (1.5%–8% slowdown over a regular JVM), though that increases dramatically for active join points ($1.3\times$–$5\times$ slower than a statically-woven version).

Finally, more generic profiling methods have been applied to AspectJ programs. Hall's CPPROFJ [11] for instance, does call-path profiling of both pure Java and (limited) AspectJ programs, allowing the runtime cost of various method execution sequences to be determined. CPPROFJ is sampling-based and is naturally much more coarse-grained than our approach.

7 Conclusions

We have presented a tool set and a systematic method for analyzing the dynamic behaviour of AspectJ programs. The main technical contributions are the definition of new metrics, as well as a novel method of computing these metrics. In particular the idea of compile-time tags that are dynamically propagated allows us to accurately attribute costs to specific language features. As discussed in Section 4, the overall system for collecting our data is complex—modifications to **J* and *ajc* were non-trivial, and this system constitutes a contribution by itself. One of the more interesting and difficult components of the system is the propagation strategy, which has to be carefully designed in order to attribute data correctly. The general paradigm could be transferred to similar situations, for example when compiling ML to Java bytecode [4]. The same ideas could be integrated in a compiler that weaves the instrumentation with the generated code, instead of using a tool like JVMPI, which was the route taken in this paper.

Our benchmark set provides the first collection of programs suitable for discussing performance of AspectJ. The benchmarks we have chosen provide a good cross section of different uses of the language. We are continuing to extend the collection, in particular using some of the examples from [19]. One small difficulty consists of programs that make use of reflection: at present our propagation tools are unable to cope with reflective calls, and wrongly attribute the cost of such calls to the base program, never to the aspect. This does not invalidate our measurements of overheads, only the numbers for BASE_CODE and ASPECT_CODE.

The conventional wisdom that AspectJ does not introduce overheads seems to be explained by typical aspect usage. First, advice generally applies to user code, yet typical Java programs spend most of their time in library calls. As a percentage of the total execution time, the cost of advice is therefore insignificant in such applications. The *Tetris* benchmark illustrates this phenomenon. Some of our benchmarks (in particular *DCM*) show the opposite behaviour,

where the advice is so expensive that the overheads of applying it are dwarfed. Finally, intertype declarations have very little overheads, except when it concerns the introduction of new constructors. This is demonstrated by *Bean* and *ProdLine* respectively.

Contrary to popular belief, we did however also find significant overheads. This has led to the following guidelines for AspectJ usage, as well as promising areas for future compiler research:

Loose pointcuts. It is easy to write a pointcut that matches too many join points. Even when some of the dynamic tests fail, such loose specification can introduce significant overheads. It is particularly important to avoid **around** advice that can apply to itself, as this forces the introduction of closures. This was illustrated in the first two versions of the *Nullcheck* benchmark. Sometimes it is however not possible to tighten pointcuts to avoid this situation, so a more careful consideration of the use of closures is a fruitful topic for future research.

Advice that is too generic. When using the very generic form of **around**, this causes a significant amount of boxing and unboxing to convert arguments to the right form.

Unwarranted use of around. Because of the above, it is generally preferable to eschew **around** in favour of **after returning** when possible. The most striking example we found of this phenomenon occurred in the final version of the *Nullcheck* benchmark. In fact, that improvement was not noticed by a number of seasoned AspectJ users to whom we showed the original code, so this is an instance where our methods give new insights.

Cflow. It is tempting to write pointcuts using **cflow**, but often this introduces significant overheads. This was illustrated by three separate benchmarks, namely *Nullcheck*, *Figure* and *LoD*. Where possible, it is better to use **withincode** in lieu of **cflow**, but this is arguably less robust with respect to refactoring. Because it is not always possible to eliminate **cflow**, we investigated various ways of improving its implementation:

- When there is no argument binding, the current use of stacks in *ajc* can be replaced by counters. We have in fact implemented this optimization in *ajc*, and found it to be highly effective.
- The use of such counters is still somewhat expensive due to the fact that we have to maintain one for each thread. If the application is known to be single-threaded, significant savings are possible, as there is no need to maintain a mapping between threads and counters.
- A whole-program analysis based on the call graph can eliminate all runtime overheads of **cflow**. An initial study in this direction, for a very small toy language, was undertaken in [30].

Pertarget. The use of **per** clauses to control aspect creation carries a non-negligible overhead, as demonstrated by the **J Pool* benchmark. It might be possible to devise a static analysis which detects that only one instance will be created in a particular application.

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.

We will be making a public release of the **J* tool so that others can collect our Java-based metrics for their own programs. To benefit from these tools, one also needs a compiler that assigns static tags; for now we are using a modified version of the standard AspectJ compiler *ajc*. Inspired by the results of the present paper, we have begun the implementation of an optimizing AspectJ compiler based on Soot [29], and this compiler includes that tagging scheme.

Acknowledgements

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

References

[1] André Arnes. Certificate revocations performance simulation project. `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] Nick Benton, Andrew Kennedy, and Claudio Russell. Compiling standard ML to Java bytecodes. In *3rd ACM SIGPLAN conference on Functional Programming*, 1998.

[5] Curtis Clifton. MultiJava: Design, implementation and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch. Technical Report 01-10, Department of Computer Science, Iowa State University, Ames, Iowa, 2001.

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

[7] 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.

[8] 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.

[9] Gustav Evertsson. Tetris in AspectJ, 2003. `http://www.guzzzt.com/coding/aspecttetris.shtml`.

[10] Joseph D. Gradecki and Nicholas Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java*. Wiley, 2003.

[11] Robert J. Hall. CPPROFJ: aspect-capable call path profiling of multi-threaded Java applications. In *Proceedings of the 17th IEEE Conference on Automated Software Engineering (ASE'02)*, pages 107–116, November 2002.

[12] Youssef Hassoun, Roger Johnson, and Steve Counsell. A dynamic runtime coupling metric for meta-level architectures. In *Proceedings of the 8th European Conference on Software Maintenace and Reengineering*, page (to appear). IEEE Computer Society Press, March 2004.

[13] Youssef Hassoun, Roger Johnson, and Steve Counsell. Emprical validation of a dynamic coupling metric. Technical Report BBKCS-04-03, Birbeck College London, March 2004.

[14] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In K. Lieberherr, editor, *Aspect-oriented Software Development (AOSD 2004)*. ACM Press, 2004.

[15] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with AspectJ. *Commun. ACM*, 44(10):59–65, 2001.

[16] 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.

[17] Gregor Kiczales, John Lamping, Anurag Menhdekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *European Conference on Object-oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[18] I. Kiselev. *Aspect-oriented programming with AspectJ*. SAMS, 2002.

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

[20] 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 2nd international conference on Aspect-oriented software development*, pages 40–49. ACM Press, 2003.

[21] 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.

[22] Roberto E. Lopez-Herrejon and Don Batory. Using AspectJ to implement product-lines: A case study. Technical report, University of Texis at Austin, September 2002.

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

[24] 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.

[25] 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.

[26] Todd Millstein, Mark Reay, and Craig Chambers. Relaxed MultiJava: Balancing extensibility and modular typechecking. In *OOPSLA 2003*, pages 224–240. ACM Press, 2003.

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

[32] 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 *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.

[33] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In *Foundations of Aspect-Oriented Languages (FOAL), Workshop at AOSD 2002*, Technical Report TR #02-06, pages 1–8. Iowa State University, 2002.

[34] 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.

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

[36] 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.

[37] 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.

[38] 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

General Tags

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

ASPECT_CODE**:** 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 the ASPECT_CODE tag.

NO_TAG**:** 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. Instructions in library classes, which have not been explicitly tagged, are also assumed to have NO_TAG.

Tags to support intertype declarations

INTERMETHOD**:** An intertype method declaration results in the body of the new method being compiled into a method on the aspect class, and a dispatch method being added to the target class. The instructions in this dispatch method have this tag.

INTERFIELDGET**,** INTERFIELDSET**:** Some intertype field declarations result in accessor methods being woven into the target class. The instructions in these accessor methods have these tags.

INTERFIELD_INIT**:** Intertype field declarations result in initialization 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**,** INTERCONSTRUCTOR_POST**:** If an aspect has an intertype 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**:** This represents overhead involved in calling methods on org.aspectj.runtime.-internal.Conversions from within a constructor added by an intertype constructor declaration.

Tags applying to all kinds of advice (before, after and around)

ADVICE_EXECUTE**:** 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**:** 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**:** 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.

Tags applying to around advice only

AROUND_CONVERSION: This represents the conversion of arguments and return values related to 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, AROUND_PROCEED: 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: **Around** advice may result in the creation of closure classes. When it does,the instructions in the constructors of these classes have this tag.

Tags applying to after advice only

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

AFTER_THROWING_HANDLER: 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.

Tags to support the cflow pointcuts and percflow aspects

CFLOW_ENTRY, CFLOW_EXIT: The **cflow** and **flowbelow** 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.

An aspect that is declared with **percflow** or **percflowbelow** clause will also lead to instructions with this tag.

Tags to support perthis and pertarget aspects

PEROBJECT_ENTRY: 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.

PEROBJECT_GET, PERBOJECT_SET: These accessor methods are added to a class to acquire instances of an aspect that is declared **pertarget** or **perthis**.

Tag for exception softening due to declare soft

EXCEPTION_SOFTENER: 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.

Tags to handle privileged aspects

PRIV_METHOD, PRIV_FIELD_GET, PRIV_FIELD_SET: 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.

Miscellaneous aspect tags

CLINIT: 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.

INLINE_ACCESS_METHOD: 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.