Report

Improving the efficiency of adaptive middleware based on dynamic AOP

Author(s):
Nicoar, Angela; Gyger, Johann; Alonso, Gustavo

Publication Date:
2004

Permanent Link:

https://doi.org/10.3929/ethz-a-006744855 →

Improving the efficiency of adaptive middleware based on dynamic AOP

Angela Nicoară, Johann Gyger, Gustavo Alonso

Department of Computer Science
Swiss Federal Institut of Technology Zürich
CH-8092 Zürich, Switzerland
{anicoara, jgyger, alonso} @ inf.ethz.ch

Abstract. Aspect-Oriented Programming (AOP) is used to express modular and orthogonal functionality in software components. The orthogonal functionality is programmed into an aspect, join points are used to describe where in the code the aspect should be inserted. Then the aspect is woven into the original program. Unlike compile time based strategies, dynamic AOP allows aspects to be woven at runtime. This has been shown to be very useful to adapt applications without interrupting service in a wide range of settings (business applications, wireless networks, robotics, etc.), thereby making dynamic AOP a prime candidate for supporting advanced, adaptive middleware platforms. A potential drawback of dynamic AOP is the performance overhead. In this paper, we tackle the performance problem by proposing a mechanism to implement dynamic AOP through method code replacement at runtime. The idea is to use the join points not to trigger the execution of the aspect (or advice) as it is done in most systems but to trigger the recompilation of the original code. The recompiled code contains the advice already woven into it or the callbacks to the corresponding advice, thereby greatly increasing the efficiency of the dynamic AOP process. In the paper we describe the technique, discuss extensive benchmarks to evaluate the performance gain of this approach, and compare the resulting system to other dynamic AOP approaches.

1 Introduction

Aspect oriented programming (AOP) [1] is a technique that allows the expression of orthogonal concerns in an application. These orthogonal concerns affect the source code in potentially many different places and, hence, it is advantageous to express them as aspects that are *weaved* into the source code but are conceptually separated from it. AOP is used in cases where traditional object-oriented techniques such as inheritance are not adequate. The general mechanism in AOP is to describe the functionality to be added as *aspects*. An aspect defines a collection of points in the execution of a program and what to do when these points are reached. The execution points are called *join points* and the action to be executed at the join points is called the *advice*. To help expressing join points in a more concise manner, pattern-like constructs (e.g., regular expressions) are often used, giving rise to the notion of *pointcut*.

Dynamic AOP extends the original notion of AOP by allowing weaving at *load* or *run time*. Although it uses similar mechanisms and concepts, 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 runtime adaptation of applications and services [17,18,19,20]. As such, it has become a very important architecture and mechanism to implement adaptive middleware. To see why, one needs to look in more detail at the weaving process. In conventional AOP, weaving takes place at compile time. That is, a special compiler takes the advices, the pointcuts, the original source code, and then compiles the whole thing into a new executable where the original code has been augmented with the advices. For obvious reasons, the woven aspects cannot be added or removed at runtime and are intended as a software development tool. AspectJ is the best example of a system supporting compile time weaving [7]. Load time weaving performs the weaving of advices into the original code at the time classes are loaded (typically into a JVM). Examples of such systems are JAC [17], AspectWerkz [19] and JMangler [20]. 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). Additionally, the namespace visibility constrains enforced by class loader hierarchies present additional problems when an advice needs to access independently loaded modules.

Runtime weaving, implemented in systems such as PROSE [5,6] and JAsCo [18], is based on a variety of mechanisms that support the insertion of advices on running programs. Typically the insertion is based on a preliminary insertion of a stub that will then make appropriate calls to other parts of the system to insert or execute the advice. In general terms, both load time and runtime weaving are considered dynamic AOP approaches. For the interested reader, a comparative analysis of several Java based dynamic AOP systems has been recently presented in [21].

The key problem for dynamic AOP techniques used in adaptive middleware is the potential overhead that might be inflicted on the application. This is a very challenging issue as it usually comes down to a trade off between flexibility, ability to adapt, and performance. In general terms, the goals are: not to slow down execution of the application (e.g., some dynamic AOP solutions require to run the JVM in debugging mode); minimize the impact when there no advices to be woven (e.g., in some systems, the overhead for pointcuts is paid regardless of whether there is an advice to weave or not); minimize the overhead involved in finding the advice to execute (potentially to be chosen from a large collection); and speed up the dispatch of the advice (i.e., the time need to resolve and call the advice that has to be executed when an active join point is reached). In this paper, we tackle these performance problems by proposing a novel mechanism to implement dynamic AOP. The idea, called *method code replacement*, is to weave the advices at runtime by triggering the recompilation of methods. That is, a pointcut affecting a method will trigger the recompilation of that method. 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. For this purpose we use the Just-in-Time compiler of an IBM Jikes Research Virtual Machine [2] and the mechanisms present in the JVM for just-in-time compilation of code. As with other dynamic AOP systems, we can weave and unweave advices at runtime to any application running on the JVM. The big advantage of the new approach is that it is far more efficient than existing solutions. In the paper we describe the technique in detail and perform extensive benchmarks to evaluate its performance. The micro-measurements show a clear performance improvement over *stub-weaving* approaches (a very common technique where the original code is augmented with stubs that are used to check whether an advice needs to be executed or not). The experiments also show a clear gain when comparing complete systems, e.g., our new system is an order of magnitude faster then JAsCo [18].

The rest of the paper is structured as follows: Section 2 discusses the performance problem in dynamic AOP systems. Section 3 describes our dynamic AOP system. Section 4 describes the design and the implementation of our approach and highlights key issues and design decisions. In Section 5 we present an extensive performance evaluation. Section 6 concludes the paper.

2 Motivation

Dynamic AOP offers many advantages in terms of possibilities to implement adaptation at the middleware level. The question is how much performance overhead can be tolerated in return for the ability to adapt. In what follows we discuss existing techniques and show where their limitations lie. Then we briefly outline the solution that we propose in this paper.

One of the first approaches to implementing dynamic AOP was to use the debugger support in the JVM. The first of such systems was PROSE (PROgrammable extenSions of sErvices) in a version that is based on the Java Virtual Machine Debugger Interface (JVMDI) [6]. The JVMDI-based weaver employs the debugger interface of the JVM to implement the pointcuts and join points. The JVMDI is a low-level, native interface that allows a user to register notification requests for execution events inside a JVM and to take control of the execution upon each event notification. PROSE uses the JVMDI to stop the execution of the JVM at join points (e.g., field changes, method boundaries, exception throws and handlers) and then invoke the corresponding advices. After the advice or advices have been executed, control is returned to the application. The resulting system is very effective and flexible but suffers from a considerable overhead as the JVM must run in debugging mode. For many applications, such an overhead is not acceptable.

The idea of using the debugger interface has also been proposed in JAsCo [18]. JAsCo, unlike PROSE which is Java based, uses a new language to define the aspects. The language 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. It allows to instantiate and initialize hooks. JAsCo allows to automatically transform a regular Java bean into a JAsCo bean by employing a preproc-

essor that inserts the traps using bytecode adaptation. Each trap refers to the JAsCo runtime infrastructure that manages the registered connectors and aspect beans. For improving the runtime infrastructure, JAsCo uses the Jutta system, intended to generate optimized code fragments that contain the combined aspectual behavior for each join point. To further optimize the aspect interpretation part of JAsCo, JAsCo employs the Java HotSwap technology which allows changing class definitions while the program that contains these classes is running. The runtime replacement of bytecode is done using the hotswap mechanism [16] of the Java Platform Debugger Architecture (JPDA). The hotswap mechanism allows a new class to be reloaded at runtime while under the control of a debugger. *Dmitriev* et al. [13] is considering a new API for HotSwap consisting of two calls: *RedefineMethod()* and *ExtendConstantPool()*. The first call accepts new bytecode for a method. The second call is used to extend the constant pool of a class which is needed to use new constant pool references in the bytecode of a redefined method.

The JAsCo hotswap implementation allows installing traps in only those methods that are subject to aspect application. When a new aspect is added, all the methods affected are replaced (hot-swapped) by a new version that contains the traps. Similar to the PROSE-JVMDI implementation, Jutta requires the virtual machine to run in debugging mode and, therefore, suffers from the same performance limitations.

Common to these two approaches, JAsCo and PROSE-JVMDI, is the fact that the weaving happens in two phases. First a hook is inserted into the bytecode. The hook indicates the join points and helps to notify that the execution has reached them. Then the execution of the advice is triggered, typically as a separate piece of code.

An alternative design has been proposed to avoid having to execute in debugging mode. A second version of PROSE uses the Just-in-time compiler capabilities of the JVM to speed up the process [5]. We will refer to this technique as stub-weaving. The idea is to modify the baseline (non-optimizing) JIT compiler. PROSE uses the JIT to add minimal hooks (or stubs) at all potential join points (such as field operations and method boundaries).

When the program execution reaches a join point, the stub checks if the join point is activated and calls the advice. The advantage of this approach is that the JVM does not need to run in debugging mode, which results in very significant performance gains [5]. The drawback of the PROSE stub weaving approach is that the stubs can only be placed on easily recognizable pieces of the bytecode. This prevents the system from using the optimizing compiler as that would make it very complex and in some cases impossible to identify the proper join points.

In this paper we take advantage of the insights of this previous work and improve on their performance by using a different approach. The basic idea is not to weave just hooks but, whenever possible, to weave the advice directly into the original code. The resulting system not only does not need to run in debugging mode, it can also skip in many cases the overhead of stopping at a hook or stub, executing the code of the stub, calling the advice, and then return execution to the original code. In the cases where the advice cannot be directly woven, what is woven is a more efficient form of hook, called a callback, that also helps to minimize the overhead. 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.

3 The PROSE Advice Weaver

Our implementation is based on the Jikes IBM Research Virtual Machine (RVM) [2] and extends the current infrastructure of PROSE [5,6]. The dynamic advice weaver instruments the bytecode of a method by adding callbacks which permit the corresponding advice execution. Aspects can also be removed, leaving the application in its original state. In this case, the weaver removes any advice whose aspects were unvowen. After the bytecode of a method has been replaced in the virtual machine, the method has to be recompiled. Method replacements are performed by a module, which ensures that method redefinitions are activated atomically. An extensive and detailed performance evaluation reveals how big the performance gain is compared with other AOP approaches.

In case of the dynamic advice weaver only those join points where aspects are applied upon are activated, therefore our approach does not weave unnecessary callbacks. When the aspects are removed the join points are deactivated. Method replacement makes re-JITing necessary once an aspect is inserted or removed.

The next sections present our dynamic advice weaver and show how it enables efficient dynamic AOP. Section 3.1 presents the architecture of the PROSE advice weaver. Section 3.2 illustrates the details related to the implementation of dynamic bytecode instrumentation support in Jikes RVM. Section 3.3 presents the AOP engine including the instruments needed for weaving aspects.

3.1 System Architecture

The architecture is divided into two layers: the *AOP engine* layer and the *execution monitor* layer. Fig.1 gives an overview of this architecture. The AOP engine accepts aspects (1) and transforms them into join point requests (2). It activates the join point requests by invoking methods 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 runtime. The upper layer accepts weaving requests (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. When the program execution reaches one of the activated join points (7), the execution monitor notifies the AOP engine which then executes an advice (8). When the aspects are removed the join points are deactivated, the weaver unweaves any advice whose aspects were removed, and the original bytecode of each method are

installed. Our goals were to define a clear interface of the execution monitor, which is responsible for the implementation of dynamic weavers on top of it, and to provide this interface at a low implementation cost.

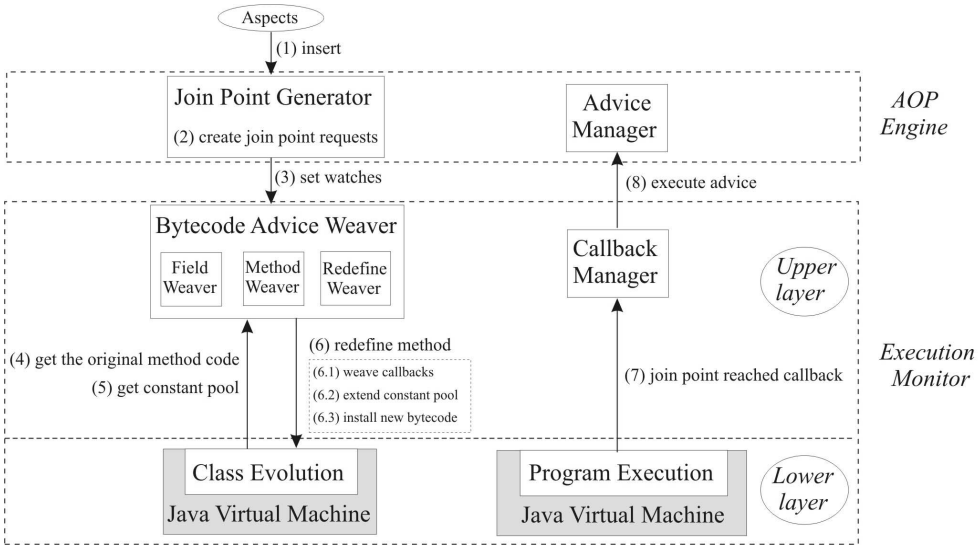

Fig.1: Architecture of the PROSE advice weaver

3.2 The Execution Monitor

The execution monitor contains the functionality for activating join points and the callback functionality for notifying the AOP engine that a join point has been reached. When a new aspect is added to the system, the AOP engine activates join points using the API methods of the execution monitor. The lower layer of the execution monitor is integrated with the JVM. It extends Jikes RVM by adding basic support for dynamic weaving, while the AOP system is treated as an exchangeable module on top of the basic support. The Bytecode Advice Weaver module is responsible for the extraction of bytecode and replacement with new bytecode simultaneously.

In this section, we will discuss the important issues related to the implementation of dynamic bytecode instrumentation support in Jikes RVM. Fig.2 contains the core interface of the execution monitor which supports VM services to change a class at runtime.

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 first parameter is the method that will be redefined. The second parameter represents the new bytecode that will be installed in the VM. 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.

```
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.2: The Class Evolution API

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 [15] 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.

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. Method redefinitions are applied one after the other and 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.

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 [11]). 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, meth-

ods which are never executed are not compiled, thus saving the overhead that would be needed to compile the method at runtime.

Implementing this API required minimal changes to the existing Jikes RVM code (90% of the necessary code was kept in the *VM_ClassEvolution* class). The advantage of implementing this API is that it is very simple to use. Our goal was to demonstrate the feasibility of dynamic bytecode instrumentation technology and measure the performance overhead related to the advice dispatching.

3.3 The AOP Engine

When an aspect is added to the system (Fig.1, step 1), the *Join Point Generator* decomposes the aspect into join point requests (step 2) and activates join points using the API methods of the execution monitor (step 3). When an active join point is reached (step 7), the *Callback Manager* notifies the AOP engine and the corresponding advice is executed (step 8).

PROSE does not define a new aspect language and provides a simple way to describe aspects. Fig.3 shows a simple PROSE aspect which redefines the original version of a method with a new one.

```
1 public class ExampleAspect extends Aspect {
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.3: Example of a PROSE aspect

Aspects in PROSE extend the *Aspect* base class (line 1). An aspect may contain one or more crosscut objects. 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.3, there is just one crosscut, corresponding to the *doRedef* instance field (line 2).

To implement a method redefinition in PROSE we introduced a new crosscut type called *MethodRedefineCut* (line 2), similar to the *around* advice construct in AspectJ. The aspect showed in Fig.3 redefines the method *bar* (line 7) from the *Foo* class (line 3) with the new code specified in the advice (line 4). When the aspect is inserted, the original bytecode of a method, in our example *bar*, in replaced with the bytecode of the advice, using the support for method code replacement at runtime.

AspectJ enables us to 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 allows to execute the surrounded code by invoking *proceed()* in the body of the advice. Our implementation doesn't support the functionality to call the redefined code in the body of the advice. The redefined code is not accessible until the aspect is withdrawn.

PROSE aspects are expressed in the Java source language. Therefore, there are some limitations in case of method redefinitions. When writing an advice method for a method redefinition, the resulting bytecode is transplanted into the methods that will be redefined. 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 was solved using Reflection which allows accessing non-public members at will.

One important issue is the support for atomic weaving. The PROSE engine provides this support as follows. Atomic weaving can be implemented by blocking the advice execution until the weaving operation completes. Every time when a join point is reached, a callback method is called, instead of the actual advice.

4 Design and Implementation

This section describes the design and the implementation of the dynamic advice weaver. Our approach weaves advice calls at runtime 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 runtime exception handler of Jikes RVM. Section 4.1 provides details concerning the bytecode instrumentation for several join point types supported by PROSE. In Section 4.2, we discuss implementation issues.

4.1 Dynamic Bytecode Instrumentation

Bytecode instrumentation [13] 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 the dynamic advice weaver employs the BCEL (Byte Code Engineering Library) [12], a bytecode manipulation library.

We will now discuss the details concerning the bytecode instrumentation for several join point types supported by PROSE.

- **Method entry join point**

  Weaving method entry advice is done by adding a call to the advice before the original bytecode of a method. The arguments of the called method are accessible from the advice because they are passed as parameters to the advice method.

- **Method exit join point**

Weaving method exit advice is much more difficult than weaving method entry advice. In this case, we have to be very careful with 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 might be 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 [14] 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 [15] 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.

- **Field access join point**

The 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.

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.

- **Field modification join point**

Weaving field modification advice can be done in the same way as for field accesses. 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.

- **Exception join points**

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 runtime 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 runtime because the handler must walk the stack for a thrown exception until it finds a *catch* block.

4.2 Implementation Details

Dynamic advice weaver provides an API in order to allow weaving advice calls at runtime into the bytecode of a method wherever they are needed. 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. 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, which performs the following steps:

1. Prepare the weaving process by initializing the BCEL bytecode generator objects.
2. If a redefine advice is registered, replace the old bytecode instructions with the transformed instructions of the advice method.
3. Iterate over all bytecode instructions. If it is a field instruction (*getstatic, getfield, putstatic, putfield*) and the referenced field is in the list of watched fields, then insert a field access/modification callback.
4. Check if a method entry join point is activated. If yes, insert a method entry callback before the first instruction.
5. Check if a method exit join point is activated. If yes, create a *try-finally* construct, move all instructions into the *try* block and add a method exit callback in the *finally* block.
6. Generate the final instructions and install them into the VM using the services of the *VM_ClassEvolution* class.
7. Clean up BCEL bytecode generator objects.

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 in-

strumentation. 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*.

Field access and modification requests are handled by the *FieldWeaver* class. For each method that references the specified field, a callback is woven. Method redefine requests are handled by the helper class *RedefineWeaver*. Method redefinition takes place if the *setRedefineAdvice()* method is called.

Weaving an advice for exception join points is different from the other join point types. 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).

5 Performance Evaluation

In order to evaluate the performance gain of our PROSE dynamic advice weaver we performed extensive benchmarks. The enhanced PROSE performance is compared to other AOP approaches. This section experimentally assesses the performance of the dynamic advice weaver approach. Section 5.1 describes the experimental setup, the benchmark suite, and the Jikes RVM configurations used in the experiments. The performance of the execution monitor introduced in Section 3.2 is assessed in Section 5.2. Finally, Section 5.3 presents the performance evaluation of the AOP engine.

5.1 Experimental Methodology

All experiments in this paper were performed on an AMD Athlon MP 1600+ 1.4 GHz, double processor machine with 1 GB RAM running Linux 2.4.20. In this paper, we compare results using the following Java environments: Jikes RVM 2.3.0.1 and Sun Java SDK 1.4.2. The IBM Jikes RVM begins execution by reading from a boot image file, which contains the core services (e.g. class loader, object allocator, compiler) of Jikes RVM precompiled to machine code [10]. Jikes RVM supports several configurations. In our experiments we ran the following Jikes RVM configurations:

- *prototype:* the Jikes RVM configured to use the baseline compiler as JIT. This configuration does not include the optimizing compiler [8] or the adaptive optimization system [9].

- *prototype-jvmai: prototype* configuration with the Jikes RVM configured to use a modified baseline compiler as a JIT. The modified baseline complier weaves minimal hooks (join point stub instructions) at native code locations that correspond to join points. This configuration does not include the optimizing compiler or the adaptive optimization system. The stub-based weaver makes use of this configuration.
- *production:* the Jikes RVM configured to use the optimizing compiler as a JIT. This is a fully functional configuration of Jikes RVM with the highest performance that includes the optimizing compiler and the adaptive optimization system. Additionally, two command line options ("-X:aos:adaptive_inlining=false" and "-X:opt:inline=false") were used to prevent inlining, and the initial compiler is set to the optimizing compiler ("-X:aos:initial_compiler=opt"). Using the last command line option all dynamically compiled methods are compiled with the optimizing compiler. The advice-based weaver makes use of this configuration.

We evaluate our approach using three benchmark applications: SPECjvm98 [4], Java Grande [3], and JAC [22]. SPECjvm98 is a benchmark suite which consists of different tests that measure the efficiency of JVM, the just-in-time (JIT) compiler, and operating system implementations. SPECjvm98 applications include text compression, MPEG decoding, compilation speed, graphics, and database functions. The Java Grande benchmark suite consists of a set of benchmarks that are relevant for testing the performance of Java for scientific computations. The benchmark suite is broken up into three different areas: low-level benchmarks that test general language features and operations; scientific and numerical application kernels; and full-scale science and engineering applications. The benchmarks used in this paper are the scientific and numerical application kernels, short codes which reflect the type of computation which might be found in the most computationally intense parts of real numerical application. The JAC benchmark consists of a set of public methods with different method signatures and simple method body implementation. This benchmark allows to measure the overhead for applied aspects.

5.2 Evaluation of the Execution Monitor

To measure the efficiency of the advice execution monitor, we performed a number of micro-measurements. We compare the advice weaver with the stub-based weaver and AspectJ. For a micro-measurement, we calculate the time needed to execute a simple operation (e.g. an empty method call, a field set, a field get, exception throw and handler). We measure the execution time of a simple operation for a large number of iterations and for each join point type. We ran all the experiments one hundred times and then we calculate the average of the execution times measured. The standard deviation for all the micro-measurements is less than 8%.

The micro-measurement results are summarized in Table 1 and Table 2. The results indicate how much time is spend in the execution monitor and is a good indicator of the AOP system efficiency. Each row contains the average time needed to execute a bytecode instruction, under various configurations of AOP support.

In the first experiment, we made the measurements with an activated join point (e.g. method boundaries (entry, exit), field operations (access, modification)). Each micro-measurement has an activated join point with a simple advice attached to it that increases

a counter each time it is executed. The results are summarized in Table 1. The first column contains the seven basic operations that we have evaluated. The second and the third columns represent the time needed to execute an instruction when the execution monitor has one active join point. An activated join point always results in a call to the advice manager component. The cost in case of the stub weaver is significant: 400 ns/instruction, roughly the time needed to execute an *invokevirtual* instruction. The cost is significantly reduced (by half) in case of the advice weaver: the times needed to execute all method invocation instructions are roughly 200 ns/instruction. In case of field instructions, the execution time is roughly 350 ns/instruction. For comparison, a similar AspectJ advice (that increases a counter each time it is executed), compiled into the test has been used. AspectJ is employed as performance reference.

	Stub weaving		**Advice weaving**		**AspectJ**	
Instruction type	Call to advice method because of an active join point		Call to advice method because of an active join point		AspectJ call to an advice	
getfield	395.2 ns		347.91 ns		11.48 ns	
putfield	389.97 ns		350.66 ns		12.44 s	
invokevirtual	409.79 ns	408.58 ns	183.13 ns	185.81 ns	17.63 ns	17.93 ns
sync invokevirtual	575.11 ns	569.9 ns	211.18 ns	206.3 ns	45.74 ns	44.79 ns
invokeinterface	672.61 ns	675.34 ns	200.16 ns	194.08 ns	16.23 ns	24.58 ns
invokestatic	395.7 ns	389.98 ns	190.14 ns	185.28 ns	14.88 ns	16.33 ns
invokespecial	400.79 ns	402.88 ns	190.48 ns	187.65 ns	17.98 ns	17.78 ns
	Met.Entry	Met.Exit	Met.Entry	Met.Exit	Met.Entry	Met.Exit

Table 1: Join point costs on the JVM with AspectJ, stub weaver and advice weaver

In the second experiment, we made the measurements under other configurations of AOP support. Table 2 illustrates the results of this experiment. The first column contains the seven basic operations that we have evaluated. The second and the fourth columns represent the time needed to execute an instruction when the execution monitor has no active join points. The third and the fifth columns contain the cost of executing an operation for which a join point was registered and then locked.

	Stub weaving			**Advice weaving**		
Instruction type	No call to advice because join point is not activated	No call to advice because join point is activated but locked		No call to advice because join point is not activated	No call to advice because join point is activated but locked	
getfield	16.45 ns	19.32 ns		3.55 ns	311 ns	
putfield	15.15 ns	20.03 ns		2.93 ns	307.35 ns	
invokevirtual	33.64 ns	35.06 ns	36.31 ns	6.72 ns	169.94 ns	162.75 ns
sync invokevirtual	185.32 ns	184.61 ns	184.61 ns	41.12 ns	184.85 ns	182.87 ns
invokeinterface	287.17 ns	284.93 ns	301.33 ns	12.89 ns	172.92 ns	170.17 ns
invokestatic	25.04 ns	27.47 ns	27.75 ns	16.53 ns	164.89 ns	163.32 ns
invokespecial	28.98 ns	31.61 ns	29.34 ns	7.15 ns	166.3 ns	163.08 ns
		Met.Entry	Met.Exit		Met.Entry	Met.Exit

Table 2: Join point costs on the JVM with stub weaver and advice weaver

When program execution reaches a join point, the code stubs emitted by the stub weaver first check if the join point is active or not. The advice weaver does not perform this check because advice callbacks are only woven for active join points. The bytecode of a method without active join points is equal to the original bytecode of the method. Therefore, the overhead for inactive join points is heavily reduced.

If a join point is active, both weaving approaches must check whether the join point is locked. The stub weaver performs this check right after checking for join point activity. Only a small overhead increase is observed compared to the case of an inactive join point. The values for the advice weaver are larger. When the advice weaver weaves an advice callback into a method, some information has to be passed to the advice: a reference to the current instance ("this") is passed as well as the method that reached the join point. Additionally, all the arguments of the method containing the callback are forwarded.

Although the advice weaver is slower for locked join points this is not a major disadvantage. The advice weaver is faster for active join points as can be seen from the previous measurements. Locked join points occur only occasionally, they are much rarer than active or inactive join points.

In order to assess the performance gain of the advice execution monitor, we made other micro-measurement tests. Table 3 illustrates the results of this experiment. In this experiment, we made the measurements with an activated exception join point (exception throw, exception catch). The results for the exception join points must be interpreted independently. Both the stub weaver and the advice weaver implement the exception join points by extending the exception handling mechanism of the VM. Exception handling is slow because the VM must walk the method frames on the stack and seek for an appropriate handler (catch block). Therefore, the measured results are bigger compared to the other join point types. The stub weaver can not use the optimizations available in Jikes RVM because only the baseline compiler is modified to emit the stubs. The advice weaver can use the adaptive optimization system of Jikes RVM and is therefore faster.

	Stub weaving	**Advice weaving**	**AspectJ**
Instruction type	Call to advice method because of an active, unlocked join point	Call to advice method because of an active, unlocked join point	AspectJ call to an advice
Exception Throw	7183.3 ns	5285.9 ns	-
Exception Catch	7192.6 ns	5309.5 ns	22.81 ns

Table 3: Join-point costs on the JVM with AspectJ, stub weaver and advice weaver

Benchmark	**Execution time with AOP support**	
	Stub weaving	**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
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

Table 4: The execution times for the stub and advice weavers with AOP support for method boundaries, method redefinition, field sets, field gets, and exception handlers

In the fourth experiment, we compare the original JVM with the JVM containing the execution monitor. In this experiment the execution monitor is not activated. 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 4 summarizes the average execution times for each test for the stub and advice weavers. Fig.4 and 5 shows 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%.

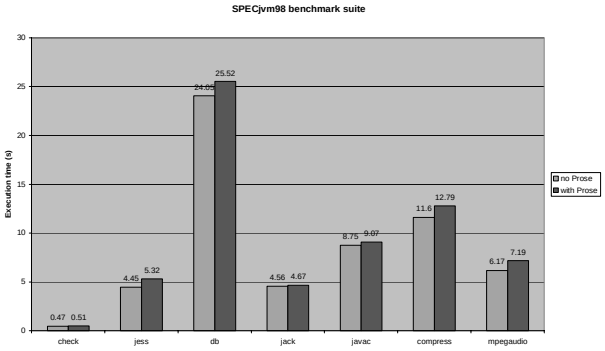

Fig.4: Relative overhead for advice weaver with AOP support for method boundaries, method redefinition, field sets, field gets, and exception handlers

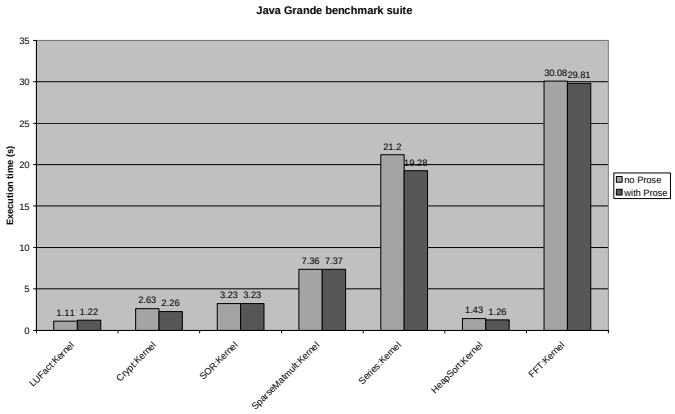

Fig.5: Relative overhead for advice weaver with AOP support for method boundaries, method redefinition, field sets, field gets, and exception handlers

When an aspect is inserted, the matching join points are activated which implies that all affected methods have to be recompiled by the JIT compiler. In this experiment, we measured the compilation time for each join point type. Each measurement has an activated join point with a simple advice attached to it that contains a simple field operation

each time it is executed. The methods are compiled on first invocation with the baseline or the optimizing compiler and no recompilations will take place. Table 5 shows the average compilation times for one hundred runs. The standard deviation is less than 5%.

Type of join points	Relative rejit overhead	
	Optimizer compiler	Baseline compiler
Method entry	3.59 ms	0.19 ms
Method exit	3.51 ms	0.26 ms
Method redefinition	1.01 ms	0.10 ms
Field access	3.46 ms	0.22 ms
Field modification	3.34 ms	0.23 ms

Table 5: Relative rejit overhead

5.3 Evaluation of the AOP Engine

Invoke type	Parameters	Jikes / PROSE (Stub weaving)		Jikes / PROSE (Advice weaving)		Jikes / AspectJ	
invokevirtual	()	37.91 µs	36.01 µs	3.82 µs	3.74 µs	0.07 µs	0.07 µs
sync invokevirtual	()	36.24 µs	36.06 µs	3.69 µs	3.44 µs	0.13 µs	0.13 µs
invokeinterface	()	35.76 µs	35.67 µs	3.63 µs	3.68 µs	0.09 µs	0.09 µs
invokestatic	()	31.57 µs	30.92 µs	3.66 µs	3.66 µs	0.04 µs	0.05 µs
invokespecial	()	35.13 µs	35.29 µs	3.67 µs	3.7 µs	0.05 µs	0.05 µs
invokevirtual	(Object, Object)	40.95 µs	40.67 µs	4.1 µs	3.78 µs	0.05 µs	0.05 µs
sync invokevirtual	(Object, Object)	41.32 µs	40.86 µs	3.82 µs	4.07 µs	0.11 µs	0.12 µs
invokeinterface	(Object, Object)	40.98 µs	40.99 µs	3.79 µs	4.09 µs	0.08 µs	0.08 µs
invokestatic	(Object, Object)	37.96 µs	37.87 µs	4.03 µs	3.79 µs	0.04 µs	0.04 µs
invokespecial	(Object, Object)	40.58 µs	40.54 µs	4.04 µs	4.08 µs	0.05 µs	0.06 µs
invokevirtual	(int, int)	38.68 µs	38.39 µs	4.17 µs	4.48 µs	0.06 µs	0.05 µs
sync invokevirtual	(int, int)	38.55 µs	38.36 µs	4.23 µs	4.39 µs	0.1 µs	0.09 µs
invokeinterface	(int, int)	38.82 µs	38.14 µs	4.16 µs	4.2 µs	0.09 µs	0.09 µs
invokestatic	(int, int)	33.88 µs	34.1 µs	4.36 µs	4.18 µs	0.03 µs	0.04 µs
invokespecial	(int, int)	37.82 µs	38.05 µs	4.32 µs	4.42 µs	0.05 µs	0.05 µs
invokevirtual	(long, long)	38.4 µs	38.33 µs	4.26 µs	4.23 µs	0.06 µs	0.07 µs
sync invokevirtual	(long, long)	38.7 µs	38.11 µs	4.47 µs	4.48 µs	0.1 µs	0.1 µs
invokeinterface	(long, long)	38.7 µs	38.49 µs	4.48 µs	4.47 µs	0.09 µs	0.09 µs
invokestatic	(long, long)	34.02 µs	34.32 µs	4.45 µs	4.46 µs	0.04 µs	0.04 µs
invokespecial	(long, long)	38.12 µs	38.63 µs	4.51 µs	4.49 µs	0.05 µs	0.05 µs
invokevirtual	(double, double)	37.96 µs	38.28 µs	4.6 µs	4.59 µs	0.06 µs	0.06 µs
sync invokevirtual	(double, double)	38.61 µs	38.15 µs	4.57 µs	4.66 µs	0.11 µs	0.11 µs
invokeinterface	(double, double)	38.8 µs	38.51 µs	4.72 µs	4.65 µs	0.08 µs	0.09 µs
invokestatic	(double, double)	34 µs	34.23 µs	4.55 µs	4.58 µs	0.05 µs	0.04 µs
invokespecial	(double, double)	38.43 µs	38.45 µs	4.85 µs	4.58 µs	0.05 µs	0.05 µs
		Met.Entry	Met.Exit	Met.Entry	Met.Exit	Met.Entry	Met.Exit

Table 6: Micro measurements with PROSE and AspectJ

To measure the efficiency of the dynamic advice weaver, we made a number of micro-measurements, this time with the complete AOP system (the AOP engine running on top of the execution monitor). In this experiment, one simple aspect is applied upon a method. This aspect describes an advice that increases a counter each time it is executed. We measured the cost of executing an advice for a method invocation operation for one

million method calls. We ran all the experiments one hundred times and then we calculate the average of the execution times measured. Table 6 illustrates the results. Each line contains the total time needed to execute a method call plus an additional advice on method entry and exit. The advices were woven dynamically using the stub weaver (column 3) and the advice weaver (column 4), and statically using AspectJ (column 5). All the method invocation instructions that we have evaluated are contained in the first column. We tested different method signatures with parameters of different types (column 2). We repeated the measurements until the standard deviation is less than 7%.

The cost of executing an advice for advice weaver is roughly 10 times faster than that of executing an advice for stub weaver. As illustrated by Table 6, the performance gain of the advice weaver is very significant. The complete advice weaver (execution monitor plus AOP engine) is significantly faster than the stub weaver.

5.4 Comparisons with JAsCo including the just-in-time compiler Jutta

In order to evaluate the performance of PROSE advice weaver, we compare it also with an other AOP approach: JAsCo including the just-in-time-compiler Jutta. We made similar experiments as Vaderperren et al. [18]. We employed the JAC benchmark application for all the experiments. The benchmark consists of a set of eight public methods with different method signatures and simple method body implementation. For each AOP approach, 100000 “direct” iterations were performed.

In the first experiment, one aspect is applied upon each public method. The aspect describes an around advice that increases a counter each time it is executed. The results of this experiment are summarized in Table 7. As illustrated by Table 7, the performance gain of the PROSE advice weaver is very significant. This is mainly the mechanism support for method code replacement at runtime. Our approach weaves the actual advice code and therefore improves the performance due to a reduction of the number of indirect references.

One around aspect / eight public methods	JAC benchmark
AspectJ 1.1.1	6 ms
PROSE 1.2.0	6 ms
JAsCo 0.5.3	247 ms

Table 7: Measurements with PROSE, JAsCo and AspectJ

In the last experiment, one single around aspect is applied upon one specific method defined within the JAC benchmark application. Table 8 illustrates the results of this experiment. The experiments show a clear gain when comparing complete systems, e.g., our new system is an order of magnitude faster then JAsCo.

One around aspect / one public method	JAC benchmark
AspectJ 1.1.1	2 ms
PROSE 1.2.0	4 ms
JAsCo 0.5.3	33 ms

Table 8: Measurements with PROSE, JAsCo and AspectJ

6 Conclusions

In this paper we have presented the PROSE advice weaver, a modular and flexible architecture intended to improve the efficiency of the dynamic AOP process. We proposed a mechanism to implement dynamic AOP through method code replacement at runtime. 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. In this paper we presented the technique in detail, perform extensive benchmarks to evaluate the performance gain of this approach and compare it with other dynamic AOP systems. The measurements show that our approach is more efficient than the existing solutions. PROSE is an open source project and can be obtained from http://prose.ethz.ch.

References

1. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, ECOOP '97 - Object-Oriented Programming 11th European Conference Jyvaskyla, Finland, volume 1241 of Lecture Notes in Computer Science, pages 220-242. Springer-Verlag, New York, NY, June 1997.
2. B. Alpern, D. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno virtual machine. In IBM System Journal, 39(1), Feb. 2000.
3. Java Grande Forum benchmark suite. http://www.javagrande.org.
4. Spec - Standard Performance Evaluation Corporation. http://www.spec.org/osg/jvm98/.
5. A.Popovici, G.Alonso, and T.Gross. Just-In-Time aspects: Efficient dynamic weaving for Java. In 2nd Intl. Conf. on Aspect-Oriented Software Development, Boston, USA, March 2003.
6. A.Popovici, T.Gross, and G.Alonso: Dynamic Weaving for Aspect Oriented Programming. In 1st Intl. Conf. on Aspect-Oriented Software Development, Enschede, The Netherlands, April 2002.
7. AspectJ website: http://www.aspectj.org
8. M.G. Burke, J.D. Choi, S. Fink, D. Grove, M. Hind, V.Sarkar, M.J. Serrano, V.C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno dynamic optimizing compiler for Java. In ACM Java Grande Conference, San Francisco, June 12-14, 1999.
9. M. Arnold, S. Fink, D. Grove, M. Hind, P. F. Sweeney. Adaptive Optimization in the Jalapeno JVM. In ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'00), 2000.
10. B. Alpern, D. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, M. Mergen, T. Ngo, J. Shepherd, S. Smith. Implementing Jalapeño in Java. In ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99), Denver, Colorado, November 1, 1999.
11. The Jikes Research Virtual Machine User's Guide 2.3.0.1. http://www-124.ibm.com/developerworks/projects/jikesrvm/, 9 Nov. 2003.

12. The Byte Code Engineering Library (BCEL) manual. http://jakarta.apache.org/bcel/manual.html.
13. M. Dmitriev. Application of the HotSwap Technology to Advanced Profiling. ECOOP'02 workshop on Unanticipated Software Evolution, 2002.
14. B. Joy, G. Steele, J. Gosling, G. Bracha. The Java Language Specification. Addison-Wesley, Second edition, 2000. http://java.sun.com/docs/books/jls/.
15. T. Lindholm, F. Yellin. The Java Virtual Machine Specification. Addison-Wesley, Second Edition, 1999. http://java.sun.com/docs/books/vmspec/.
16. M. Dmitriev. Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications. In: Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001 International Conference, Tampa Bay, Florida, USA, October 14-18, 2001.
17. R. Pawlak, L. Seinturier, L. Duchien, G. Florin. 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, September 2001.
18. W. Vanderperren, D. Suvée. Optimizing JAsCo dynamic AOP through HotSwap and Jutta. In: Proceedings of the 2004 Dynamic Aspects Workshop (DAW04), pages 120-134, Lancaster, England, March 2004.
19. J. Boner, A. Vasseur. AspectWerkz website: http://aspectwerkz.codehaus.org, 2004.
20. G. Kniesel, P. Constanza, M. Austermann. JMangler - A Framework for Load-Time Transformation of Java Class Files. IEEE Workshop on SCAM'01, November 2001.
21. R.Chitchyan, I. Sommerville. Comparing Dynamic AO Systems. In: Proceedings of the 2004 Dynamic Aspects Workshop (DAW04), pages 120-134, Lancaster, England, March 2004.
22. JAC website: http://jac.objectweb.org.