Dynamic AOP with PROSE

Angela Nicoară Gustavo Alonso

Department of Computer Science
Swiss Federal Institute of Technology Zürich (ETH Zürich)
CH-8092 Zürich, Switzerland
{anicoara,alonso}@inf.ethz.ch

Abstract. Dynamic Aspect-Oriented Programming (d-AOP) is an important tool to implement adaptation in a wide variety of applications. In particular, large distributed infrastructures, middleware, and pervasive computing environments can greatly benefit from d-AOP to adapt software systems at run time. In this paper, we discuss the design of PROSE, an open source, generic platform for software adaptation. The paper discusses the join-points needed in practical systems together with examples of how they can be used, different implementation strategies and their practical implications. It also describes a highly efficient mechanism to implement stub and advice weaving.

1 Introduction

Aspect oriented programming (AOP) [1] is a technique that allows the expression of orthogonal concerns in an application. Dynamic AOP (d-AOP) extends the original notion of AOP by allowing weaving at *load* or *run time*. Dynamic AOP is intended for problems that are quite different from those addressed by compile time AOP. In particular, dynamic AOP has been shown to be a very suitable mechanism for run time adaptation of applications and services [2–5].

In this paper we present a new version of PROSE, a middleware platform for dynamic adaptation. While previous versions of PROSE explored the issue of dynamic AOP (interception through the *Java Virtual Machine Debugger Interface (JVMDI)* [6], and *JIT based weaving* [7]), in this new version we provide a complete and flexible adaptation platform. The platform is flexible in that it supports different forms of weaving: *stub* weaving and *advice* weaving. The two of them are combined to give designers the ability to fine tune the trade-off between performance and flexibility in the adaptation.

The rest of the paper is structured as follows: Section 2 describes PROSE. Section 3 describes the design and the implementation of the new version of PROSE, including the new weaving method. In Section 4 we present the performance evaluation and comparison with other systems. Section 5 discusses related work and Section 6 concludes the paper.

The work presented in this paper was supported (in part) by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

2 The PROSE System

PROSE is an open source (http://prose.ethz.ch), adaptive middleware platform. PROSE is available in two versions: one based on the JIKES IBM Research Virtual Machine (RVM) [8] and another one based on the SUN JVM. In this paper, we discuss the version based on JIKES RVM.

2.1 Flexibility in PROSE

PROSE was one of the first platforms that tackled the problem of dynamic AOP. As a result, PROSE has evolved through a number of versions, each one of them based on different forms of interception and weaving. *The first version of PROSE* was intended to demonstrate the potential of dynamic AOP. It used the *Java Virtual Machine Debugger Interface (JVMDI)* to convert join-points into stop points. Once the application had been stopped, the advice was executed externally to the application although the advice had access to the context where it was being executed (e.g., stack frames, calling parameters for methods, etc.) [6]. *The second version of PROSE* extended this model by giving the option of, instead of using the debugger, using the *baseline JIT compiler*. The idea was to weave *hooks* into the application at *native code locations* that correspond to *all* potential join-points. When executed, the hooks determine whether an advice needed to be invoked for that particular join-point and called the advice [7].

In what follows, we describe *the new version of PROSE* where hook weaving has been replaced with (a) an *advice weaving* strategy for method replacement join-points and (b) a *stub weaving* mechanism for join-points involving external advices. An important characteristic of PROSE is that these different weaving mechanisms are meant as alternatives. The idea is to mirror the flexibility offered by middleware platforms in which designers can choose different approaches depending on the type of application involved. Thus, the new version of PROSE presented in this paper supports JVMDI based weaving, hook weaving, stub weaving, and advice weaving.

2.2 PROSE aspect language

By design [9], PROSE does not define a new aspect language. The aspect language is Java. As an example of how this is done, Fig. 1 depicts a simple PROSE aspect which redefines the original version of a method with a new one.

```
1  public class ExampleAspect extends DefaultAspect {
2     public Crosscut doRedef = new MethodRedefineCut() {
3        public void METHOD_ARGS (Foo ob, int x) {
4           // the new method code
5        }
6        protected PointCutter pointCutter() {
7           return Within.method("bar");
8        }
9     };
10 }
```

Fig. 1. Example of a PROSE aspect

As the figure shows, all aspects in PROSE extend the `DefaultAspect` base class (line 1). An aspect may contain one or more crosscut objects[1]. A crosscut object defines an advice method called `METHOD_ARGS` (line 3) and a pointcut method (line 6) which defines a set of join-points where the advice should be executed. In Fig. 1, there is just one crosscut, corresponding to the `doRedef` instance field (line 2). This crosscut type is called `MethodRedefineCut` (line 2), similar to the `around` advice construct in AspectJ [10]. The aspect showed in Fig. 1 tells PROSE to replace the method `bar` (line 7) from the `Foo` class (line 3) with the code specified in the advice (line 4).

Using Java as the aspect language has the advantage that any competent Java programmer can use PROSE in a very short time. Because of the layered architecture of the system, this does not prevent the use of other aspect languages. The only requisite is to develop the corresponding aspect management module (called the *AOP engine*, see below).

2.3 Join-points in PROSE

In order to provide support for a wide range of adaptive middleware scenarios, PROSE currently supports the following set of join-points:

Method boundaries: encompassing *method entry* and *exit*. These join-points enable adaptations that extend an existing application by adding new functionality. Examples are encryption of messages before they are being sent and adding transactional controls to method calls [11].

Method redefine: replaces method bodies. Method redefine is used for adaptations that involve changes to the current behavior. Examples involve replacement of communication protocols, temporal offloading of functionality by replacing the method with a call to another device or a server [12], and testing and instrumentation.

Field access and modification: track access to variables. These joint-points are very useful in, e.g., adaptations that involve orthogonal persistence [11] and in correcting errors in measurements taken by sensors. They can also be used to implement a dynamic form of shared memory among mobile devices and to control potential race conditions.

Exception join-points: including *exception catch* and *throw*. These join-points are essential to be able to deal with real applications. For instance, adding transactional boundaries to existing methods require to deal as well with the exceptions those methods might raise [11]. They are also important in correcting anomalous situations in small devices and mobile computing settings.

2.4 PROSE architecture

The architecture is divided into two layers: the *AOP engine* layer and the *execution monitor* layer (Fig. 2). The *AOP engine* acts as the middleware platform

[1] The *crosscut object* construct in PROSE corresponds to a *pointcut* and an associated advice in AspectJ terminology.

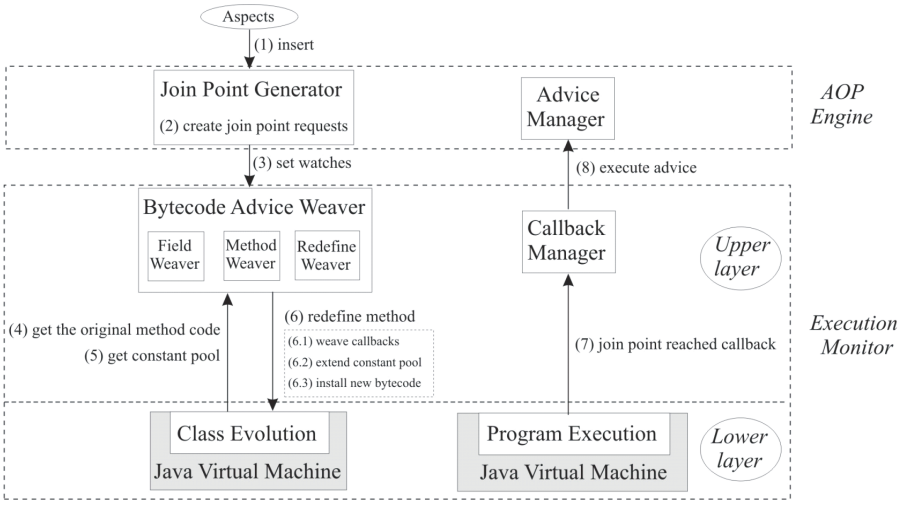

Fig. 2. PROSE architecture

in PROSE. It is in charge of the management of aspects and of the execution of the advices. The execution monitor deals with weaving and resides in the JVM. The two layers are independent of each other and interact through a well defined API. The idea is that the AOP engine can be replaced as needed depending on the application at hand.

The general procedure for aspect insertions is as follows (Fig. 2). The AOP engine accepts aspects (1) and transforms them into join-point requests (2). This module can be changed to accept aspects described in a language other than Java. After the information on join-points has been extracted, the AOP engine activates the join-point requests by using the API of the execution monitor (3).

The *execution monitor* is divided into two layers. The lower layer extends the Jikes RVM by adding support for method code replacement at run time. The upper layer accepts weaving requests from the AOP engine (3), gets the original bytecode of each method (4) and the constant pool bytecode of the classes which actually contains the methods (5), instruments the affected methods (by adding advice callbacks at the corresponding bytecode locations (6.1), extends the constant pools of all affected classes (6.2)) and installs the instrumented methods in the virtual machine, using the services offered by the `Class Evolution` module (6.3). The bytecode manipulations and the methods instrumentation (4 - 6.3) are performed by the `Bytecode Advice Weaver` module. This module contains three main classes: `FieldWeaver`, `MethodWeaver` and `RedefineWeaver`. The main class that handles advice weaving at bytecode level is `MethodWeaver`. Field access and modification requests are handled by the `FieldWeaver` class, whereas method redefine requests are handled by the `RedefineWeaver` class. For advices that are executed externally, when the program execution reaches one of the activated join-points (7), the execution monitor notifies the AOP engine which then executes the corresponding advice (8) (or checks whether the advice needs to be executed, e.g., by checking the current time if it is a time dependent

advice). When aspects are removed, the join-points are deactivated, the weaver unweaves any advice whose aspects were removed, and the original bytecode of each method is installed.

2.5 The Execution Monitor

The execution monitor is in charge on weaving. In the new version of PROSE, weaving takes place by insertion of stubs or by direct advice weaving. In both cases, the weaving procedure is based on code replacement. Whether what is woven is a stub or the entire advice, this is done by working on the original bytecode which is augmented with stubs or entirely overwritten. Then the new code is inserted into the JVM. When the execution reaches the modified code, the JVM is forced to JIT the new code before it is executed. In this way, what is executed is not the old code but the new code containing the stub or the method being redefined. We illustrate the process using method redefinition (other join-point types are discussed later). Fig. 3 contains the core interface of the execution monitor to change a class at run time.

When a new method is redefined, the execution monitor replaces the original bytecode of a method with the new bytecode, using the `redefineMethod` method of the `VM_ClassEvolution` class. The new method becomes active only when the `commit` method is invoked. It is possible to redefine several methods before committing them. The `commit` method should be called after instrumenting all required methods.

The bytecode of a method is obtained by the `getMethodCode` method. More exactly, the bytecode that represent the `method_info` structure defined in the Java Virtual Machine Specification [13] are returned. Jikes RVM doesn't store the bytecode of a method when loading the class containing this method. There is a field called `VM_Method.bytecode` but this only represents the `Code` attribute which corresponds to the body of a method (a part of the `method_info` structure). Therefore, in order to implement the `getMethodCode` method, we adapted the `VM_Method.readMethod` method and added an additional field to the `VM_NormalMethod` class which finally contains the whole bytecode of a method.

```
1  public final class VM_ClassEvolution {
2     // get the bytecode of the method "m"
3     public static byte[] getMethodCode(Method m);
4     // replace the bytecode of method "m" with the new bytecode "codes"
5     public static void redefineMethod(Method m, byte[] codes);
6     // get the constant pool bytecode of the class "c"
7     public static byte[] getConstantPoolCode(Class c);
8     // extend the constant pool bytecode of the class "c" with new
      bytecode "codes"
9     public static void extendConstantPool(Class c, byte[] codes);
10    // install the new redefined methods
11    public static void commit();
12 }
```

Fig. 3. The Class Evolution API

The method responsible for retrieving the bytecode that makes up the constant pool of a class is `getConstantPoolCode`. If the constant pool has been extended before with the `extendConstantPool` method, the extended version will be returned even if those changes are not committed. This allows increasingly extending the constant pool of a specific class without committing. In order to implement `getConstantPoolCode` in Jikes RVM, we had to apply some changes to the `VM_Class` class. The constant pool bytecode is processed in the constructor of the `VM_Class` class but is not saved. Therefore, we added another field to the `VM_Class` class which holds the bytecode of the constant pool.

In order to support dynamic bytecode instrumentation in Jikes RVM, two lists are maintained: one for the method redefinitions and one for the constant pool extensions. The constant pool extensions are installed first because the method redefinitions rely on them.

The new method gets activated by calling `VM_Class.updateMethod` on the class object where the method is declared. The static, virtual or interface method table entry is updated (JTOC (Jikes RVM table of content) for static methods, TIB (type information block) for virtual methods, or IMT (interface method table) for interface methods [14]). This entry contains a pointer to the native machine code that will be executed next time when the method is invoked. However, it is not the JIT compiled code of the new method which is used. A lazy compilation stub is used instead that will trigger the JIT compilation when the method is being executed for the first time. With this optimization, methods which are never executed are not compiled, thus saving the overhead that would be needed to compile the method at run time. Support for weaving in the new version of PROSE required minimal changes to the existing Jikes RVM code (90% of the changed code was kept within the `VM_ClassEvolution` class).

2.6 The AOP Engine

The AOP engine is in charge of everything that has to do with aspects and advices except the weaving. The role of the AOP engine is very important to understand how PROSE works and its overall performance. Although several performance comparisons among dynamic AOP system have been made, e.g., [15, 16], several of them do not take into consideration that PROSE has two layers and that it is the AOP engine where most of the performance penalties are paid. The comparisons typically involve the overhead of weaving in other systems vs the overhead of weaving and the AOP engine of PROSE. To understand why this comparison is not meaningful, it is enough to look at the functionality PROSE offers at the AOP engine level. This functionality is also the essence of the adaptive middleware features that PROSE provides.

Since its inception, PROSE supports *atomic weaving*, a critical feature in realistic distributed applications. Atomic weaving corresponds to the activation of join-points matched by an aspect A in one single step. The PROSE engine provides this support as follows. It activates join-points one by one (non-atomically) but sets a flag in the A's advice method that makes its execution a do-nothing operation. As more join-points are activated, A's advice *is* actually invoked, but

it has no visible effect at run time. Once all the join-points corresponding to A's advice have been properly activated in the execution monitor, the engine unsets the flag. From this point on, reaching a join-point matched by A is followed by the execution of the actual advice.

The AOP engine is also responsible for executing the advices that are not directly woven into the application. As explained, the reason is to achieve the necessary degree of indirection to implement additional functionality. Thus, the AOP engine supports a filter pool on the advices that control whether they need to be executed or not according to external events. One such event is the insertion at other JVMs -used to implement atomic weaving. Other events involve, for instance, timing constraints (advices executed only at particular times), context constraints (e.g., advices executed only when other applications are present, advices executed only when the network is overloaded), history constraints (e.g., advice is executed only after having been called X number of times). The use of these filters is what gives PROSE the power to implement a wide range of dynamic adaptations that are very useful in the context of middleware. Yet, they also add certain overhead since the execution of an advice is conditional to those filters and the filters are checked every time a callback is made to an external advice.

3 Weaving at Join-points

This section describes the weaver API and the details concerning the *bytecode instrumentation* for several join-point types supported by PROSE. Our approach weaves advice calls at run time into the bytecode of a method wherever they are needed, but *not* at all potential join-points. When an aspect is inserted, the matching join-points are activated which implies that all affected methods are instrumented with the corresponding advice call. These methods have to be recompiled afterwards by the JIT compiler. Exception join-points (e.g., exception throws and handlers) are treated differently. They are implemented by extending the run time exception handler of Jikes RVM.

3.1 The weaving process

The main class that weaves advice calls at bytecode level is `MethodWeaver`. There is a one-to-one relationship between method weavers and methods. Instances of the class `MethodWeaver` are responsible for exactly one method. Each method has exactly one method weaver object. If no method weaver exists for a certain method then it is created during the first invocation of `getMethodWeaver()`.

All modified method weavers will install their new method bytecode if the static method `commit()` is called (this is part of the API used to support atomic weaving). We iterate over all method weaver objects and check if they are modified. If yes, the weaver will weave callbacks for all activated join-points of this method, using the `weave()` method.

The `commit()` method should be called after instrumenting all required methods. It is possible to add as many callbacks as needed before actually committing

bytecode instrumentation. After all method weavers have installed the new bytecode, the changes are activated in the VM.

All the methods which have been woven with the corresponding callbacks can be restored, using `restoreAll()` method. We iterate over all method weavers and install the original bytecode of each method. Finally, all the changes are committed using the services of `VM_ClassEvolution`.

Bytecode instrumentation [17] is a term used to denote various manipulations of the bytecode, typically performed automatically by tools and libraries according to a relatively high-level specification. The current implementation of PROSE employs the BCEL (Byte Code Engineering Library) [18], a bytecode manipulation library.

3.2 Method entry

Weaving method entry advice is done by adding a stub to the advice before the original bytecode of a method. The stub is woven before the first instruction of the method but after the call has taken place so that the stub can inspect the stack, extract the parameters of the call and pass these parameters to the advice code. The arguments of the called method are accessible from the advice because they are passed as parameters to the advice method.

3.3 Method exit

Weaving a method exit advice is also done through a stub although is more involved than weaving a method entry advice. The main problem is the control flow. For example, consider a method that just returns a value. In that case we could simply weave the method advice before the `return` bytecode instruction. This solution is not correct if the method contains two or more `return` statements or, even worse, if an exception is thrown.

Therefore, we chose another solution: to introduce a `try-finally` construct. The original body of the method is put into the `try` block and in the `finally` block we invoke the method exit advice. According to the Java Language Specification [19] the `finally` block is guaranteed to be executed after the `try` clause. It doesn't matter whether the `try` block finishes successfully or because an exception has been thrown. The Java Virtual Machine Specification [13] presents detailed information about how the `try-finally` construct is translated from Java source to bytecode.

For normal control transfer from the `try` block the compiler makes use of two special instructions: `jsr` ("jump to subroutine") and `ret` ("return from subroutine"). The instructions of the `finally` clause are located in the same method, much like exception handlers. Before each `return` instruction, the returned value (if any) is stored into a local variable, and then a `jsr` to the start of the `finally` instructions is performed. The last `finally` instruction is the `ret` instruction; it fetches the `return` address from the local variable and transfers control to the instruction at the `return` address.

In the case of abrupt control transfer, i.e., when an exception has been thrown in the `try` clause, an exception handler is added. This handler catches instances of the class `Throwable`, thus exceptions of any type. In the `catch` block a `jsr`

instruction does a subroutine call to the code for the `finally` block, similar to normal control transfer. After that, the exception is thrown again.

3.4 Method redefinition

Method redefinitions take place through direct advice weaving. They are performed by the following steps: (1) read the bytecode of the new method, (2) create a new method instance, (3) replace the old method with the new one (in the list of the declared methods in its class, in all subclasses and in the static/virtual/interface method lists) and (4) install and activate the new method. Method redefine requests are handled by the helper class `RedefineWeaver`. Method redefinition takes place if the `setRedefineAdvice()` method is called.

Advice weaving imposes certain restrictions on how the advice that redefines a method can be written. For instance, AspectJ can execute the advice around the join-point. An around construct applied to an AspectJ `execution` join-point will replace the captured method code by the advice, resulting in a method redefinition. AspectJ can nevertheless execute the surrounded code by invoking `proceed()` in the body of the advice. Our implementation doesn't support such functionality. The redefined code is not accessible until the aspect is withdrawn. For the same reasons, references to the current aspect instance are not allowed since the aspect object is not available in the captured methods. This means that the `this` keyword cannot be used in the advice method. Even implicit `this` references are not allowed (no instance field reference and no instance method invocation on the current object). Especially, `getThisJoinPoint()` is not possible in such an advice method. The second limitation is related to Java class-member access protection. Consider that we want to access non-public fields of the `Foo` class in the advice method. In this case the Java compiler will refuse to compile the aspect. However, this problem has been solved using Reflection which permits access to non-public members.

3.5 Field access

Field access advices are woven using a stub. The main problem with field access crosscuts is not the weaving process itself but to get the information about where an actual field access happens. For example, assume that we want to execute an advice before a public field declared in a class is accessed. Potentially, every method in every class loaded into the VM could access this field, which makes the search cumbersome.

One way to solve this problem is to scan all methods and check if they access this field. But this would be very inefficient and time consuming. Therefore, we chose a different solution: When a class is loaded into the VM, for each method declared in this class we track every field access. These accessors (set of methods that access a certain field) must be computed for every field available.

There are two bytecode instructions that deal with field accesses, `getfield` and `getstatic`. The second instruction is used to read the content of a static field, and the first one for normal (non-static) instance fields. We use these instructions to track field accesses.

Field access requests are handled by the `FieldWeaver` class. For each method that references the specified field, the corresponding stub is woven.

3.6 Field modification

Weaving field modification advice can be done in the same way as for field accessses. Instead of a set of accessor methods for each field, we need a set of modifiers. These modifiers can be computed by looking at each `putfield` and `putstatic` instruction. Field modifications requests are handled by the `FieldWeaver` class.

3.7 Exceptions

PROSE supports two exception join-point types: *exception throw* and *exception catch*. With these join-points it is possible to execute advice when an exception is thrown or when it is caught by a `catch` block. Weaving an advice for exception join-points is different from the other join-point types. One possibility is to add bytecode instructions which call an advice at the appropriate join-points. To realize this, one must know every method which throws a certain exception or which declares a certain handler. This leads to the same problem as with the field join-points where we need a mapping from the field to the methods which access or modify this field. Therefore, we chose a different solution. We implement these types of join-points by extending the run time exception handler of Jikes RVM. We adapt the Jikes RVM exception handler by adding callbacks. On the VM side, exception handling can not be done at compile time. It must be done at run time because the handler must walk the stack for a thrown exception until it finds a `catch` block.

In case of exceptions, the Jikes RVM JIT compilers translate each `athrow` instruction into an invocation of the `VM_Runtime.athrow()` method which finally invokes `deliverException()`. Therefore, we get the event of a thrown exception for no additional cost. Exception handling is performed in `deliverException()`. The method invocation stack is walked until an appropriate `catch` block is found. The thread will be terminated if the execution is not caught. If there is a `catch` block, exception handling is delegated to a subclass of `VM_ExceptionDeliverer`, depending on the JIT that compiled the method which contains the `catch` clause (`VM_BaselineExceptionDeliverer` for methods compiled by the baseline compiler, and `VM_OptExceptionDeliverer` for methods compiled by the optimizing compiler).

4 Performance Evaluation

In order to evaluate the performance of the new version of PROSE we have performed different benchmarks. All experiments in this paper were performed on Linux, running on an AMD Athlon MP 1600+ 1.4 GHz, double processor machine with 1 GB RAM. We compare results using the Jikes RVM 2.3.0.1. In our experiments the stub and advice-based weavers use the optimizing compiler as a JIT. Additionally, the inlining suppport is turned off. In case of hook-based weaver, a modified baseline compiler is used as a JIT. We evaluate our approach using two benchmark applications: SPECjvm98 [20] and Java Grande [21].

In our experiment, we compare the original JVM with the JVM containing the execution monitor. In this experiment the execution monitor is not activated.

Table 1. The execution times for the hook weaver and the stub and advice weavers with AOP support for method boundaries, method redefinition, field sets, field gets, and exception handlers

Benchmark	Execution time with AOP support	
	Hook weaving	Stub and advice weaving
SPECjvm98 benchmark suite		
check	1.35 (s)	0.51 (s)
jess	34.36 (s)	5.32 (s)
db	54.23 (s)	25.52 (s)
jack	20.95 (s)	4.67 (s)
javac	37.8 (s)	9.07 (s)
compress	40.82 (s)	12.79 (s)
mpegaudio	33.62 (s)	7.19 (s)

Benchmark	Execution time with AOP support	
	Hook weaving	Stub and advice weaving
Java Grande benchmark suite		
LUFact:Kernel	2.4 (s)	1.22 (s)
Crypt:Kernel	3.35 (s)	2.26 (s)
SOR:Kernel	12.92 (s)	3.23 (s)
SparseMatmult:Kernel	13.69 (s)	7.37 (s)
Series:Kernel	21.37 (s)	19.28 (s)
HeapSort:Kernel	3.62 (s)	1.26 (s)
FFT:Kernel	30.14 (s)	29.81 (s)

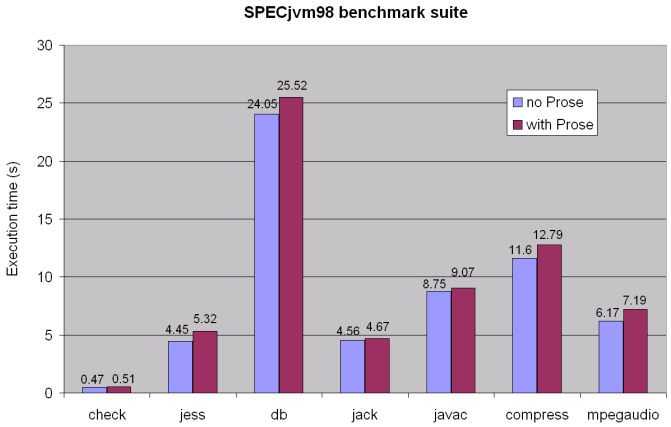

Fig. 4. Relative overhead for stub and advice weavers with AOP support for method boundaries, method redefinition, field sets, field gets, and exception handlers

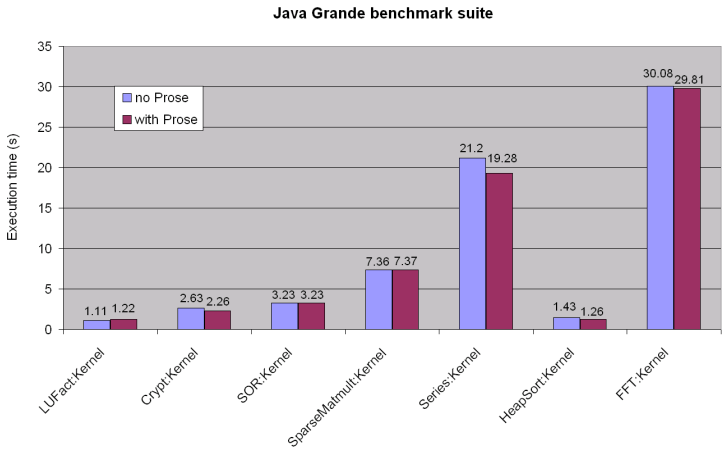

Fig. 5. Relative overhead for stub and advice weavers with AOP support for method boundaries, method redefinition, field sets, field gets, and exception handlers

To measure the performance loss incurred by the existence of the AOP support, on the SPECjvm98 benchmarks, we report the average of the execution times measured for one hundred runs, all run during a single JVM execution, with the size 100 (large) inputs. For the Java Grande benchmark, we report the average times for one hundred runs, each run in a separate VM. Table 1 summarizes the average execution times for each test for the hook weaver and stub and advice weavers. Fig. 4 and Fig. 5 show the relative overhead of the AOP enhanced JVM for the SPECjvm98 and Java Grande benchmarks. The standard deviation for this experiment is less than 7%.

5 Related Work

5.1 Load time approaches

Load time weaving performs the weaving of advices into the original code at the time classes are loaded into a JVM. Examples of such systems are JAC [2], AspectWerkz [3] and JMangler [5]. JAC uses the Javassist bytecode manipulation library to alter the bytecode of a Java object at class load time. AspectWerkz uses a modified classloader to weave the aspects with the base-code instead. It hooks directly into the bootstrap classloader and can then weave aspects to any classes loaded by the preceding classloaders. JMangler modifies the base class of the Java class loader hierarchy, thereby enforcing transformations for classes that are loaded by arbitrary class loaders, except the bootstrap class loader. For adaptation purposes, load time weaving has the disadvantage of breaching the Java security mechanism (e.g., AspectWerkz).

5.2 Run time approaches

Run time weaving is based on a variety of mechanisms that support the insertion of advices on running programs [15].

Similar to the PROSE JVMDI implementation, JAsCo [4] employs the debugger interface of the JVM. JAsCo uses a new language that introduces two additional entities: aspect beans and connectors. An aspect bean contains one or more hooks that describe join-points or pointcuts and the corresponding advice. A connector is used for deploying one or more aspect beans within a concrete component context. JAsCo employs Java HotSwap to change class definitions while the program that contains these classes is running.

Steamloom [16] is an implementation of dynamic join-point support as an extension to IBM's Jikes RVM. The mechanism used in Steamloom for weaving and unweaving of aspects at run time is similar to the one we have presented in this paper. Steamloom only supports method entry and exit join-points.

A dynamic aspect-weaving approach for the .NET platform is presented by Schult and Polze in [22]. Aspects can be woven at instantiation time, i.e. aspects are defined to be active for a particular object at the time when this object is created. The weaving is achieved by dynamically creating a subclass and overriding the crosscutted methods with their woven counterparts. However, this weaving approach is not dynamic in the sense of PROSE. In PROSE, aspects

can be inserted and withdrawn at any time. In the .NET approach, aspects are either active or not during the whole lifetime of an object.

Wool [23] is a dynamic AOP framework that supports two different dynamic weaving strategies. Similar to PROSE, the Wool system employs the JDI to intercept the execution of the base program. Aspects can also be inserted into the target join-points by employing Java HotSwap. Aspects are able to implement their own heuristics for deciding whether they are invasively inserted or not. The classes containing the applicable join-points need to be hotswapped again.

6 Conclusions

In this paper we have presented PROSE, a modular and flexible platform for dynamic adaptation. In addition, we have proposed a mechanism to implement dynamic AOP through method code replacement at run time. The idea is to weave the advices at run time by triggering the recompilation of methods. As part of the recompilation, and depending on the nature of the advice, the advice itself or an efficient callback mechanism are woven into the original bytecode. The system takes advantage of the JIT compiler (when an aspect is inserted, the affected methods are automatically recompiled), but can also use optimizing versions of the compiler, thereby introducing even greater performance gains. PROSE is an open source project and can be downloaded from http://prose.ethz.ch.

7 Acknowledgements

We wish to thank Dr. Andrei Popovici, the initial developer of the PROSE system. This work was possible through the dedicated work of several master and term-project students. Thanks to Johann Gyger, Gerald Linhofer, Philippe Schoch, Kaspar von Gunter, Philipp Sieber, Stephan Markwalder, Gregor Wieser, and Marcel Müller, who contributed to the PROSE project.

References

1. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J.: Aspect oriented programming. In: ECOOP '97, Jyvaskyla, Finland, volume 1241 of LNCS, pages 220-242. Springer-Verlag. (1997)
2. Pawlak, R., Seinturier, L., Duchien, L., Florin, G.: JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, pages 1-24, Kyoto, Japan. (2001)
3. Boner J., Vasseur A.: AspectWerkz website. (http://aspectwerkz.codehaus.org)
4. Vanderperren, W., Suvee, D.: Optimizing JAsCo dynamic AOP through HotSwap and Jutta. In: Proceedings of the 2004 Dynamic Aspects Workshop (DAW04), pages 120-134, Lancaster, England. (2004)
5. Kniesel, G., Constanza, P., Austermann, M.: JMangler - A Framework for Load-Time Transformation of Java Class Files. In: IEEE Workshop on SCAM'01. (2001)
6. Popovici, A., Gross, T., Alonso, G.: Dynamic Weaving for Aspect Oriented Programming. In: 1st International Conference on Aspect-Oriented Software Development, Enschede, The Netherlands. (2002)

7. Popovici, A., Alonso, G., Gross, T.: Just in Time Aspects: Efficient Dynamic Weaving for Java. In: 2nd International Conference on Aspect-Oriented Software Development, Boston, USA. (2003)
8. Alpern, B., Attanasio, D., Barton, J.J., Burke, M.G., P.Cheng, Choi, J.D., Cocchi, A., Fink, S.J., Grove, D., Hind, M., Hummel, S.F., Lieber, D., Litvinov, V., Mergen, M., Ngo, T., Russell, J.R., Sarkar, V., Serrano, M.J., Shepherd, J., Smith, S., Sreedhar, V.C., Srinivasan, H., Whaley, J.: The Jalapeno virtual machine. In: IBM System Journal, 39(1). (2000)
9. A. Popovici: PROSE, a study on Dynamic AOP. Dissertation No. 15176 ETH Zurich (2003)
10. AspectJ website. (http://www.aspectj.org)
11. Popovici, A., Alonso, G., Gross, T.: Spontaneous Container Services. In: ECOOP'03, Darmstadt, Germany. (2003)
12. Popovici, A., Frei, A., Alonso, G.: A proactive middleware platform for mobile computing. In: Proc. of the 4th ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil. (2003)
13. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley, Second Edition. http://java.sun.com/docs/books/vmspec/ (1999)
14. The Jikes Research Virtual Machine User's Guide 2.3.0.1. (http://www-124.ibm.com/developerworks/projects/jikesrvm/)
15. R.Chitchyan, Sommerville, I.: Comparing Dynamic AO Systems. In: Proceedings of the 2004 Dynamic Aspects Workshop (DAW04), pages 120-134, Lancaster, England. (2004)
16. Bockisch, C., Haupt, M., Mezini, M., Ostermann, K.: Virtual Machine Support for Dynamic Join Points. In: AOSD'04, Lancaster, England. (2004)
17. Dmitriev, M.: Application of the HotSwap Technology to Advanced Profiling. In: ECOOP'02 Workshop on Unanticipated Software Evolution. (2002)
18. The Byte Code Engineering Library (BCEL) manual. (http://jakarta.apache.org/bcel/manual.html)
19. Joy, B. and Steele, G. and Gosling, J. and Bracha, G.: The Java Language Specification. Addison-Wesley, Second edition. (http://java.sun.com/docs/books/jls/)
20. Spec - Standard Performance Evaluation Corporation. (http://www.spec.org/osg/jvm98/)
21. Java Grande Forum benchmark suite. (http://www.javagrande.org)
22. Schult, W., Polze, A.: Dynamic Aspect Weaving with .NET. In: Workshop zur Beherrschung nicht-funktionaler Eigenschaften in Betriebssystemen und Verteilten Systemen, TU Berlin, Germany. (2002)
23. Sato, Y., Chiba, S., M.Tatsubori: A selective. Just-in-Time Aspect Weaver. In: GPCE 2003 Proceedings, volume 2830 of LNCS, pages 189-208. Springer. (2003)