Journal of Universal Computer Science, vol. 13, no. 6 (2007), 753-766
submitted: 19/1/07, accepted: 22/3/07, appeared: 28/6/07 © J.UCS

Optimized Compilation of Around Advice for Aspect Oriented Programs

Eduardo S. Cordeiro, Roberto S. Bigonha, Mariza A. S. Bigonha, Fabio Tirelo

(Universidade Federal de Minas Gerais, Brazil
{cordeiro, bigonha, mariza, ftirelo}@dcc.ufmg.br)

Abstract: The technology that supports Aspect-Oriented Programming (AOP) tools is inherently intrusive, since it changes the behavior of base application code. Advice weaving performed by AspectJ compilers must introduce crosscutting behavior defined in advice into Java programs without causing great performance overhead. This paper shows the techniques applied by the *ajc* and *abc* AspectJ compilers for around advice weaving, and identifies problems in code they produce. The problems analyzed are *advice and shadow implementation repetition* and *context variable repetition*. Performance gain provided by solving these problems is discussed, showing that bytecode size, running time and memory consumption can be reduced by these optimizations. It is assumed that the reader is familiar with AOP and AspectJ constructs.

Key Words: Aspect-Oriented Programming, Optimized Compilation, Advice Weaving
Category: D.3.4

1 Introduction

Advice weaving is the process of combining crosscutting behavior, implemented in advice, into the classes and interfaces of a program. AspectJ defines three types of advice, which are activated upon reaching certain points in the execution of programs: before and after advice are executed in addition to join points; around advice may completely replace join points, though a special *proceed* command activates these points at some moment after the advice execution has begun.

The compilation of the *proceed* command in around advice at *bytecode* level requires join points to be extracted to their own methods. Furthermore, this command might appear inside nested types in the advice body, which requires passing context from the advice's scope to extracted join points. While discussing around advice weaving, join points are also called shadow points, or simply *shadows*. During weaving, a join point comprising a single Java command is often composed of several *bytecode* instructions.

There are currently two major AspectJ compilers: the official AspectJ Compiler (*ajc*) [AspectJ Team, 2006], and the extensible, research-oriented Aspect-Bench Compiler (*abc*) [Aspect Bench Compiler Team, 2006]. *ajc* builds on the JDT Java compiler[1], and provides incremental compilation of AspectJ programs. It accepts Java and AspectJ code, as well as binary classes, and produces modified classes as output. *abc* also accepts Java and AspectJ code, and its output is semantically equivalent to that of *ajc*, but instead of providing fast compilation by means of an incremental build process, *abc* produces optimized *bytecode*. This compiler is also a workbench for experimentation with new AspectJ constructs and optimizations, providing researchers with extensible front- and back-ends.

[1] `http://www.eclipse.org/jdt`

These compilers apply the same basic techniques for weaving before and after advice. Around advice, however, is woven differently. Performance analyses on a benchmark of AspectJ programs showed that around advice is one of the performance degradation agents in code produced by the *ajc* compiler [Dufour et al., 2004]. Based on this insight, the developers of *abc* created another approach for around advice weaving [Kuzins, 2004, Avgustinov et al., 2005].

Both approaches for around advice weaving, however, still present problems related to repeated code generation. These problems are due to advice inlining and shadow extraction for advice applications, and can be fixed by small modifications in the advice weaving process. The remainder of this paper describes these problems, their proposed solutions and results obtained from their application to a small set of AspectJ programs.

1.1 AspectJ Compilers

As described in [Hilsdale and Hugunin, 2004], the *ajc* compiler is built upon the extensible JDT Java compiler, which allows the introduction of hooks in the compilation process that modify its behavior. These hooks are then used to adapt the front- and back-ends to compile both Java and AspectJ source-code. Java code for classes and interfaces is directly transformed into *bytecode*. Definitions of aspects, however, are handled in a different way: first, *bytecode* is produced to implement aspects as classes, so that code defined in advice and methods can be executed by standard JVMs. Finally, after *bytecode* has been generated for both Java and AspectJ source, the *weaver* introduces crosscutting behavior defined in aspects into the *bytecode* for the program's classes and interfaces, using crosscutting information gathered from the parsing phase.

During compilation, in-memory representations of *bytecode* are used for code generation and weaving, and actual *bytecode* files are only generated at the end. The *ajc* compiler uses BCEL [Dahm et al., 2003] as a *bytecode* manipulation tool. BCEL interprets *bytecode* contained in class files, and builds in-memory representations of the classes and interfaces they define. It provides facilities for adding and removing methods and fields to existing classes, modifying method bodies by adding or removing instructions, and creating classes from scratch. Its representation of *bytecode* is very close to its definition [Lindholm and Yellin, 1999], providing direct access to such low-level structures as a class' constant pool.

Extensibility in the *abc* compiler, as described in [Avgustinov et al., 2004], is achieved by combining two frameworks: Polyglot [Myers, 2006] for an extensible front-end, and Soot [Vallée-Rai et al., 1999] for an optimizing, extensible back-end. Polyglot is a Java LALR(1) parser, and its grammar can be modified to add or remove productions. Soot implements several optimizations for Java *bytecode*, including peephole and flow-analysis optimizations such as copy and constant propagation. Extensions are linked to Soot at runtime, via a command line flag, thus requiring no modifications to its source code. This extension model, however, isn't flawless, and it can be difficult to implement optimizations that modify the weaving algorithms for existing AspectJ constructs. Difficulties found during the development of this work are presented further in this paper.

Out of the four intermediate representations provided by Soot, *abc* uses only the Jimple representation. Jimple is a typed 3-address code that makes it easier to perform analyses like use-definition chains than the stack code of *bytecode*. Weaving is performed in *abc* with Jimple representations of classes and interfaces.

1.2 The Compilation Process

The compilation process for AspectJ programs differs from ordinary Java compilation in that crosscutting behavior defined in aspects must be combined to classes and interfaces. This process is called *weaving*, and is usually done at binary-code level. The *ajc* compiler performs weaving at *bytecode* using BCEL, and *abc* uses one of the Soot framework's intermediate representation for this, called Jimple.

In both compilers, Java and AspectJ source code is transformed into ASTs and then intermediary representations of the binary code. On *ajc*, *bytecode* is generated and manipulated directly via in-memory representations of its structure using the BCEL framework; on *abc*, Jimple code is used. The front-end is also responsible for generating crosscutting information for the weaver. This structure identifies locations on classes and aspects where advice must be woven into. The advice weaver then applies advice to join points, producing the final woven code for AspectJ programs.

An advice is transformed into regular a Java method, and the weaving phase applies calls to this method at its join points. For instance, the weaving of a before advice includes a call to the advice implementation before its join points, leaving the join points themselves unmodified. Weaving of around advice is more complex, however, as the original join points must be replaced by calls to advice implementations. This stage gives rise to problems with repeated code generation, and is discussed in detail in Section 2.

2 Around Advice Weaving

The most powerful type of advice defined in AspectJ is the around advice. It can be used to simulate the behavior of both before and after advice, as well as to modify or completely avoid join points. Context used in the shadow may be captured in the advice, but must also be passed on to shadow execution. The power of modifying the behavior of join points − also called shadows − inside around advice comes from the *proceed* command, which activates the shadow captured by the executing advice: context variables captured by the advice may be modified before the *proceed* call. Avoiding shadow execution altogether is achieved by omitting this command.

Listing 1 shows a small AspectJ program. Line 7 contains a shadow of the around advice defined in lines 21 - 23. This advice has no effect on the semantics of its join points, since it simply proceeds to shadow execution.

The remainder of this section presents the weaving techniques applied in the compilation of this program by the *ajc* and *abc* compilers.

2.1 The *ajc* Approach

Since around advice shadows must be executed as a result of *proceed* calls, these instructions are extracted from their original locations to separate methods, called *shadow methods*. The proceed call inside around advice bodies is then replaced by calls to these methods. The process and techniques for around advice weaving in the *ajc* compiler is briefly described in [Hilsdale and Hugunin, 2004].

Figure 1 is a visual representation of this process. The darkened boxes in this figure represent the shadow in line 7 of Listing 1. Notice that only the parts

```
public class Circle {
    private int radius;
    private int x,y;
    public Circle(int x, int y, int radius) {
        setX(x);
        setY(y);
        setRadius(radius);
    }
    public int  getRadius()           { return radius; }
    public void setRadius(int radius) { this.radius = radius; }
    public int  getX()                { return x; }
    public int  getY()                { return y; }
    public void setX(int x)           { this.x = x; }
    public void setY(int y)           { this.y = y; }

    public static void main(String[] args) {
        Circle c = new Circle(0,0,10);
    }
}
public aspect RadiusCheckAspect {
    void around(): call(void Circle.setRadius(int)) {
        proceed();
    }
}
```

Listing 1: The running example for this paper.

affected by weaving are shown in this figure. The result of weaving the around advice defined in `RadiusCheckAspect` into class `Circle` is a modified version of this class. Each around advice shadow in a given class is extracted into its own method. For each shadow method, an inlined implementation of the advice is generated, whose proceed call is replaced with a call to the shadow method.

Method `shadow1` in the woven `Circle` class shown in Figure 1 contains a shadow, which is a call to method `setRadius`. The instance of `Circle` and the argument to this method, which are context variables required for executing this shadow, are passed from the join point to the advice implementation and then on to the shadow. Context passing can be seen in Figure 1 as the target object and the arguments from the shadow's `setRadius` call are passed as arguments to the advice and shadow methods.

The *proceed* call may appear inside nested types in the advice body. In this scenario, this call may attempt to access local variables in the advice environment after its scope has ended. A different approach is used to handle this special case, which involves creating an object to store both the advice environment variables and the shadow code to be executed at the *proceed* call.

Objects used to implement this type of around advice application are called *closure objects*. These objects are implementations of an interface called `AroundClosure`, which defines a method `run` to contain the shadow code. Environ-

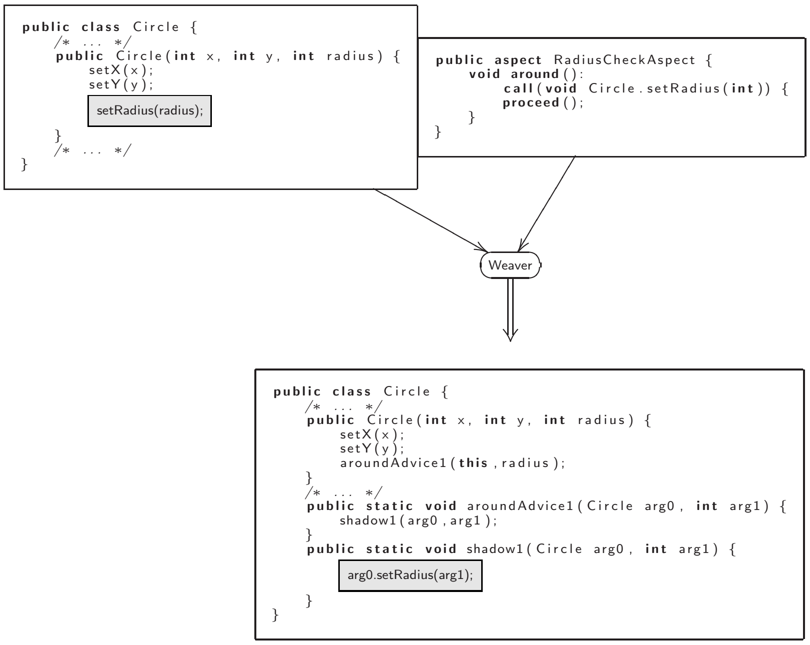

Figure 1: Around advice weaving in *ajc*.

ment variables are placed on the shadow environment as arguments to its closure's `run` method. Thus for each advice application at runtime an object must be created to cope with the *proceed* call.

2.2 The *abc* Approach

Kuzins details, in [Kuzins, 2004], the structure used to implement around advice weaving in the *abc* compiler, and presents benchmarks suggesting that the code produced for around advice in *abc* is faster than the one produced by *ajc*. The performance gain is related to avoiding closure object creation, which is required in *ajc* for around advice that contains proceed calls inside nested types in the advice body.

In the *abc* approach, each shadow is labeled with an integer identifier, called *shadowID*, and the class that contains it is also labeled with an identifier called *classID*. All shadows for an around advice in a given class C are extracted to a single shadow method introduced into C. The shadow method for class C contains all its shadows, and execution is routed to each one via the *shadowID*. This identifier is a parameter to the shadow method and is set by every inlined advice implementation at the *proceed* call. Each advice implementation sets the *shadowID* according to its shadow.

The *classID*-*shadowID* pair appears on code generated by previous versions of the *abc* compiler. On version 1.1.0, however, this approach has been taken a step further, avoiding shadow selection at runtime. This is achieved by inlining advice methods and shadows for each one of the advice's applications.

When closures are necessary to implement an around advice a, *abc* makes class C that contains shadows of a implement an interface called `AroundClosure`. This interface defines a method to which shadows are extracted. The advantage of this approach, when compared to the one adopted by *ajc*, is that the class containing a shadow is itself a closure object, and thus there is no need to create a new object at advice applications. This weaving strategy introduces fields in class C that are linked at runtime, during preparation for the advice call, to environment variables required for advice and shadow execution.

Code woven for around advice by *abc* is similar to that produced by the algorithm applied by *ajc*, except that advice methods are created in the *bytecode* class that represents the aspects in which they were declared, rather than the class where their shadows appear.

3 Repeated Advice Implementations

Repeated advice implementations are generated during around advice weaving when a class C contains several identical shadows of an around advice a. If a class contains n identical shadows of any given around advice, the around weaving strategy described in Section 2 create n identical pairs of advice and shadow implementations. This generation of repeated advice implementations appears on code compiled by both *ajc* and *abc*. It is due to the naive generation of inlined around advice implementations, with no regards as to whether or not other identical implementations have already been generated for identical shadows in the same class.

Consider modifying the code base presented in Listing 1 to add to the `main` method defined in class `Circle` a call to `setRadius`. Listing 2 shows the resulting `main` method. This creates another shadow of the around advice defined in Listing 1, thus producing repeated advice implementations.

```
public static void main(String[] args) {
    Circle c = new Circle(0,0,10);
    c.setRadius(-1);
}
```

Listing 2: The `main` method, modified to contain an advice shadow.

Class `Circle` has now two shadows of the existing around advice: one in its constructor, and another in the `main` method. The woven code for class `Circle`, originally shown in Figure 1, now contains another advice implementation, and can be seen in Listing 3. Notice, however, that the single difference between the advice implementations `aroundAdvice1` and `aroundAdvice2` is the shadow method called. Since these shadows are equivalent, it can be said that the advice implementations are also equivalent, and thus redundant.

Two methods are said to be equivalent if their signatures (parameters and return types) and instruction lists are the same. Eliminating any of these advice implementations and replacing the call to it with a call to the other one doesn't modify the semantics of this program. This reduces the size of generated code for AspectJ programs that use around advice. The optimized code for the example in Listing 3 would be free of methods `aroundAdvice2` and `shadow2`, and the call to `aroundAdvice2` in line 10 would be replaced with a call to `aroundAdvice1`. Notice that the resulting code is smaller, but still semantically equivalent to the original.

This optimization can be performed in two different approaches: a post-weaving *unification* phase, or advice implementation *caching* during the weaving process. At post-weaving, one must identify repeated advice implementations and eliminate all but one of them, and fix the calls to removed implementations. During weaving, one is required to detect that a given shadow has already been woven into in a class, and reuse the advice implementation created for that shadow instead of generating another inlined implementation.

```
public class Circle {
    /* ... */
    public Circle(int x, int y, int radius) {
        setX(x);
        setY(y);
        aroundAdvice1(this, radius);
    }
    public static void main(String[] args) {
        Circle c = new Circle(0,0,10);
        aroundAdvice2(this, -1);
    }
    public static void aroundAdvice1(Circle arg0, int arg1) {
        shadow1(arg0, arg1);
    }
    public static void shadow1(Circle arg0, int arg1) {
        arg0.setRadius(arg1);
    }
    public static void aroundAdvice2(Circle arg0, int arg1) {
        shadow2(arg0, arg1);
    }
    public static void shadow2(Circle arg0, int arg1) {
        arg0.setRadius(arg1);
    }
}
```

Listing 3: Class `Circle` after the weaving of two shadows.

The second approach, caching generated advice implementations during weaving, is more fitting for integration to the existing AspectJ compilers, since it avoids unnecessary work. Repeated advice implementations are never generated, and so they need not be removed. This approach has been suggested to *ajc* developers as a bug report [Cordeiro, 2006a].

The Soot optimization framework defines a phase model in which *bytecode* is modified gradually. In the *abc* compiler, only the peephole and flow analysis phases are activated. However, in these phases, one isn't able to modify the structure of the optimized program, and thus optimizations are restricting to handling method bodies. Since eliminating repeated advice implementations requires eliminating methods from classes as well as modifying method bodies, it is not possible to implement this optimziation as an *abc* back-end extension. During development of this study the developers of *abc* have implemented this solution by modifying the compiler's around weaving algorithm, as suggested to the *ajc* developers, integrating reuse of advice implementations in the weaving process.

3.1 Results

Removing advice and shadow implementation replicas from the code generated for an AspectJ program produces smaller code by eliminating from it several structures required to represent these methods in *bytecode* format. This decrease in code size is proportional to the number of around advice applications in each of the program's classes, as well as the size of advice and shadow bodies.

Table 1 shows the sizes of a set of AspectJ programs that use around advice. *Singleton* is the test program that accompanies Hannemann's Singleton pattern implementation [Hannemann and Kiczales, 2002]. Its main method contains three identical shadows of an around advice.

SpaceWar is a sample AspectJ programs that features several language constructs and idioms. It is available along with the Eclipse AspectJ Development Tools[2] (AJDT). The around advice used in this program captures user and computer commands given to ships, ensuring that their respective ship is alive at the time the command is issued.

Laddad presents, in [Laddad, 2003], a thread-safety aspect that can be applied to programs written using the Swing library. This aspect has been applied to the *Rin'G* program [Cordeiro et al., 2004], which is mostly based on user interaction and thus makes great use of Swing classes.

The reduction in *bytecode* size achieved by eliminating repeated advice and shadow implementations is shown in this table, as the optimized programs are smaller than the original ones. Decrease in *bytecode* size is proportional to the number of around advice applications in the program. The decrease percentage for a given program also depends on its total size: for instance, the decrease percentage for the *SpaceWar* program is smaller than for *Singleton*, since the latter is actually much smaller.

The greatest reduction presented in Table 1 is for the *Rin'G* program. Since this program is user-interface-oriented, there are roughly 500 around advice shadows spread over 83 classes, thus making the program size / reduction size ratio more noticeable.

[2] `http://www.eclipse.org/ajdt`

Application	Original Code – A (*bytes*)	Optimized – B (*bytes*)	Decrease (%)
Singleton			
abc	8115	7539	**7.1**
ajc	17403	16667	**4.2**
SpaceWar			
abc	150869	145391	**3.9**
ajc	222446	215995	**2.9**
Rin'G			
abc	947179	805162	**15**
ajc	1212273	1001661	**17.4**

Table 1: Code generated for AspectJ programs by original and optimized compilers.

4 Repeated Context Variables

Advice in AspectJ can capture context from join points, via the `args`, `target` or `this` clauses. Context information gathered by these clauses comprises arguments and targets of method calls and member variable operations, as well as the executing object at the join points. Once captured, these variables are made available to the advice body. In around advice, captured context variables must be passed on to shadows in the *proceed* call.

However, even if the programmer doesn't capture context variables explicitly in pointcut expressions, the shadow's environment must be kept after it has been extracted to a shadow method during weaving. This is done by passing context as arguments from the original join point environment to the advice method, and then on to the shadow method, as can be seen in the woven code of Figure 1.

If the programmer uses the context capture clauses, there is always an intersection between this explicitly captured context and the set of variables required for shadow extraction. Therefore, whenever an around advice uses context capture clauses in its definition, redundant parameters are introduced in its implementations' signatures.

Context variable repetition leads to three problems in generated *bytecode* for around advice:

– redundant parameters add to the size of method definitions in *bytecode*, thus resulting in larger code;

– memory consumption is larger than necessary, since activations of advice methods in the execution stack allocate local variables for redundant parameters;

– execution time is wasted loading redundant arguments to advice method calls.

Consider replacing the around advice from Listing 1 with the one in Listing 4, which captures the argument of calls to `setRadius`. The woven `Circle` class after this modification is shown in Listing 5. Notice that the advice implementation

```
public aspect RadiusCheckAspect {
    void around(int r):
            call(void Circle.setRadius(int)) && args(r) {
        proceed(r < 0 ? 0 : r);
    }
}
```

Listing 4: Around advice capturing and modifying context from its join points.

```
public class Circle {
    /* ... */
    public Circle(int x, int y, int radius) {
        setX(x);
        setY(y);
        aroundAdvice1(this, radius, radius);
    }
    public static void aroundAdvice1(Circle arg0, int arg1,
            int arg2) {
        if (arg1 < 0)
            shadow1(arg0,0);
        else
            shadow1(arg0,arg1);
    }
    public static void shadow1(Circle arg0, int arg1) {
        arg0.setRadius(arg1);
    }
```

Listing 5: Class `Circle` after the weaving with context passing.

contains an unused parameter, and the same local variable is used at the join point as an argument for both repeated parameters.

Capturing environment variables required for shadow execution is part of the shadow extraction process presented in Section 2. A corresponding parameter is added to the advice implementation's signature for each one of these variables in this step. While the advice is being inlined, variables explicitly captured by the programmer are also added as parameters to the advice implementation. Failure to detect the intersection between the sets of variables captured in these two separate steps leads to redundant parameters in advice implementations.

This problem can be fixed by keeping a record of captured local variables during shadow extraction, so that they won't be captured a second time while inlining the advice method. This solution has been suggested to both *ajc* and *abc* developers, and its implementation is currently being discussed [Cordeiro, 2006b, Cordeiro, 2006c].

Table 2 shows reduction in code size as a result of eliminating repeated context variables. *Production Line*, which serves as an example for this optimization, is a dynamic programming solution to the problem proposed in [Cormen et al., 2002, Chap. 15]; there are two production lines with equal sequences of machines that perform the same job, but at different latencies. The problem is to find the minimum time required to go through the production line, considering that artifacts produces by a machine in one line may be transfered to the other line at a time cost. In this case, however, AspectJ was used to implement transparent memoization in the recursive solution to the problem.

Eliminating parameters from advice implementations, at *bytecode* level, removes structures used to describe the type of these parameters and instructions to load them. When compared with total program size, this reduction is small, as can be seen in Table 2.

The entry in Table 2 for the code generated by *abc* for the *Production Line* program shows that the decrease percentage is ten times larger than for the *ajc* version. This is due to a collateral effect of eliminating repeated context variables, in which *abc* is able to eliminate repeated advice and shadow implementations. After weaving, the inliner attempts to identify replicas among the generated methods. Since this is done on Jimple code, there are local variables that bind context variables to advice parameters, which makes each inlined implementation different. Once context variables are eliminated, these local variables are also removed from the Jimple code, and the inliner manages to identify that the advice implementations are equivalent[3].

Application	Original Code (*bytes*)	Optimized (*bytes*)	Decrease (%)
Production Line			
ajc	14693	14568	**0.90**
abc	7481	6802	**9.00**
SpaceWar			
ajc	222446	222310	**0.06**
abc	150881	150746	**0.09**

Table 2: Code sizes for original and optimized AspectJ programs.

Table 3 shows average measures for the running time of calls to a few methods in the *SpaceWar* program. These methods are captured by an around advice, and thus measuring their execution time shows the effect in advice activation time of eliminating parameters from advice implementation signatures. The average times shown here were collected from a sample of 33 executions of each method for diminishing the impact of OS scheduling and other external runtime interferences.

[3] Equivalence between methods is determined in *abc* by string representations of Jimple code, rather than their semantics.

Method	Original (ms)	Optimized (ms)	Decrease (%)
ajc			
`fire`	2.691	2.595	**3.57**
`rotate`	0.0117	0.0113	**3.42**
`thrust`	0.0125	0.0114	**8.80**
abc			
`fire`	2.358	2.291	**2.84**
`rotate`	0.0107	0.0104	**2.80**
`thrust`	0.0138	0.0120	**13.04**

Table 3: Average execution times for methods affected by an around advice in the *SpaceWar* program.

Though the impact in execution time of a single advice call is almost negligible, as shown in Table 3, the impact in the running time of programs with a great number of advice calls at runtime can add up to be quite large. This is especially the case when around advice applies to recursive methods, as in the *Production Line* algorithm; the running time of this program for random production lines of different sizes is shown in Table 4.

Another important consequence of eliminating repeated context variables from around advice is a decrease in memory consumption. Parameters are stored as local variables in *frames* by the JVM for each method activation. Thus removing some parameters from an advice implementation makes its activation frames smaller, which allows programs with around advice applied to recursive methods to run for larger inputs. This is shown in the last four lines of Table 4: the *bytecode* compiled by the original *ajc* runs for production lines of up to 967 machines, while the optimized version runs for inputs of up to 1017 machines – about 5% larger. The same happens with the code compiled by *abc*, with the optimized version running for inputs of about 8% more machines than the original.

5 Conclusions

This paper presented a study of around advice weaving techniques applied by two AspectJ compilers: the AspectJ Compiler, *ajc*, and the AspectBench Compiler, *abc*. Code repetition problems have been identified in these techniques.

Repeated advice and shadow implementations appear in *bytecode* generated by *ajc* and *abc* when a single class contains several identical shadows of an around advice. By eliminating advice and shadow implementation replicas, this optimization decreases the *bytecode* size for AspectJ programs. During development of this work the developers of *abc* have also implemented this optimization in their weaver.

While capturing context variables for around advice implementations, some local variables are captured more than once, producing repeated context variables in advice implementations. This problem appears when context variables are explicitly captured by the programmer in pointcut expressions, by means of the `args`, `target` and `this` clauses. Solutions to this problem have been experimentally integrated into the *abc* and *ajc* compilers, and operate in the weaving process. Once repeated variables have been eliminated, the resulting code

Input size (machines)	Original (ms)	Optimized (ms)	Decrease (%)
100			
ajc	4.856	2.508	**48.35**
abc	4.562	2.388	**47.65**
500			
ajc	6.397	3.991	**37.61**
abc	5.724	3.411	**40.41**
900			
ajc	9.248	6.372	**31.10**
abc	6.045	3.622	**40.08**
967			
ajc	9.363	6.399	**31.66**
abc	7.529	4.844	**35.66**
1017			
ajc	–	6.472	–
abc	7.752	5.101	**34.20**
1230			
ajc	–	–	–
abc	7.083	4.860	**31.39**
1341			
ajc	–	–	–
abc	–	5.089	–

Table 4: Average execution times of the *Production Line* program for random inputs.

is smaller, uses less memory and runs faster. Memory consumption and time reductions are more relevant in programs that have around advice applied to recursive methods, where several advice activation frames coexist in the execution stack.

Code generated by the *abc* compiler shows clearly that the AspectJ language constructs are not inherently expensive, but rather implemented in an expensive way in the *ajc* compiler. These constructs can, in fact, be implemented efficiently, as in the *abc* compiler, though this is not the priority for *ajc* developers. Efforts in *ajc* development have been concentrated on compilation and weaving speed, as well as the introduction of load time weaving for the AspectJ language.

The main contribution of this paper is the identification of two problems caused by around advice weaving in AspectJ compilers. Solutions to these problems have been proposed to the developers of these compilers and are currently under discussion [Cordeiro, 2006a, Cordeiro, 2006b, Cordeiro, 2006c]. A formal evaluation of the proposed optimizations remains as a future work at the time of writing.

Though the AspectJ language is currently used in software development in production environments, this study shows that small optimizations may still improve the performance of programs written in this language, which indicates that the compilation techniques for aspect oriented programs are still in a stage of continuous evolution.

References

[Aspect Bench Compiler Team, 2006] Aspect Bench Compiler Team (2006). Official *abc* project page. http://www.aspectbench.org. Last visited in December 2006.

[AspectJ Team, 2006] AspectJ Team (2006). Official page for the AspectJ project and *ajc* compiler. http://www.eclipse.org/aspectj. Last visited in December 2006.

[Avgustinov et al., 2004] Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. (2004). Building the abc AspectJ compiler with Polyglot and Soot. Technical Report abc-2004-2, The abc Group.

[Avgustinov et al., 2005] Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. (2005). Optimising AspectJ. *PLDI'05*.

[Cordeiro et al., 2004] Cordeiro, E., Stefani, I., Soares, T., and Tirelo, F. (2004). Rin'g: Um ambiente não-intrusivo para animação de algoritmos em grafos. In *XII WEI, em Anais do SBC 2004 - XXIV Congresso da Sociedade Brasileira de Computação*, volume 1.

[Cordeiro, 2006a] Cordeiro, E. S. (2006a). Around advice weaving generates repeated methods. *Bug report* available at `https://bugs.eclipse.org/bugs/show_bug.cgi?id=154253`.

[Cordeiro, 2006b] Cordeiro, E. S. (2006b). Around weaving produces repeated context variables. *Bug report* available at `https://bugs.eclipse.org/bugs/show_bug.cgi?id=166064`.

[Cordeiro, 2006c] Cordeiro, E. S. (2006c). Around weaving produces repeated context variables. *Bug report* available at `http://abc.comlab.ox.ac.uk/cgi-bin/bugzilla/show_bug.cgi?id=77`.

[Cormen et al., 2002] Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2002). *Algoritmos: Teoria e Prática*. Editora Campus.

[Dahm et al., 2003] Dahm, M., van Zyl, J., and Haase, E. (2003). Official BCEL Project Page. http://jakarta.apache.org/bcel. Last visited in December 2006.

[Dufour et al., 2004] Dufour, B., Goard, C., Hendren, L., et al. (2004). Measuring the Dynamic Behaviour of AspectJ Programs. *OOPSLA'04*.

[Hannemann and Kiczales, 2002] Hannemann, J. and Kiczales, G. (2002). Design pattern implementation in java and aspectj. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173, New York, NY, USA. ACM Press.

[Hilsdale and Hugunin, 2004] Hilsdale, E. and Hugunin, J. (2004). Advice Weaving in AspectJ. *AOSD'04*.

[Kuzins, 2004] Kuzins, S. (2004). Efficient Implementation of Around-advice for the AspectBench Compiler. Master's thesis, Oxford University.

[Laddad, 2003] Laddad, R. (2003). *AspectJ in Action*. Manning Publications Co.

[Lindholm and Yellin, 1999] Lindholm, T. and Yellin, F. (1999). *The Java Virtual Machine Specification*. Addison-Wesley Professional, second edition. Available at `http://java.sun.com/docs/books/vmspec/index.html`.

[Myers, 2006] Myers, A. (2006). Official polyglot project page. `http://www.cs.cornell.edu/Projects/polyglot`. Last visited in December 2006.

[Vallée-Rai et al., 1999] Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., and Co, P. (1999). Soot - a Java Optimization Framework. In *Proceedings of CASCON 1999*, pages 125–135.