Weave Now or Weave Later: A Test-driven Development Perspective on Aspect-oriented Deployment Models

Rakesh B. Setty, Robert E. Dyer and Hridesh Rajan

TR #08-02
Initial Version: Feb 26, 2008.

Keywords: incremental, compilation, weaving, aspect-oriented programming, load-time weaving

CR Categories:
D.3.3 [*Programming Languages*] Language Constructs and Features - Control structures
D.1.5 [*Programming Techniques*] Object-oriented Programming
D.3.4 [*Programming Languages*] Processors — Code generation; Incremental compilers; Run-time environments

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Weave Now or Weave Later: A Test Driven Development Perspective on Aspect-oriented Deployment Models

Rakesh B. Setty Robert E. Dyer Hridesh Rajan
Dept. of Computer Science, Iowa State University
{rsetty, rdyer, hridesh}@cs.iastate.edu

ABSTRACT

The choice to use static or load-time weaving techniques in the development cycle of large AspectJ programs is not clear. It is a common practice to iteratively remove errors from programs by making small changes, recompiling, and testing the change. Previous research has shown that incremental compilation of aspect-oriented programs using static weavers can take longer compared to object-oriented programs, which in turn increases the time spent in each iteration. It has been suggested that utilizing load-time weavers can potentially alleviate the problem. However, there is a trade-off involved which is the increased execution time due to the overhead involved in weaving while loading classes. In this paper, we report on a case study in which we examine the parameters that differentiate the two techniques during the edit-compile-test cycle and determine which technique is more favorable as these parameters vary. Our results show that the parameters that differentiate the techniques are the number of classes loaded, the size of the project and the number of join points executed including repetitions. We also find that load-time weaving does solve the problem of incremental compilation in aspect-oriented programming to some extent under some favorable values of the parameters mentioned. We find that the performance of static weaving with respect to load-time weaving is directly proportional to the number of classes loaded during test, and the performance of load-time weaving with respect to static weaving is directly proportional to the size of the project and the number of join points executed. Our results also show that the percentage of join points affected by aspects do not differentiate between the two techniques.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.3.4 [**Programming Languages**]: Processors — Code generation; Incremental compilers; Run-time environments

General Terms

Design, Human Factors, Languages

Keywords

incremental, compilation, weaving, aspect-oriented programming, load-time weaving

1. INTRODUCTION

Test-driven development techniques [5] have proven to be very effective in reducing the defects and improving the overall quality of software systems [28]. The key characteristics of a test-driven development process (apart from the acclaimed difference that tests play a key role in the design of the system) is that tests are executed very often. It is generally recommended that development proceeds in small increments followed by rigourous testing of these increments (commonly known as the edit-compile-test cycle).

The speed of this development technique directly affects developer's productivity in nontrivial systems [27] and a slow edit-compile-test cycle is reported to cost as much as forty percent of developer's productivity [2]. The slow down in this cycle is primarily due to the speed of incremental compilers. Fortunately, research in separate and incremental compilation for imperative [6, 42, 39, 37] and object-oriented languages [3, 10, 14, 17, 36] has significantly improved the speed of procedural and object-oriented incremental compilers and state of the art compilers tend to incrementally compile in near instant time.

Recently, aspect-orientation has emerged as a promising technique for improved separation of concerns [12, 24]. Aspect-oriented (AO) software development techniques [24, 12] are generating significant research interest. Large-scale industrial adoption is being reported [9, 27, 38, 35] and a number of books have appeared (e.g. [26, 21, 8, 13, 34]).

Aspect-oriented software development processes, like their object-oriented counterparts, can also significantly benefit from the test-driven development techniques, however, a problem remains. Some recent reports and users' mailing lists seem to suggest that AO incremental compilation tends to take longer compared to object-oriented (OO) incremental compilation [1, 41, 7, 27, 31], in some cases resorting to full builds. In a test-driven aspect-oriented software development process increased incremental compilation is likely to result in a longer edit-compile-test loop, which in turn is likely to affect developer's productivity [27, 2]. This problem may be particularly pronounced for full builds, which are reported to tempt the programmer to switch to another task entirely (e.g. email, Slashdot headlines) [27].

A number of suggestions to solve this problem are also informally provided. The primary suggestion is to use approaches that defer some compilation work, thereby decreasing the compile-time [1, 41]. One such technique is load-time deployment that instead of fully compiling aspect-oriented programs only compiles object-oriented constructs and parts of the aspect-oriented con-

structs and defers the compilation of the rest of aspect-oriented constructs. This deferred compilation is performed right before a class is loaded for execution (generally by a custom class loader).

The speed of a test-driven development technique depends on both compilation and test time, therefore, it may appear at the first glance that deferring the compilation may not have an influence on the speed of a test-driven development technique because the deferred compilation will eventually need to be performed during test execution. It turns out that due to the property of Java-like languages that only those classes are loaded that are referred to by currently loaded classes, only a subset of the total classes in an application in such languages are loaded. Therefore, the deferred compilation work only needs to be completed for a subset of classes, whereas the static approach will fully compile all classes. This is precisely where the decrease in edit-compile-test cycle shows. It is still not clear, however, whether the savings in compilation time due to deferred compilation is likely to always exceed the overhead of load-time weaving. Furthermore, the factors that influence these savings and the cost are also not rigorously analyzed in informal suggestions that advocate load-time deployment.

In this work we report on an empirical study of this tradeoff. Several others have evaluated different characteristics of AO approaches. For example, early assessments of modularity benefits were conducted by Mendhekar *et al.* [29], Kersten and Murphy [22], Walker *et al.* [40], etc. Mendhekar *et al.* [29] used RG, an environment for creating image processing systems to evaluate aspect-oriented programming. Kersten and Murphy [22] used Atlas, a web-based learning environment to evaluate aspect-oriented programming. Walker *et al.* [40] also conducted an initial assessment of aspect-oriented programming where they show that AspectJ may improve the understandability when the effect of the aspect code has well-defined scope and that AOP could change the strategies used by programmers to address tasks associated with aspect code. Hannemann and Kiczales [18] compared the object-oriented and aspect-oriented implementations of the Gang of Four design patterns [15] using qualitative metrics. Garcia et al. [16] used a set of quantitative metrics to compare the object-oriented and aspect-oriented implementations. The closest related work is by Lesiecki [27] that makes observations about the incremental compilation time of AspectJ programs. Compared to these earlier results, the empirical study presented in this work rigorously analyzes the AO test-driven software development process that serves to evaluate the potential utility of load-time deployment. In summary, this work makes the following contributions.

- Analysis of the AO incremental compilation process, which serves to provide insights into factors that contribute towards increased incremental compilation time,
- empirical results on incremental compilation time of a mature, industrial strength aspect-oriented compiler for two large projects, Eclipse [48] and Azureus [47],
- empirical results on the AO test-driven development process for two projects, Ant [45] and JBossCache [49] with significant test infrastructure,
- a rigorous analysis of factors that contribute to the speed of a test-driven development technique in both compile-time and load-time deployment models, and
- insights into scenarios where a load-time deployment approach can be successfully used for mitigating the effects of increased AO incremental compilation time.

Our results show that the size of the project, the number of classes loaded during execution and the number of join points executed including repetitions are important parameters that differentiate the static deployment and load-time deployment. The results also show that the compilation time for load-time deployment is always lower than the that for static deployment and that the gap between the two increases as the size of the project increases. The execution time of load-time deployment is almost always higher than that of static deployment. The ratio between the two increases as the number of classes loaded increases and decreases as the number of join points executed including repetitions decreases.

This paper is organized as follows. Section 2 presents necessary background. Section 3 studies the AO incremental compilation process. Section 4 introduces the factors that differentiates the two deployment models. Section 5 studies the AO test-driven development process and Section 6 concludes.

2. DEPLOYMENT MODELS

There are a number of dimensions along which AO languages differ. A dimension of interest for this work is what is called the *deployment model*. To understand the notion of deployment models, let us consider a requirement that may benefit from AO languages. Assume that we are adding a resource-sharing policy to an application, e.g. database connection sharing, thread pooling, etc. To implement this policy one can identify resource creation throughout the application and replace all such creation by a request from the resource pool instead, however, such an implementation strategy would couple all such parts of the application with the implementation of the resource-pooling requirement. Moreover, implementation of the resource pooling would be fragmented and spread across the program, which would make it hard to evolve the implementation of resource pooling.

An AO solution to this problem would localize the implementation of the resource pooling policy into a module, use declarative constructs to identify resource creation throughout the application, and use another set of declarative constructs to override such creation with a resource request instead without leaving the boundary of the module, thereby solving both problems. At some point, however, these declarative AO constructs will have to be composed (also called weaved) with the application's original code such that at execution-time the desired (interleaved) behavior is manifested. This is often called *deployment* of AO constructs.

AO languages support a variety of *deployment* models, *compile-time* deployment, where the weaving happens during compilation, *load-time* deployment [25], where the weaving happens when a class is loaded for execution by a virtual machine (VM), and *run-time* deployment [30], where the weaving happens during the application's execution. AO approaches support all three deployment models and each has respective advantages and disadvantages. For example, a compile-time deployment model is likely to incur the cost of weaving once during compilation and thereafter the compiled application will run without the need for weaving. On the other hand, such deployment model is unlikely to provide much dynamic flexibility, unless all such flexibility is anticipated and compiled into the application, which may have additional overheads.

A load-time deployment model, however, is likely to incur the cost of weaving everytime a class is loaded, however, it offers much more flexibility. The composition of AO code for resource pooling with the original code can be deferred until load-time. This for example, allows one to start an application with or without resource pooling without having to recompile it. A runtime deployment model is likely to incur the cost of weaving during execution, although such costs can be significantly reduced [11], however,

they allow one to defer the composition of AO code for resource pooling with the application until runtime. For example, one could introduce a resource pooling policy in a running webserver if the resource availability is critically low and remove it subsequently when the underlying problems are solved, without having to restart the application.

```
public class Hello {
 public static void main(String[] args){
   System.out.println("Hello");
 }
}
public aspect World {
  pointcut main(): execution(* Hello.main(..));
  after returning(): main() {
    System.out.println("World");
  }
}
```

Figure 1: A simple aspect-oriented application

As an example, consider a typical compile-time weaving technique that is demonstrated using a simple AspectJ [23] application shown in Figure 1. We emphasize AspectJ for the maturity of its design and the availability of a robust and usable implementation, however, note that other such aspect-oriented languages as Eos [32, 33] would exhibit roughly similar behavior. Our application has one class `Hello` (shown inside the white box) and one aspect `World` (shown inside the grey box). The class `Hello` declares a method `main` that prints the string `"Hello"` on the screen and exits. The aspect `World` declares that after the execution of the method `main` the string `"World"` will be printed. The execution event "execution of the method main" is a type of *join point*, the mechanism used to select the join point is called a *pointcut*, and the method-like construct that prints `"World"` is called *advice*. There are three common types of advice: before a join point, after a join point, and around a join point.

2.1 Object Code View

We compiled our *HelloWorld* application using the AspectJ compiler, *ajc*. We disassembled the class files using *javap*, the disassembler for Java. Figure 2 shows the disassembled intermediate code represented by Java byte code notations.

Figure 2 shows the disassembled code of `Hello` in the white box and the disassembled code of `World` in the grey box. As can be observed, the intermediate code to invoke `World` at join points is inserted into the class `Hello` in the method `main`. As a result, changes in `World` may require recompilation of the `Hello` class.

Now consider the same application, but now compiled with ajc for load-time deployment. The Standard Bytecode representation, shown in Figure 3, is similar to Figure 2 with a few differences. First, the `Hello` concern is free of code from the `World` concern. Second, the `World` concern has additional Java annotations attached to the class and methods that inform the load-time weaver of how to weave the `World` concern into other classes as they are loaded. As a result a change in `Hello`, which is not a crosscutting concern, will only affect the intermediate code representation of the `Hello` module. Similarly, a change in `World`, which is a crosscutting concern, will only affect the intermediate code representation of the `World` module. The changes are thus traceable to a limited number of modules at the intermediate code level, resulting in improved incremental compilation time compared to static deployment models. We study this in detail in the next section.

3. CASE STUDY I: COMPILATION TIME

```
public class Hello {
  static void main(java.lang.String[]);
0:  getstatic       #21; //Field System.out
3:  ldc             #22; //String Hello
5:  invokevirtual   #28; //Method println
    //Code inserted for aspect invocation
8:  goto            20
11: astore_1
12: invokestatic    #38; //Method World.aspectOf
15: invokevirtual   #41; //Method World.ajc$0
18: aload_1
19: athrow
20: invokestatic    #38; //Method World.aspectOf
23: invokevirtual   #41; //Method World.ajc$0
26: return
}
public class World {
  public static final World ajc$perSingletonInst;
  static {}; // Static initializer
0:  invokestatic #14; //Method ajc$postClinit
3:  goto         11
6:  astore_0
7:  aload_0
8:  putstatic    #16; //Field ajc$initFailureCause
11: return
//Advice ajc$0, constructor World, and methods
//hasAspect, aspectOf and ajc$postClinit elided.
}
```

Figure 2: An AspectJ aspect compiled to standard bytecode: the generated code for the `World` concern is in gray

Incremental compilation is defined as the property of a compiler such that a small change in syntax or semantic structure requires only a small amount of reprocessing to reflect the change [4]. The speed of incremental compilation affects the responsiveness of the integrated development environments (IDE) such as Eclipse. As Hölzle and Ungar point out, in the context of object-oriented programming and more specifically the Self language, an IDE must provide response as quickly as possible after programming changes in order to increase programmer productivity [20]. Hölzle also argues that "compilation must be quick and non-intrusive" [19]. These requirements are valid for aspect-oriented IDEs as well. A report (although slightly outdated but still relevant) on the usage of AspectJ [23] in the development of a J2EE web application for Video Monitoring Services of America showed that incremental compilation using the AspectJ compiler usually takes at least two to three seconds longer than the near instant compilation using a pure Java compiler [27]. The increased incremental compilation time may have an effect on programmer productivity. In particular, the increased incremental compilation time can potentially affect the build-test-debug cycle common in many agile software development processes. For example, Lesiecki observed that due to the increase in incremental compilation time, "human attention can wander and it can take time to re-contextualize after the compilation. This problem is particularly pronounced for the full builds, which tempt the programmer to switch to another task entirely (e.g. email, Slashdot headlines)".

3.1 Effects of Increased Incremental Compilation Time

We independently confirmed the latter observation in detail by measuring the incremental compilation time of AspectJ [46] programs after making minor modifications to the source code, when compiled with *ajc* 1.5.4, the AspectJ compiler. We studied some common changes that might occur during the development of an AO system, which are described

```
public class Hello {
  static void main(java.lang.String[]);
0:  getstatic       #21; //Field System.out
3:  ldc             #22; //String Hello
5:  invokevirtual   #28; //Method println
8:  return
}
@PointcutDeclaration("execution(*␣Hello.main(..))")
public class World {
  public static final World ajc$perSingletonInst;
  static {}; // Static initializer
0:  invokestatic #14; //Method ajc$postClinit
3:  goto         11
6:  astore_0
7:  aload_0
8:  putstatic    #16; //Field ajc$initFailureCause
11: return
//Advice ajc$0, constructor World, and methods
//hasAspect, aspectOf and ajc$postClinit elided.
}
```

Figure 3: The example from Figure 1 compiled for load-time deployment: the generated code for the `World` concern is in gray

below, and then measured the incremental compile times of such changes for both static and load-time weaving techniques. The `-XterminateAfterCompilation` flag was given to ajc when performing incremental compiles for load-time weaving (this flag stops all weaving). All tests were performed on a Linux machine with 3.2 GHz processor and 2 GB of main memory.

- **Aspect Added** - Measures the impact of adding a new aspect to an already compiled system. The aspect we added generates a trace of method executions in a package. The targeted package contained around 25 compiled class files.
- **Aspect Removed** - Measures the impact of removing an existing aspect from a system. The tracing aspect in the previous test was removed from the compiled systems.
- **Changed Pointcut** - Measures the impact of modifying a pointcut in an aspect. The number of classes advised by the pointcut was changed from 10 to 25 for both systems.
- **Changed an Un-advised Method Body** - Measures the impact of modifying a method body in a class that is not advised by an existing aspect. The tracing aspect was used and the pointcut set to target a package that did not contain the class being modified.
- **Changed an Advised Method Body** - Measures the impact of modifying a method body in a class that is advised by an existing aspect. The tracing aspect was used and the pointcut set to target the package containing the class being modified.
- **Changed Advice** - The final test was broken down into two tests: changing the body of an aspect's advice where no run-time reflective information was used and changing the body of an aspect's advice by adding the use of run-time reflective information. The tracing aspect was used again.

Figure 4 shows the incremental compilation times of the Azureus peer-to-peer application, a medium-scale system with around 3500 compiled classes and 2000 source files. In the majority of cases, a small change in the aspect resulted in a full build of the system. For this project, a full build takes around 10 seconds and may not inconvenience the programmer. The problem is more pronounced for larger systems.

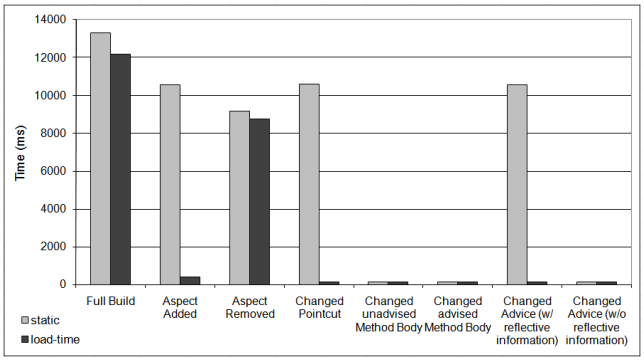

Figure 4: Incremental compile times of Azureus for static and load-time weaving

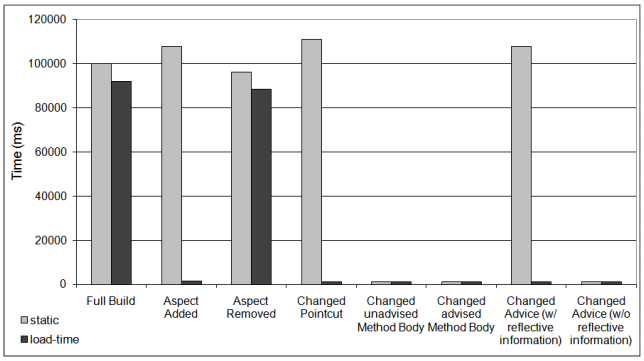

Figure 5: Incremental compile times of Eclipse for static and load-time weaving

Eclipse, a Java integrated development environment, is a large-scale system with over 23,000 compiled classes and 13,000 source files. Figure 5 shows the incremental compilation time of Eclipse. For this project, a full build takes over 1.5 minutes. With incremental builds taking this long, it is conceivable that developers would switch tasks as Lesiecki pointed out.

The two weaving techniques performed similar for both projects. For several of the small changes performed, static weaving techniques trigger a full build. The incremental compile mode of ajc runs like a server. It performs one full build and then waits for a trigger to perform an incremental build. During the initial full build, it maintains data structures to aid it in performing incremental builds. It is possible that as this feature matures, full builds will be triggered less often.

OBSERVATION 3.1. *Load-time weaving does not trigger full builds for most small changes in the source files.*

Load-time weaving techniques have a tighter correspondence between source and compiled files and thus for small changes to the source file only need to recompile a small portion of the system. This gives it the advantage of being able to perform truly incremental builds in most [1] cases. Thus for large AO projects being itera-

[1]Note that in both projects, removing an aspect from the system triggered a full build for load-time weaving. We believe this to be a bug in the incremental compiler, as all it needs to do is remove any class files generated from previously compiling the aspect.

tively developed, using load-time weaving instead of static weaving will shorten the development cycle.

4. DIFFERENTIATING PARAMETERS

The results described in the previous section showed that the time taken to compile using static weaving is always higher than that taken by load-time weaving since static weaving performs the full compilatation statically, whereas load-time weaving defers some of the compilation. However, load-time weaving takes longer time than static weaving during execution since it has to weave into the classes at the time of loading. Since the speed of test driven development process is dependent on the sum of compile time and execution time, we need to know the parameters and their impacts on these times for both the techniques to reason about their performance.

We claim that three parameters play an important role in differentiating the two techniques.

- The size of the project,
- the number of classes loaded during execution, and
- the number of join points executed including repetitions during execution.

In the rest of this section, we discuss these in detail.

4.1 Size of the Project

The size of the project can be measured in terms of the total number of join points present in the system. This parameter affects the compilation time taken by the two techniques. In the static weaving, all classes of the basecode and aspects are compiled by ajc or some AspectJ compiler which involves weaving the advice into the basecode.

We can see from Figure 4 and Figure 5 that Eclipse takes much longer to compile (with static weaving) than Azureus because Eclipse is larger. As we will show later in our experiments, static weaving takes considerably longer at compilation time than load-time weaving where the base code is compiled by the javac or some Java compiler and only the aspects are compiled by ajc or some AspectJ compiler which does not involve any weaving.

As we will discuss later, this parameter is the only one that helps load-time weaving perform better than static weaving. In other words, the time saved in the compilation due to the size of the project is the only time that load-time weaving can gain over static weaving. During execution, load-time weaving cannot take lesser time compared to static weaving since load-time weaving will involve the overhead of weaving classes as they are loaded.

4.2 Number of classes loaded

The number of classes loaded (C) is an important parameter that affects the execution time taken by load-time weaving. As the number of classes loaded increases, the execution time taken by load-time weaving increases as there is an overhead involved in matching and weaving all the applicable classes (determined by the advice as well as aop.xml) that are loaded. There is no such overhead involved in static weaving as the required classes are woven during compilation and the time taken in just loading a class without weaving is considerably less than weaving and loading. However the benefit for load-time weaving is that only those classes which are actually required during runtime will be woven, thus saving the time required to weave all the other classes. In any case, the time taken to load a class is common to both the techniques with load-time weaving involving the overhead of weaving before loading. Later in our experiments, we use another derived parameter which is the ratio of the number of classes loaded to the number of classes present in the basecode (c).

4.3 Number of join points executed

The total number of join points executed including repetitions (J) is also a parameter that affects the ratio of the total time taken by load-time weaving to the total time taken by static weaving. As the total number of join points executed increases, the overhead involved in loading the classes by load-time weaving becomes smaller and smaller compared to the total time. In other words, the more the number of join points executed, the more is the utility of a join point that was loaded. However, in no case will the execution time of load-time weaving be lower than that of static weaving, i.e. the ratio of the times taken will never be less than 1. It is just that the ratio of the times taken moves closer to 1 as the value of the parameter increases. In our experiments, we use another related parameter which is the ratio of the number of join points executed including repetitions to the total number of join points loaded (j).

5. CASE STUDY II: COMPILATION AND TEST EXECUTION TIME

We conducted several experiments on two large open source projects, Apache Ant and JBossCache, with good test infrastructures to support or reject our claims mentioned in the previous section. The primary criteria for project selection were the presence of a unit test suite and medium to large project size. We ran the experiments on a Linux machine with processor speed of 3.8 GHz and 3.5 GB of main memory.

5.1 Candidate Projects

Apache Ant is a Java-based build tool [45]. The basecode has close to 800 Java source files and over 1000 compiled class files. It has a very well developed unit test suite, with close to 300 source files.

JBossCache project aims to provide enterprise-grade clustering solutions to Java-based frameworks, application servers and custom-designed Java SE applications([49]). The basecode has almost 400 Java source files and 500 compiled class files. It also has a very well developed unit test suite, with over 400 source files. Although the project is smaller than Ant, it has a very good test infrastructure to conduct the experiments. The smaller size of the project (compared to Ant) also helps us see how the size of the project plays a role in distinguishing the static and load-time weaving techniques.

Our candidate projects have unit tests for only object-oriented parts. This is, however, not a threat to the validity of our experiment because we are only using very simple advice that can be verified by inspection as well. In the future, we could also use Raspect-like approach [44, 43] for automatically generating unit test cases for newly introduced aspects in the candidate project.

5.2 Impact of the parameters involved

Following are the notations used in the experiments that we conducted to find the impact of the parameters involved.

- c − the ratio of the number of classes loaded to the number of classes present
- C − the total number of classes loaded during execution
- j − the ratio of number of total join points executed including repetitions to the number of all join points loaded

- J – the total number of join points executed including repetitions

As we will see in the results section later, as c increases, performance of load-time weaving decreases and as j increases the performance of load-time weaving increases. Hence the ratio c/j could be a better differentiating factor.

If s is the average size of the classes loaded in terms of the number of join points present in the classes, then it can be argued that the time taken by load-time weaving to run the test case is an algebraic expression involving J, C and s like $m*J+n*C*s$ where m and n are constants. Similarly, the time taken by static weaving to run the test case is approximately $m*J$ assuming that the time taken for loading a class is very less compared to the time taken to load a class(after weaving) by load-time weaving. Hence the ratio $(m*J+n*C*s)/(m*J)$ could be another differentiating factor.

In this experiment, we study the effects of these parameters on the two techniques. We believe that these parameters have the same kind of impact in differentiating the techniques irrespective of the project since the results are for each testcase in any way. The size of the project is an important factor since that is the only factor that is in favor of load-time weaving (bigger the project, better it is for load-time weaving as that means more compilation time for static weaving). We will also see the impact of the size of the project in differentiating the two techniques.

5.2.1 Normalization Factor

In comparing the times taken by the load-time and static weaving techniques, there are some additional factors that may affect them. One of them is the join point density which may be roughly defined as the ratio of the number of join points per line of code in the code that was executed. One more factor is the times taken to execute by different kinds of join points. In general, we saw cases where the execution times were lesser when there were more number of join points executed and viceversa. However this will not be the case if we consider the difference in time between the original unmodified execution time and the execution time with the aspect for the technique that we are considering. Hence we have used the original unmodified time as the normalization factor to rule out the effect of the extraneous factors mentioned above.

5.2.2 Experimental Setup

In this experiment, we run individual testcases present in the Apache Ant [45] and JBossCache [49] projects after adding an aspect that advises every join point. The percentage of join points covered is maintained at 100% to eliminate the possibility of percentage of join points covered in different testcases to vary. To keep the percentage of join points covered to 100%, we use the universal aspect shown in Figure 6.

For each testcase, we see what fraction of classes were loaded, what fraction of the join points loaded get executed and the ratio of the total times taken by the two techniques. For each testcase and for each technique, we do a rebuild followed by a run of the testcase.

When we compile the project with the *UniversalAspect* using ajc with the showWeaveInfo option, we can know the number of join points woven from the trace generated which will give us the total number of join points present in the basecode.

We needed to find out the values of the parameters c, C, j and J. To get them, we did the following. We had the whole basecode covered by the *traceAll* pointcut, for which we had a suitable advice that recorded the class and the join point that invoked the advice making use of reflection API provided by AspectJ. Then we ran an experiment with such an advice to know things like the

```
1 pointcut traceAll()
2   :(
3     call(* *.*(..))
4     || execution(* *.*(..))
5     || handler(*)
6     || get(* *)
7     || set(* *)
8     || initialization(*.new(..))
9     || preinitialization(*.new(..))
10    || staticinitialization(*)
11  )
12  &&
13  !(
14    within(*..UniversalAspect+)
15    || get(* *UniversalAspect.*)
16    || set(* *UniversalAspect.*)
17    || initialization(*UniversalAspect.new(..))
18    || preinitialization(*UniversalAspect.new(..))
19    || staticinitialization(*UniversalAspect)
20  );
```

Figure 6: UniversalAspect that traces all join points

classes loaded, the number of classes loaded, the join points that got executed, the number of times each join point got executed and so on. We used a hashmap to store this information and dumped them to files to get the values of the mentioned parameters. Only the values of the parameters mentioned were recorded from such an experiment. The time taken by the experiment with such an advice was not considered for experiment results as it had unpredictable factors like I/O. To measure the time taken, we used the aspect with which we wanted to test the performance.

5.2.3 Impact of Size of the project

Figures 7 and 8 show how each individual test performs with respect to compile+test time for Ant and JBossCache respectively. The Y-axis represents the compile+test time taken by each testcase.

Figure 9 shows how the two techniques - load-time weaving and static weaving fare for Ant and JBossCache projects. As mentioned earlier, Ant is almost double the size of JBossCache. The Y-axis represents the number of tests that performed better for the technique mentioned in the graph.

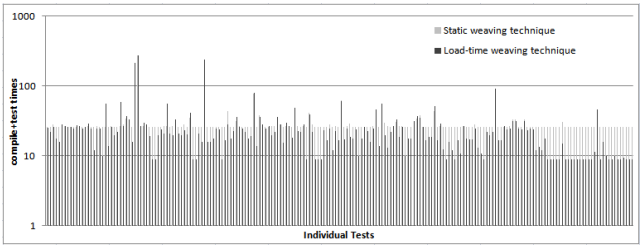

Figure 7: The comparison of the compile+test times for the Ant project. Observe that load-time weaving performs better for most number of test cases

OBSERVATION 5.1. *For individual tests, with respect to compile+test times, load-time weaving performs better for most of the testcases of Ant which is a bigger project with smaller testcases, whereas static weaving performs better for most of the testcases of JBossCache which is a smaller project with bigger testcases.*

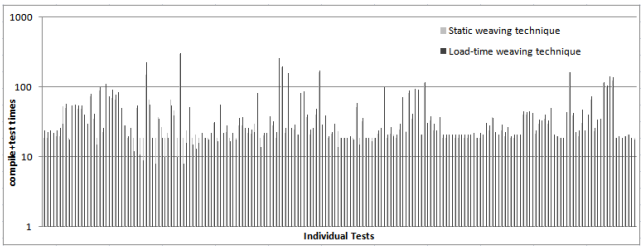

Figure 8: The comparison of the compile+test times for the JBossCache project. Observe that static weaving performs better for most number of test cases

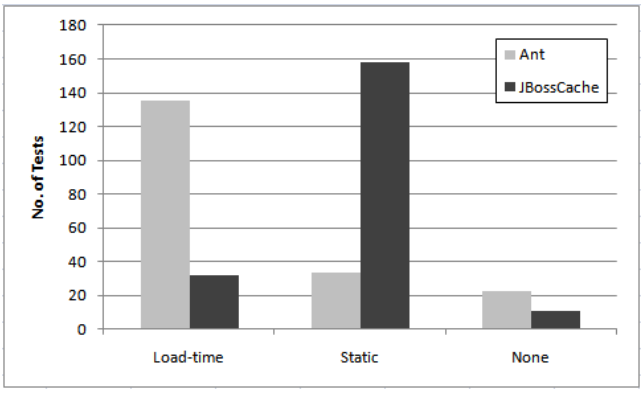

Figure 9: The comparison of the two techniques for the projects Ant and JBossCache. The size of Ant is almost the double the size of JBossCache and execution times of most tests in Ant are smaller compared to tests in JBoss. Observe that load-time weaving performs better for most testcases in Ant, while static weaving performs better for most testcases in JBossCache.

From Figures 7, 8 and 9, we can see that load-time weaving performs better for most of the testcases in Ant where the compilation time is a big factor as there are more number of classes to compile and also in general, most test cases are small enough. However for JBossCache which is a smaller project, the compilation time is not too significant fraction of the total time, as in general, the execution times are much higher compared to the compilation times. As a result, we can see static weaving performing better for most testcases of JBossCache. Hence our hypothesis that the load-time weaving performs better overall as the size of the project is more is validated.

5.2.4 *Impact of No. of classes loaded on the test-times*

Figure 10 shows how the normalized test times taken by the compilation techniques vary as the fraction of classes loaded (c) vary. Each data point in the graph represents a testcase of the project indicated by the corresponding legend. The X-axis represents the fraction of classes loaded i.e. the ratio of classes loaded to the classes present. The Y-axis represents the ratio of normalized test time taken by load-time for the testcase to the test time taken by static weaving for the same testcase.

OBSERVATION 5.2. *As the number of classes loaded increases, the test time taken by load-time weaving increases faster than static weaving.*

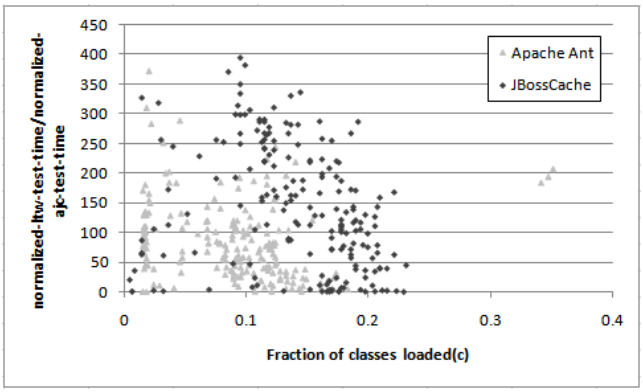

Figure 10: The effect of the fraction of classes loaded c on the normalized test times taken by the two techniques for Ant and JBossCache. Observe that as c increases, the ratio of normalized execution time taken by load-time weaving gets worse compared to static weaving

In load-time weaving, the classes are woven as they are loaded. This is an overhead while running the tests. This is advantageous when the classes loaded are less as that means less classes are woven. However, as the classes loaded increases, the overhead at runtime increases. Hence static weaving should perform better as the number of classes loaded increases. Figure 10 does not clearly support this since the experimental setup does not ensure that the number of join points executed including repetitions is not kept constant throughout (as it only involves running the individual tests). In fact we observed that as the number of classes loaded increased, so did the number of join points executed including repititions in general. However, the ratio c/j will resolve this issue and hence will give a better indication as can be seen from Figure 12.

5.2.5 *Impact of number of join points executed including repetitions on test-times*

Figure 11 shows how the normalized test times taken by the compilation techniques vary as the ratio of total join points executed including repetitions to distinct join points loaded vary. Each data point in the graph represents a testcase of the project indicated by the corresponding legend. The X-axis represents the ratio of total join points executed including repetitions to distinct join points loaded. The Y-axis again represents the ratio of normalized test time taken by load-time for the testcase to the test time taken by static weaving for the same testcase.

OBSERVATION 5.3. *As the number of join points executed including repetitions increases, the test time taken by load-time technique decreases faster than static weaving.*

The ratio of the total join points executed including repetitions vs. join points loaded (j) also plays a role in determining which compilation technique performs better. As this ratio increases, the overhead in loading a class becomes much less compared to the time taken in executing the join points inside it i.e. the time taken to run the test by load-time weaving becomes closer to that of static weaving. Figure 11 supports this hypothesis by showing that load-time weaving gets closer to static weaving as j increases.

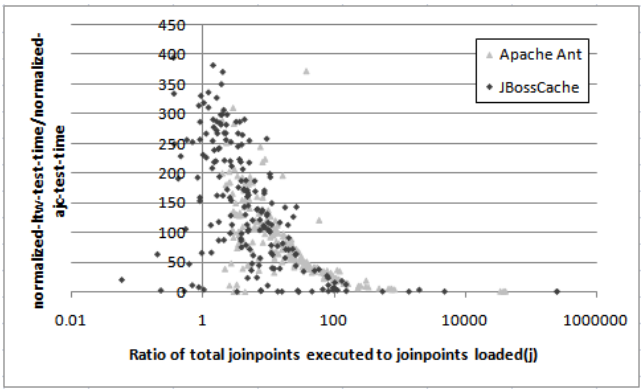

Figure 11: The effect of the ratio of total join points executed including repetitions vs. join points loaded(j) on the normalized test times taken by the two techniques for Ant and JBossCache. Observe that as j increases, the ratio of normalized execution time taken by load-time weaving gets closer to that of static weaving

5.2.6 *Impact of c/j on test-times*

Figure 12 shows how the normalized test times taken by the compilation techniques vary as the ratio c/j vary. Each data point in the graph represents a testcase of the project indicated by the corresponding legend. The X-axis represents the ratio c/j. The Y-axis again represents the ratio of normalized test time taken by load-time for the testcase to the test time taken by static weaving for the same testcase.

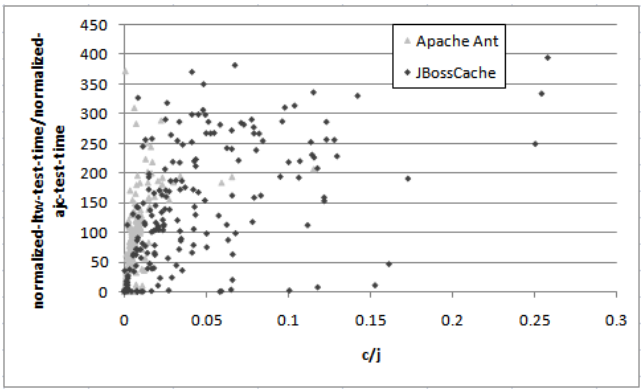

Figure 12: The effect of the ratio c/j on the normalized test times taken by the two techniques for Ant. Observe that as c/j increases, the ratio of normalized execution time taken by load-time weaving gets worse compared to the static weaving

OBSERVATION 5.4. *As c/j increases, the test time taken by load-time technique increases faster than static weaving.*

Since both the fraction of classes loaded(c) and the ratio of total join points executed vs. join points loaded (j) have a say on determining which technique performs better although in opposite ways, it is conceivable that their ratio could be a better parameter to determine which technique performs better. Figure 12 supports this hypothesis by showing that as the ratio c/j increases, load-time weaving gets closer to static weaving.

5.2.7 *Impact of C and J on test-times*

Figure 13 shows how the normalized test times taken by the compilation techniques vary as the ratio $(J + 100 * C * s)/J$ vary. Each data point in the graph represents a testcase of the project indicated by the corresponding legend. The X-axis represents the ratio $(J + 100 * C * s)/J$. The Y-axis again represents the ratio of normalized test time taken by load-time for the testcase to the test time taken by static weaving for the same testcase.

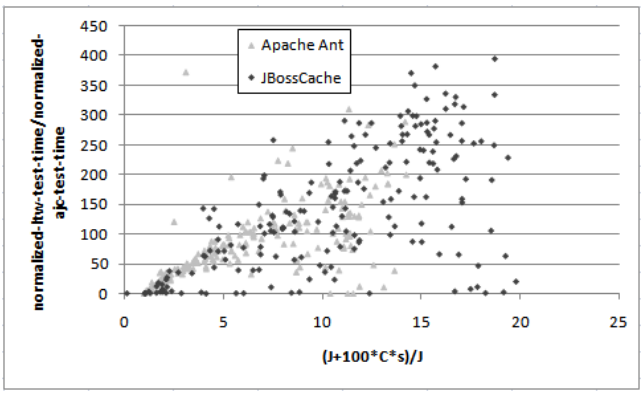

Figure 13: The effect of the ratio $(J + 100 * C * s)/J$ on the normalized test times taken by the two techniques for Ant and JBossCache. Observe that as $(J + 100 * C * s)/J$ increases, the ratio of normalized execution time taken by load-time weaving gets worse compared to the static weaving

OBSERVATION 5.5. *As $(m * J + n * C * s)/(m * J)$ increases, the test time taken by load-time weaving increases faster than static weaving.*

The hypothesis that the ratio of the test-times of load-time weaving to static weaving is $(m * J + n * C * s)/(m * J)$ is supported by Figure 13. For the purpose of this graph, the constants m, n are taken as 1 and 100 respectively although the trend remains the same irrespective the values chosen for them.

5.2.8 *Analysis*

All the parameters discussed so far i.e. c, j, C, J all directly affect the execution times of the tests. So, we saw graphs that involve test-time rather than total-time. Size of the basecode is the only differentiating factor of compilation times of the techniques. The total time depends on both execution time(test-time) and the compilation time(compile-time).

The results of these experiments clearly demostrate that two of the parameters that we mentioned - the number of classes loaded and the number of join points executed including repetitions play major role in differentiating between the two techniques. We also saw that load-time weaving performs better when the size of the project is big and the tests are smaller meaning the execution times are low compared to the total times taken including the compilation times. Section 4.1 and subsection 5.2.3 shows how the size of the project affects the compilation times of the two techniques.

5.3 Impact of join point Variation

It is natural to wonder if varying the aspect code helps in differentiating the techniques. We can vary the kind of pointcuts and the percentage of basecode covered by varying the aspect. The percentage of base code covered is the percentage of join points in

the base code that is advised by the aspects present in the project. In this experiment, we will see if varying the aspect code helps in differentiating between techniques.

5.3.1 *Experimental Setup*

In this experiment, we add an aspect to the source code of Ant and JBossCache projects which are pure java projects. We keep modifying the pointcut such that it covers different percentages of join points ranging from 2% to 100%. We used showWeaveInfo option of ajc to determine the number of join points covered by the aspect during compilation. For each value of the percentage of join points covered, we compile(full build) the project and run the testsuite that is part of the project and compare both the compile time as well as compile+test time of static and load-time weaving techniques. We used longer advice to ensure that the execution time is much higher than the noise that could be introduced due to extraneous factors. Due to noise factor, shorter the advice, bigger the role of noise in the results. However, the trend should remain the same irrespective of the length of the advice.

5.3.2 *Impact of join point coverage on the compilation times*

Figures 14 and 15 compares the compilation times as the percentage of join points covered by the aspect is varied for Ant and JBossCache projects. Here, the X-axis represents the percentage of the join points affected by the advice. The Y-axis represents the time taken in seconds to do a full build of the project with the advice. Lower the time taken, better is the technique used.

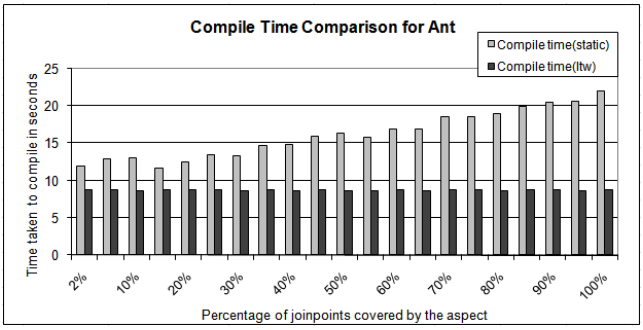

Figure 14: The comparison of compilation times for Ant as the percentage of join points covered is varied. Observe that as the percentage of join points affected increases, the compilation times taken by static weaving increases whereas for the load-time weaving it remains roughly the same.

From Figures 14 and 15, we can see that load-time weaving performs far better than static weaving for all cases. Also, the compilation time for load-time weaving remains pretty much the same irrespective of the percentage of join points covered by the aspect. This is so because pure java classes are compiled by javac and the aspect is compiled separately by ajc in load-time weaving. Hence changing the aspect does not affect the compilation of the java classes. Also, the compilation time for static weaving increases linearly as the percentage of join points covered by the aspect increases. This is because the advices are weaved in to the classes during compilation itself.

5.3.3 *Impact of join point coverage on the total times*

Figures 16 and 17 compares the compilation+test times as the percentage of join points covered by the aspect is varied for Ant and

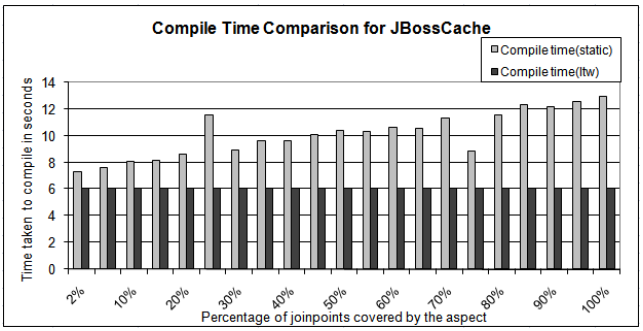

Figure 15: The comparison of compilation times for JBossCache as the percentage of join points covered is varied. Observe that as the percentage of join points affected increases, the compilation times taken by static weaving increases whereas for the load-time weaving it remains roughly the same.

JBossCache projects. Once again, the X-axis represents the percentage of the join points affected by the advice. The Y-axis represents the time taken in seconds to do a full build of the project with the advice and run the entire testsuite associated with the project. Again, lower the time taken, better is the technique used.

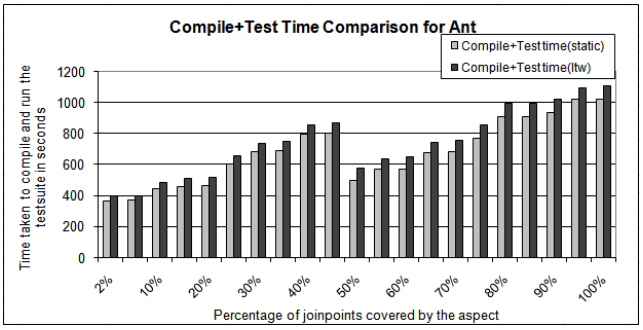

Figure 16: The comparison of compilation+test times for Ant as the percentage of join points covered is varied. Observe that even as the percentage of join points affected varies, the total time taken by load-time weaving always remains higher than that of static weaving.

OBSERVATION 5.6. *For regression testing, both Ant and JBossCache results confirm that percentage of join points covered has no role in determining which technique performs better.*

From Figures 16 and 17, we can see that static weaving always performs better than load-time weaving. This is in spite of load-time weaving performing better during compilation. We can see that the compilation+test time keeps increasing linearly for both static and load-time techniques till the percentage of join points is about 45% and then there is a fall followed by a similar increase. This could be because even though the percentage of join points is increased, it could be the case that the percentage of join points covered in the classes that are actually tested is decreased. But what matters to us is that static weaving consistently performs better irrespective of the percentage of join points covered by the aspect. This shows that the aspect code does not help in differentiating between the two techniques.

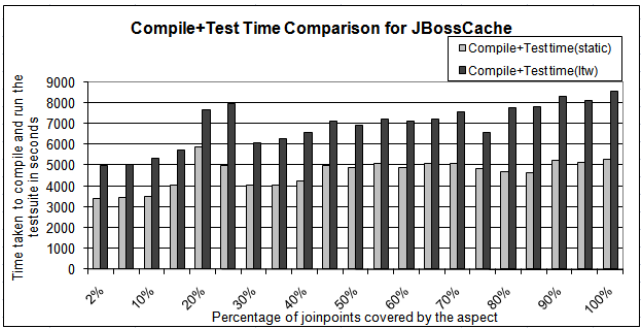

Figure 17: The comparison of compilation+test times for JBossCache as the percentage of join points covered is varied. Observe that even as the percentage of join points affected varies, the total time taken by load-time weaving always remains higher than that of static weaving.

6. CONCLUSION AND FUTURE WORK

In this paper, we have shown that incremental compilation is an issue in big AspectJ projects. We then compared the static weaving and load-time weaving techniques and showed that load-time weaving solves this problem to an extent under some favorable values of the parameters that differentiate between the two techniques which are the size of the project, the number of classes loaded, and the number of join points executed including repetitions. In particular, we have shown that:

- as the size of the project increases the compile time taken by static weaving increases faster than that of load-time weaving;
- as the number of classes loaded increases, the test time taken by load-time weaving increases faster than that of static weaving;
- as the number of join points executed including repetitions increases, the test time taken by load-time weaving increases faster than that of static weaving; and
- the percentage of join points affected by aspects do not differentiate between the two techniques.

Based on these results, we plan to automate a process which given an aspect-oriented program, will predict and recommend the more efficient technique. One way to design such a process could be using program analysis techniques to roughly calculate the values of the parameters that differentiate the techniques and then based on the values choose the more suitable technique. For example, the number of classes loaded during a test execution can be conservatively computed using Control Flow Graphs(**cite**).

Acknowledgements

This work is supported in part by the National Science Foundation under grant CNS-0627354. Thanks to Yuly Suvorov for helping with some implementation.

7. REFERENCES

[1] S. Aguiar. Re: need suggestion on how to. `http://dev.eclipse.org/mhonarc/lists/aspectj-users/msg07377.html`, Jan 2007.

[2] G. Ammons. Grexmk: Speeding up scripted builds. In *Fourth International Workshop on Dynamic Analysis (WODA)*, may 2006.

[3] D. Ancona, F. Damiani, S. Drossopoulou, and E. Zucca. Polymorphic bytecode: compositional compilation for java-like languages. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 26–37, New York, NY, USA, 2005. ACM Press.

[4] L. V. Atkinson, J. J. McGregor, and S. D. North. Context sensitive editing as an approach to incremental compilation. *The Computer Journal*, 24(3):222–229, 1981.

[5] K. Beck. *Test-driven Development: By Example*. Addison-Wesley Professional, 2003.

[6] L. Cardelli. Program fragments, linking, and modularization. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 266–277, New York, NY, USA, 1997. ACM Press.

[7] M. Chapman. Making aspectj development easier with ajdt. `http://www.infoq.com/articles/aspectj-with-ajdt`, Nov 2006.

[8] A. Colyer, A. Clement, G. Harley, and M. Webster. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*. Pearson Education, 2005.

[9] A. Colyer, R. Harrop, R. Johnson, A. Vasseur, D. Beuche, and C. Beust. Point: Aop will see widespread adoption/counterpoint: Aop has yet to prove its value. *IEEE Software*, 23(1):72–75, 2006.

[10] S. Drossopoulou, S. Eisenbach, and D. Wragg. A fragment calculus - towards a model of separate compilation, linking and binary compatibility. In *14th Annual IEEE Symposium on Logic in Computer Science (LICS'99)*, page 147, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

[11] R. Dyer and H. Rajan. Nu: a dynamic aspect-oriented intermediate language model and virtual machine for flexible runtime adaptation. In *AOSD '08: Proceedings of the 7th International Conference on Aspect-oriented Software Development*, page To appear, New York, NY, USA, April 2008. ACM.

[12] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001.

[13] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. pages 21–35. Addison-Wesley, Boston, 2005.

[14] M. Flatt and M. Felleisen. Units: cool modules for hot languages. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 236–248, New York, NY, USA, 1998. ACM Press.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[16] A. Garcia, U. Kulesza, C. Sant'Anna, C. Lucena, E. Figueiredo, and A. von Staa. Modularizing design patterns with aspects: a quantitative study. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 3–14, New York, NY, USA, 2005. ACM Press.

[17] N. Glew and G. Morrisett. Type-safe linking and modular assembly language. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 250–261, New York, NY, USA, 1999. ACM Press.

[18] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 161–173. ACM Press, 2002.

[19] U. Hölzle. *Adaptive optimization for self: reconciling high performance with exploratory programming*. PhD thesis, Stanford, CA, USA, 1995.

[20] U. Hölzle and D. Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst.*, 18(4):355–400, 1996.

[21] I. Jacobson and P.-W. Ng. *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley, 2005.

[22] M. Kersten and G. C. Murphy. Atlas: a case study in building a web-based learning environment using aspect-oriented programming. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 340–352, New York, NY, USA, 1999. ACM Press.

[23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001*, pages 327–353. Springer-Verlag, Hungary, June 2001.

[24] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland, June 1997. Springer-Verlag.

[25] G. Kniesel, P. Costanza, and M. Austermann. Jmangler-a framework for load-time transformation of java class files. In *SCAM 2001*, pages 100–110. IEEE Computer Society, 2001.

[26] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003.

[27] N. Lesiecki. Applying AspectJ to J2EE application development. In *AOSD '05*, 2005.

[28] E. M. Maximilien and L. Williams. Assessing test-driven development at ibm. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 564–569, Washington, DC, USA, 2003. IEEE Computer Society.

[29] A. Mendhekar, G. Kiczales, and J. Lamping. RG: A case-study for aspect-oriented programming. Technical Report SPL97-009 P9710044, Xerox PARC, February 1997.

[30] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02*, pages 141–147, USA, 2002. ACM Press.

[31] H. Rajan, R. Dyer, Y. Hanna, and H. Narayanappa. Preserving separation of concerns through compilation. In *SPLAT 06*, Mar 2006.

[32] H. Rajan and K. J. Sullivan. Eos: instance-level aspects for integrated system design. In *ESEC/FSE-11*, pages 297–306, Sept 2003.

[33] H. Rajan and K. J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *ICSE '05*, pages 59–68, USA, 2005. ACM Press.

[34] A. Rashid. *Aspect-Oriented Database Systems*. Springer-Verlag, 2004.

[35] D. Sabbah. Aspects: from promise to reality. In *AOSD '04*, pages 1–2, USA, 2004. ACM Press.

[36] Z. Shao. Typed cross-module compilation. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 141–152, New York, NY, USA, 1998. ACM Press.

[37] Z. Shao and A. W. Appel. Smartest recompilation. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 439–450, New York, NY, USA, 1993. ACM Press.

[38] D. Thomas. Transitioning aosd from research park to main street. In *KeyNote: AOSD '05*, USA, 2005. ACM Press.

[39] W. F. Tichy. Smart recompilation. *ACM Trans. Program. Lang. Syst.*, 8(3):273–291, 1986.

[40] R. J. Walker, E. L. A. Baniassad, and G. C. Murphy. An initial assessment of aspect-oriented programming. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 120–130, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[41] D. Wampler. Re: age-old incremental compile question. `http://dev.eclipse.org/mhonarc/lists/aspectj-users/msg07436.html`, Jan 2007.

[42] A. L. Wolf, L. A. Clarke, and J. C. Wileden. Interface control and incremental development in the pic environment. In *ICSE '85: Proceedings of the 8th international conference on Software engineering*, pages 75–82, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[43] T. Xie and J. Zhao. A framework and tool supports for generating test inputs of aspectj programs. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 190–201, New York, NY, USA, 2006. ACM.

[44] T. Xie, J. Zhao, D. Marinov, and D. Notkin. Detecting redundant unit tests for aspectj programs. In *17th International Symposium on Software Reliability Engineering (ISSRE'06)*, pages 179–190.

[45] The Ant project: a Java based build tool. `http://ant.apache.org`.

[46] AspectJ programming guide. `http://www.eclipse.org/aspectj/`.

[47] The Azureus project: a peer-to-peer file sharing application. `http://www.azureus.com`.

[48] The Eclipse project. `http://www.eclipse.org`.

[49] The JBoss Cache project: open source data grids. `http://labs.jboss.com/jbosscache/`.