Virtual Machine Support for Dynamic Join Points

Christoph Bockisch Michael Haupt Mira Mezini Klaus Ostermann
Darmstadt University of Technology, Germany
{bockisch,haupt,mezini,ostermann}@informatik.tu-darmstadt.de

ABSTRACT

A widespread implementation approach for the join point mechanism of aspect-oriented languages is to instrument areas in code that match the static part of pointcut designators, inserting dynamic checks for that part of matching that depends on run-time conditions, if needed. For performance reasons, such dynamic checks should be avoided whenever possible. One way to do so is to postpone weaving of advice calls until run-time, when conditions determining the emergence of join points hold. This calls for *fluid code*—code that adapts itself to the join point emergence at run-time, and suggests that AOP concepts should be integrated into the execution model underlying a VM. In this paper, we present first steps toward such an integration in *Steamloom*, an extension of IBM's Jikes Research Virtual Machine. Steamloom is fairly restricted, but our initial experimental results indicate that aspect-aware VMs and fluid code are promising w.r.t performance. While the focus in this paper is on performance, there are other advantages of aspect-aware VMs to be investigated in the future.

1. INTRODUCTION

Dedicated support for fluid code at virtual machine level is needed for an efficient implementation of dynamic join points. This is the message we want to put forward in this section. We start by explaining our understanding of dynamic crosscutting, followed by a discussion of strategies to enable it. Next, we discuss why dynamic crosscutting should have VM support for fluid code, especially (but not only) for performance reasons, and present in a nutshell our approach to such support.

1.1 Dynamic Join Points

The notion of join points is an important concept of aspect-oriented programming. Crosscuts are sets of related join points which are defined by pointcut designators. A special class of pointcuts are those that can directly be mapped to locations in the program code. A method execution pointcut, e. g., can be directly mapped to the place(s) in the code implementing the method. In the following, we call such crosscuts *code-level crosscuts*. Other crosscuts such as the one described by "whenever x is executed in the control flow of y" cannot directly be mapped to a location in the code. They rather emerge depending on the dynamics of the program execution, hence, we call them *dynamic crosscuts*.

We distinguish two classes of dynamic crosscuts. First, there is the class of *statically bound dynamic crosscuts* for which we can statically determine a set of *potentially* affected code locations (join point shadows in the terminology of [12]). An example of this family is the crosscut defined by `cflow(aPCD) && otherPCD`[1], where `otherPCD` is assumed to be a code-level crosscut. In this case, we can statically determine the set of shadows for the `otherPCD` part and then make sure at run-time that only those statically determined shadows that occur in the control flow of `aPCD` actually yield a join point.

The second class of dynamic crosscuts are those whose correspondence to code locations cannot be restricted in a reasonable way before run-time, for which reason we call them *unbound dynamic crosscuts*. Consider, e. g., an aspect that counts the invocations of certain methods during the execution of a program. The set of methods whose invocations are counted has, however, to be dynamically editable by the user, so it can only be known at run-time. In this case, the static shadows will include all method invocation locations in code. Also unbound are crosscuts resulting from dynamic aspects, i. e., aspects that are woven at run-time, as supported by some AOP approaches [25, 8, 16].

Let us now consider how well these different levels of dynamic crosscuts can be supported with static machine code (non-fluid code). Statically bound dynamic join points can be implemented by instrumenting the set of potentially affected code locations with dynamic checks, which can either be generated by a compiler, bytecode weaver or class loader or programmed manually, e. g., by conditional logic in the advice code. Manual checks are tedious and fragile because a small change in the program or requirements may invalidate them. There are different languages that offer built-in constructs for expressing certain dynamic crosscuts, e. g., via `cflow` in AspectJ. Mapping such PCDs to locations in code by integrating the needed check dynamics is done by the compiler (weaver, class loader), which makes the maintenance problem of manual checks less severe. The problem remains with dynamic crosscuts that lack direct support, such as the unbound dynamic one discussed above. In As-

[1] *PCD* is an abbreviation for *pointcut designator*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AOSD 04, March 2004, Lancaster UK.
Copyright 2004 ACM 1-58113-842-3/03/0004 ...$5.00.

pectJ, one could solve the described problem by implementing an aspect that is parameterised by a `HashSet` containing the names of the methods to count and implements an advice that checks at *every* method invocation whether the counting functionality applies to the current method. The same check could also be implemented by an `if` PCD, but would still be done manually.

This brings us to unbound dynamic crosscuts. A technique often used to enable such a dynamic binding of pointcuts is to have the compiler insert dynamic checks at *all* join points, that is (depending on the granularity of the join point model) basically at every instruction. This is, in principle, the approach taken by several implementations [25, 8, 16] that support some form of dynamic aspect binding (cf. Sec. 4 for a more detailed discussion). During execution, the inserted hooks check whether aspects are registered for a reached join point, and, if so, call the respective advice. With this workaround, flexibility is gained at the cost of a very significant performance slowdown.

1.2 Programmatic Aspect Deployment

Most of the approaches discussed so far have in common that the scope of an aspect's effect is declaratively defined as part of the aspect definition. A different approach relevant to this work is *programmatic aspect deployment*, a feature available in CAESAR [21]. In CAESAR, an aspect has to be *deployed* for its pointcuts and advice to take effect. An aspect can be statically deployed by adding the modifier `deploy` to its declaration: compiling such an aspect is equivalent to compiling a corresponding AspectJ aspect.

In addition CAESAR aspects can also be deployed using the `deploy(anAspectInstance) { aBlock }` statement. Pointcuts and advice of `anAspectInstance` are effective only inside the control flow of the code within the `deploy` block. Deployment is always local to the thread in which it is encountered. Actually, `deploy` can be thought of as an operation understood by aspect instances, the semantics of the latter being basically that of containers of PCDs and advice associated with them. Such an operation can be part of arbitrary computations, i. e., the result of arbitrary computations may determine where/when an aspect is deployed, hence the attribute "programmatic" to the deployment mechanism of CAESAR.

To illustrate how programmatic deployment enables dynamic crosscuts, consider the simple application in Fig. 1 that determines nth Fibonacci number. The AspectJ aspect in Fig. 2 is used to determine the number of `fib()` calls during `fibstart()` execution when the latter occurs within the control flow of a call to `m2()`. The aspect sets a counter to 0 at the beginning of `fibstart()`'s execution, increments it by one at the beginning of each execution of `fib()`, and prints its value after the execution of `fibstart()`.

The CAESAR version of this aspect is given in Fig. 3. The aspect class `DeploymentAspect` is statically deployed; it decorates any call to `TestApp.m2()` with an `around()` advice, within the body of which an instance of `FibonacciAspect` is created and deployed in the context of a block containing the `proceed()` call. This way, `FibonacciAspect` affects executions mentioned in its PCDs only when they occur within the `deploy` block, i. e., in the control flow of `TestApp.m2()`. Note that `FibonacciAspect` does not contain a `cflow` PCD.

We highlight programmatic deployment in connection with the issue of efficiently supporting dynamic crosscuts for two

```
class TestApp {
  public void m1(int n) { fibstart(n); }
  public void m2(int n) { fibstart(n); }
  public void fibstart(int n) { fib(n); }
  public int fib(int k) {
    return (k > 1) ? fib(k-1)+fib(k-2) : k;
  }
}
```

Figure 1: Determining the nth Fibonacci number.

```
public aspect FibonacciAspect {
  private int ctr = -1;
  pointcut m2cf(): cflow(call(void TestApp.m2(int)));
  before(): execution(void TestApp.fibstart(int))
      && m2cf() { ctr = 0; }
  after(): execution(void TestApp.fibstart(int))
      && m2cf() { System.out.println(ctr); }
  before(): execution(int TestApp.fib(int))
      && m2cf() { ctr++; }
}
```

Figure 2: Counting `fib()` invocations in AspectJ.

reasons. First, we feel that from the perspective of the programming model, programmatic deployment is a more natural way to specify dynamic conditions under which an aspect applies (the dynamic scope of the aspect) than supporting more declarative dynamic pointcut specifications in the language. Especially, programmatic deployment enables the programmer to express dependencies between join points that can be taken into consideration to postpone creating shadows for some join points only after other join points that the former depend on are reached. For example, the pointcuts of `FibonacciAspect` in Fig. 3 need to be considered only if `call(void TestApp.m2())` has matched.

In addition to expressing dependencies among join points, CAESAR's deployment mechanism also enables *aspectual polymorphism.* Since the type of an aspect instance passed to a `deploy` block is statically known only by upper bound, the same code may be executed decorated with aspectual behaviour or not, depending on whether the code is included within a `deploy` block or not. Furthermore, the set of affected join points as well as the advice code to be executed at these points are dynamically bound. For illustration, recall the example of counting the invocations of a set of methods

```
public class FibonacciAspect {
  private int ctr = -1;
  before():
    execution(void TestApp.fibstart(int)) { ctr = 0; }
  after(): execution(void TestApp.fibstart(int)) {
      System.out.println(ctr);
  }
  before(): execution(int TestApp.fib(int)) { ctr++; }
}

deploy public class DeploymentAspect {
  around(): call(void TestApp.m2(int)) {
    deploy (new FibonacciAspect()) { proceed(); }
  }
}
```

Figure 3: Caesar version of the `fib()` counter aspect.

which is only bound at run-time. In CAESAR, one would write a static deployment aspect that calls the application whose methods we want to count within a `deploy` block parameterised by an aspect of some type `CountingAspect`. At run-time, we can pass different instances of the invocation counter type, each intercepting the invocations of different sets of methods.

In [21], we discuss CAESAR's programmatic deployment and aspectual polymorphism from the perspective of language design and typing issues. In this paper, we consider implementation issues and will show that when supported at the VM level, dynamic crosscuts as enabled by CAESAR's programmatic deployment become feasible from the performance point of view. Our initial experimental results indicate that programmatic deployment backed by VM support enables both statically bound and unbound dynamic crosscuts more efficiently than approaches based on weaving residual checks at all code locations that might yield dynamic join points.

1.3 Structure-Preserving Compilation

At the implementation level all approaches mentioned so far, including a first implementation of CAESAR as an extension of AspectJ [10], lose information during weaving because they employ *pre-run-time invasive weaving*. Weaving is invasive in that it flattens the module structure: aspects are (partly) woven into the code of other modules and hence are not identifiable as units with aspect semantics at runtime. The extent to which modular structure is flattened may be different in different cases. Some approaches might in-place weave advice code in the identified code locations, thus completely losing aspects as identifiable units at runtime. Other approaches, e. g., AspectJ, turn aspects into "normal" Java classes and weave calls to methods in these classes, which are generated out of advice code. In this case, the modular specification of the crosscut is flattened. Advice exist as identifiable units at run-time, but such units are turned into "standard" Java methods and have nothing special to identify them as aspectual units. There is no notion of a pointcut available at run-time—this information is "woven away".

We identify the loss of modular structure as the *impedance mismatch* between current aspect-oriented languages and their execution model, which is basically that of object-oriented languages. Our vision is to develop an execution model within which the module structure of aspect-oriented source-code is still available at run-time. Invasive weaving should be replaced by *structure-preserving compilation* (SPC), meaning that the program's original structure should still be available or at least easily recoverable at run-time.

Intercepting join points should not be enabled by "patching in" interception logic at points in code: every piece of code should implicitly be ready to be extended with join points. An analogy is the implicit support for late binding in object-oriented code: we do not need to post-process or recompile source code or otherwise manipulate existing classes when new subclasses are added; nevertheless, methods existing in the original program might get rebound when called in the context of executing objects of added subclasses.

We envisage that SPC for aspect-oriented languages will bring about a number of important advantages. In this paper, we focus on advantages in terms of efficiently supporting dynamic crosscuts. There are other benefits, though, which remain to be elaborated on in future work. For example, SPC enables reasoning about aspects at run-time; similar to meta-object protocols, meta-aspect protocols become conceivable. Furthermore, debugging and profiling are much easier when there is a direct correspondence between run-time and source code entities; true separate compilation is another positive side-effect of replacing weaving by SPC.

1.4 Steamloom in a Nutshell

In this paper, we present initial results of supporting unbound dynamic join points, programmatic deployment, and structure preserving compilation of aspects in IBM's Jikes Research Virtual Machine [17]. We make use of the openness of its design that allows us to add initial support for AOP, while still being able to execute standard Java bytecode with low overhead. Our extension is called *Steamloom*.

No pre- or post-processing of the base code is needed in Steamloom to integrate an aspect: advice integration and execution are rather taken care of by the VM. An aspect has a first-class representation in Steamloom as a container of code-level pointcuts and advice associated with them. Pointcuts are themselves represented as first-class entities, which is crucial for supporting unbound dynamic join points.

Steamloom also has dedicated support for CAESAR's deployment approach: the execution of a `deploy` statement with an aspect as a parameter triggers aspect weaving, i. e., the hooks needed to execute advice can be added and deleted at run-time. Hence, the set of code-level crosscuts to be intercepted can be determined at run-time. This mechanism works even in the presence of sophisticated optimisation techniques like inlining.

Two different classes of dynamic scoping mechanisms are supported: thread-local and instance-local deployment. The former means that an aspect is activated only in the scope of a particular thread, whereas the latter means that the aspect applies only to selected instances of a class. The rationale behind supporting these two kinds of scoping mechanism is that we hold these two mechanisms to be particularly useful and hard to implement efficiently without VM support. Thread-local deployment is a way to reconcile dynamic aspect deployment and multi-threading because we think that dynamic aspect deployment is a big challenge with respect to maintaining the consistency of the system. Instance-local deployment is useful for aspects that fulfil a similar purpose as roles in role models, where the behaviour of individual objects may change at run-time.

The set of code-level pointcuts currently supported by Steamloom is fairly primitive, supporting only the method execution primitive pointcut. However, a rich code-level pointcut model is not in the focus of this paper; we believe that other pointcuts can be added easily. Our focus is on showing that dynamic crosscuts can be enabled efficiently when directly supported by the VM. The results we have achieved so far in this respect are indeed encouraging. Our experiments show that VM-supported dynamic deployment results in better performance than statically weaving conditional logic to achieve equivalent context dependent activation of aspectual functionality.

1.5 Organisation of the Paper

The remainder of the paper is organised as follows. Sec. 2 presents the Steamloom architecture, while the results of our performance evaluation are presented in Sec. 3, along

with future work directions. Sec. 4 discusses related work and briefly evaluates it. Sec. 5 summarises the paper.

2. STEAMLOOM

Steamloom supports dynamic integration of aspects with global, thread-local, or instance-local scope, as outlined in the introduction. Steamloom is implemented on top of IBM's Jikes Research Virtual Machine (RVM) [17]. The extensions to the RVM are basically threefold: (a) an API, (b) a bytecode manipulation toolkit (BAT, "Bytecode Augmentation Toolkit" [4]), and (c) support for weaving. The API and bytecode toolkit are isolated in packages, while weaving support consists of extensions to several of the RVM's classes concerning VM-internal class and object representation and native code generation. We will now shortly describe the programming model supported by the Steamloom API, followed by a discussion of support for weaving. The bytecode toolkit will be mentioned when relevant, but a discussion of it is out of the scope of this paper.

2.1 Steamloom's Programming Model

The Steamloom API provides access to aspect building and deploying functionality. Pointcuts, advice and aspects are modelled as first-class entities. A pointcut can easily be created by instantiating one of the pointcut classes, and an advice is represented by an instance containing the signature of an arbitrary Java method along with information about parameter passing. To build aspects, the `associate(Pointcut, Advice)` method can be called on an instance of the `Aspect` class. After that, the aspect itself knows which methods it affects and what advice have to be called at which join points. To Steamloom, an aspect is a *container* that maps pointcuts to advice. Global aspect deployment takes place by invoking `deploy()` on the appropriate aspect instance; aspects are undeployed by calling `undeploy()`. The `deploy()` method can be passed a `Thread` instance if the aspect is to be deployed thread-locally, and/or some object if it is to be deployed on that instance only.

At the moment, `before()` and `after()` advice for method execution join points can be declared. They can be arbitrary methods that can retrieve information from the join point's context, e. g., the caller, callee or method parameters. Such methods are encapsulated in instances of the `Advice` class, along with parameter passing information that is set by calling special methods at advice construction time.

2.2 Weaving Support in Steamloom

We will now first give a short overview of the concepts in the RVM we have exploited to support dynamic weaving. Next, we will present our extensions to these concepts.

2.2.1 *An Overview of the RVM*

The overall openness of the RVM's architecture makes it very valuable for experimental extensions. The RVM is a virtual machine for Java implemented in Java. There is no interpreter; the RVM is completely based on just-in-time compilation, offering the choice among three different compiler systems: a "baseline" compiler that performs no optimisation whatsoever, an optimising compiler [20], and an adaptive optimisation system (AOS) built on top of the other compilers that performs online profiling [1, 2]. The compiler architecture is chosen at RVM building time. Unlike other approaches that use the baseline compiler for their extensions to the RVM (cf. Sec. 4), we have chosen to incorporate support for dynamic join points with the AOS, allowing for treating advice code like normal method code, including performance advantages gained by optimised compilation.

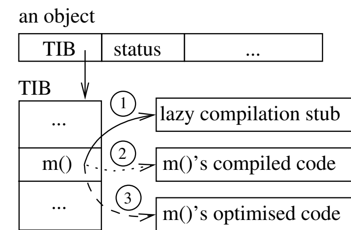

Figure 4: The RVM's normal treatment of methods.

For every class loaded in the RVM, there exists an instance of `VM_Class` holding a list of the class's members. An instance is represented in memory by an object that is a concatenation of slots for header, attributes and other information. Central for our approach is the slot containing a pointer to the TIB (Type Information Block) of the respective object's class. The TIB contains pointers to all virtual methods of the class. Every such method is represented by an instance of `VM_Method`. Static methods are held in the global JTOC (Jikes Table of Contents) that moreover contains the TIBs of all loaded classes.

Prior to the first invocation of a method, any TIB and JTOC method entry points to the singleton *lazy compilation stub*, which is itself a Java method, as illustrated by Fig. 4 (index 1). The first time a method is invoked, the stub is executed. It inspects the call stack to retrieve the callee object, its class, and the called method. Using this information, the corresponding `VM_Method` holding the method's bytecode can be retrieved. The stub compiles the method, sets the TIB (or JTOC, in case of a static method) entry to point to the compiled code and executes the method (Fig. 4, index 2). Next time the method is called, the compiled code is executed. This technique is called *lazy compilation.*

The AOS has several subsystems [1]. The *run-time measurements subsystem* gathers profiling data and stores it in the AOS database. The *controller* receives events from the measurements subsystem and, based on them and stored data, decides on method recompilation. Finally, the *recompilation subsystem* which is notified by the controller whenever a method has to be recompiled, takes care for doing so, invoking the optimising compiler at the appropriate optimisation level.

In the presence of the AOS, every method is initially baseline-compiled. As soon as the controller decides that an optimised version of that method will yield a performance improvement, it is recompiled accordingly and reinstalled while the application is running (Fig. 4, index 3).

2.2.2 *Adding Weaving Support*

In adding support for dynamic aspect weaving to the RVM we had to take into account that methods to be decorated with advice code may already have been compiled by the baseline or optimising compiler. Moreover, they may have been inlined during the compilation of other methods. Below we will show how we addressed these issues for class-wide, instance- and thread-local aspect deployment.

```
REC = {m_0};
M = REC;
do
    M' = ∅;
    foreach m ∈ M do
        M' ∪= inline_locations(m);
    M = M' \ M;
    REC ∪= M;
until M = ∅;
```

Figure 5: Algorithm to determine the set of recompilation candidates.

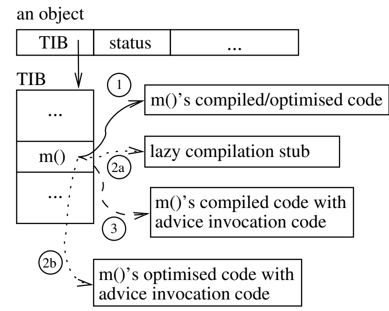

Figure 6: Deployment of a class-wide aspect.

Aspect weaving in Steamloom is done by first modifying affected methods' bytecodes using BAT and then recompiling them (or scheduling them for lazy recompilation). To expose its methods' bytecodes to BAT's API, each class that is loaded into the VM must be represented in BAT's own format, which we have achieved by modifying the RVM's class loader.

Recompilation of an optimised method always retains the method's optimisation level. Thus, performance improvements gained by optimising it are not lost. Special treatment is needed if a method that is to be decorated with advice code was inlined somewhere by the optimising compiler. In this case, it is not sufficient to simply recompile the method since its native code may be inlined in various places all over the loaded classes. Instead, all inline locations of the method have to be recompiled as well, and all locations where those methods were inlined and so forth, resulting in a cascading recompilation.

We have added a set of `VM_Method`s to each method, every element of which corresponds to an inline location of the method owning the set. Whenever recompilation of an optimised method is due, an algorithm (cf. Fig. 5) retrieves the set of methods that also need to be recompiled due to inlining, and all such methods are immediately recompiled.

Starting from the method m_0 that is decorated with advice code the algorithm finds all methods that need to be recompiled and stores them in the set REC. The function $inline_locations$ returns, for a given method m, all methods where m is directly inlined. The algorithm follows a generational approach, where m_0 forms generation 0, and all methods that directly inline a method from generation k belong to generation $k+1$. If a method m is inlined both in methods of generations a and b where $a < b$, we define m to have generation b to avoid multiple inline location retrieval operations. Methods of generation k are stored in the set M. The inline locations of all methods in M are stored in M'. Next, all methods of generation $k+1$ are added to REC (these are the methods that are found in M' but do *not* belong to generation k). As soon as an empty generation $k+1$ is retrieved, the algorithm terminates.

2.2.2.1 Class-Wide Aspects.

An aspect's `deploy()` and `undeploy()` methods trigger all actions needed to activate or deactivate the aspect by iterating over all affected methods and appropriately changing their bytecodes: if an aspect comprises, e. g., a `before()` advice, code for invoking it is prepended to the affected methods' bytecodes. The affected classes' constant pools are updated automatically if needed. In any case, affected methods have to be recompiled for advice to take effect.

This is straightforward for baseline-compiled methods: to trigger recompilation the lazy compilation stub is reinstalled by having the corresponding JTOC and TIB entries point to it. Thus, baseline-compiled methods are invalidated and marked for recompilation to take place automatically the next time they are invoked, as with normal baseline compilation. This is illustrated in Fig. 6. At first (Fig. 6, index 1), the method pointer from the TIB points to the compiled code. Once the aspect is deployed, the method is invalidated and its code pointer now references the lazy compilation stub (index 2a). As soon as the method is invoked after that, the lazy weaving stub is executed (index 3).

If a method has been compiled by the optimising compiler due to an AOS controller decision, aspect deployment logic does not reinstall the lazy compilation stub but immediately recompiles the method at the optimisation level it was previously compiled at (Fig. 6, index 2b). To deal with a cascading recompilation due to inlining, the aforementioned algorithm is used.

2.2.2.2 Instance-Local Aspects.

Usually only one TIB exists per class and is pointed to by all instances of that class. Furthermore, only one instance of `VM_Method` exists for every implemented method. When an aspect is deployed that affects only a particular instance of a class, modifying this single method is not feasible since it would affect the whole class. Instead, Steamloom clones the affected object's TIB and changes the object's TIB pointer to reference the clone. The `VM_Method` object in question is also cloned and the advice is registered for that clone. If the method was baseline-compiled, the respective method pointer in the cloned TIB is set to point to the lazy compilation stub.

In Fig. 7, the instance-local deployment of an aspect is illustrated. Initially, the TIB pointers of both objects `o1` and `o2` reference the same TIB since they are instances of the same class (index 1). If the affected method is baseline-compiled, the TIB is cloned upon deployment of the aspect on the object `o2` and its respective entry is changed to point to the lazy compilation stub (indices 2, 2a). Upon the first invocation of the affected method on `o2`, the stub is executed, compiles the method for this instance and lets the TIB entry point to the decorated code (index 3).

If the affected method was compiled with optimisations, clones of the method and of the TIB are created just like for baseline-compiled methods, and, as in the case of class-wide aspects, the method is immediately recompiled at the

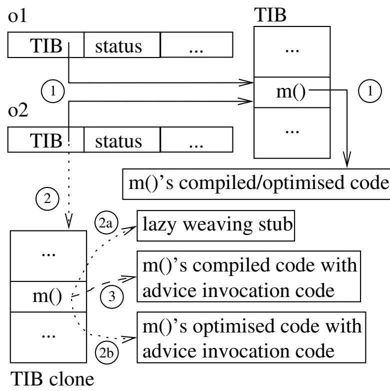

Figure 7: Deployment of an instance-local aspect.

appropriate optimisation level (indices 2, 2b).

One problem remains if the decorated method was inlined. In case of instance-local aspects, this is harmful because virtual method dispatch is performed via the object's TIB. An inlined method is *not* looked up via the TIB: its native code is directly executed in place. Thus, if a method is affected by an instance-locally deployed aspect, it *must not be inlined*. In Steamloom, every method stores a flag that is set to `true` if an instance-specific version of that method exists. Upon recompilation, inlining is forbidden for such methods. This *only* includes methods affected by an *instance-locally* deployed aspect. For class-wide aspects, no restrictions are imposed on inlining.

Concerning cascading recompilation, the following procedure is followed in case of instance-local deployment. The decorated method, m_0 (which belongs to generation 0 in the terminology of the algorithm in Fig. 5), is not inlined because it is specific not only for its class, but also for a given instance. All methods inlining m_0 (i. e., methods from generation 1 and above) may themselves be subject to inlining in other places. For them, inlining is not forbidden because their implementations do not have to be dispatched by instance.

2.2.2.3 Thread-Local Aspects.

To support thread safety, a brief snippet of code is inserted before every call to advice functionality. This code checks the thread identity and skips the advice invocation if the respective aspect is not meant to be active in the current thread. This checking code is inserted only at join points that belong to a thread-locally deployed aspect.

3. EVALUATION AND FUTURE WORK

In this section, we will report on some performance measurements we have made and will discuss some observations with the current implementation, as well as some thoughts about steps to be taken in the future.

3.1 Performance Impacts

We have evaluated Steamloom along the following criteria:

- the cost of the implementation measured in terms of lines of code needed for extending the RVM to implement Steamloom,
- the performance penalty caused by the extension implementing Steamloom on top of the RVM,
- performance of Steamloom-supported dynamic crosscuts,
- the impact on the size of generated native code.

Benchmark	Relative Performance Overhead		
	RVM/Sun	SL/RVM	SL/Sun
compress	129.9 %	94.5 %	122.9 %
jess	216.5 %	106.7 %	231.1 %
db	96.1 %	102.5 %	98.9 %
javac	233.9 %	111.6 %	261.3 %
mpegaudio	225.9 %	99.2 %	235.6 %
mtrt	276.8 %	106.6 %	295.1 %
jack	240.1 %	104.3 %	250.7 %
average	**202.7 %**	**103.6 %**	**213.7 %**

Table 1: Performance measurement results.

Extending the RVM with dynamic weaving functionality resulted in adding about 500 new lines of code to the VM itself. Steamloom API and data structures add another 1400 lines. Lines solely containing comments, curly braces or whitespaces were not counted.

To measure performance, we have run the SPECjvm98 benchmarks on three VMs. We have compared Steamloom to an unmodified build of the RVM version 2.2.1, which is also the version used to implement Steamloom. Moreover, we have compared both Steamloom and the RVM to Sun's HotSpot Server VM version 1.4.2_01. The benchmark problem sizes were 100 (maximal) in all cases, and all benchmarks were run 20 times. All performance tests were run on a 2,6 GHz Pentium IV Linux machine with 1 GB memory. Both the RVM and Steamloom were compiled to use the baseline compiler for creating the boot image, the AOS as run-time compiler infrastructure and the semi-space garbage collector. Assertion checking was disabled. Due to BAT's memory requirements (cf. Sec. 2) we have doubled the heapsize to 512 MB. The results were obtained by computing averages and are summarised in Table 1.

The RVM performs about 103 % slower than HotSpot. The overhead incurred by Steamloom is about 4 %, as compared to the unmodified RVM. This overhead results from additional operations Steamloom performs at class-loading time (BAT data structures have to be created) and from recording inlining information during optimised compilation.

We will now evaluate the performance of applications running on Steamloom as compared to the same applications implemented with AspectJ. We have run the AspectJ implementation on HotSpot also to relate Steamloom's performance to that of a production VM.

For comparison we have used the Fibonacci application introduced in Sec. 1 (cf. Fig. 1). We have chosen this application because it gives the VMs good opportunities to inline, which in turn urges Steamloom to recompile inlined methods. Moreover, the application spends much time calling the short `fib()` method which allows for measuring the cost of `cflow` and programmatic deployment rather than that of some long-running method. The counting aspect from Fig. 2 is used in a static and a dynamic form using `cflow` to decorate the application.

The results of running different versions (with respect to decoration with aspects) of the test application are collected

in Table 2. The times displayed in the "Fibonacci application" section are average execution times of `m1()` and `m2()` (they were run 10 times in each case). The table shows the numbers from three comparisons. First, the application was run with no aspect deployed, and next with the counting aspect statically deployed; i. e., *all* calls to `fib()` lead to a counter increment and the counter is reset and output at the beginning and end of *each* execution of `fibstart()`. In the third run, the aspect was deployed in a way that decorates only `m2()`'s control flow with counting functionality (cf. Sec. 1, Fig. 3). The table shows that an undecorated version of the test application runs at almost exactly the same speed on both the RVM and Steamloom, which was to be expected given the numbers in Table 1.

Static aspect deployment is achieved in Steamloom by deploying the aspect before the application does anything else. The corresponding column from Table 2 shows that the statically decorated application runs at roughly the same speed on both the RVM with AspectJ and Steamloom. There is a small deployment overhead in Steamloom, and the actual execution times of `m1()` and `m2()` are slightly lengthened due to a different implementation of advice instance retrieval. However, the bytecode generated by Steamloom does not differ much from the code output by the AspectJ compiler: both insert into the original code first a call to a static method that returns the instance on which advice functionality is to be invoked, and next a virtual method call to that functionality. Steamloom's deployment operation does, in this case, not comprise any recompilation; it only modifies the affected methods' bytecodes. Recompilation does not occur because weaving takes place before any of the decorated methods have been called and therefore have not yet been baseline-compiled.

Now, let us compare explicit programmatic deployment with Steamloom to the performance of dynamic crosscutting with AspectJ's `cflow`. For the AspectJ version of the measurement application, we have used the counting aspect from Fig. 2 in its original form using `cflow`. In the Steamloom version, we have implemented the CAESAR example from Fig. 3 as follows: an aspect that decorates `m2()` with a `before()` and an `after()` advice is deployed *statically.* This aspect's `before()` advice *thread-locally* deploys the actual `fib()` invocation counting aspect, while its `after()` advice undeploys it. That way, the counting functionality is active only in the control flow of `m2()`, and only in the thread that has invoked this method.

The numbers in column 4 of Table 2 show significant differences between AspectJ on both HotSpot and the RVM (which is seemingly not very well suited for managing `cflow`) and Steamloom. Steamloom again has a small deployment overhead which results from statically decorating `m2()`. On the other hand, AspectJ strongly suffers from a permanent overhead due to the maintenance of an execution stack. Steamloom does not need an execution stack to achieve control flow specific activation of aspects; it works by dynamically recompiling methods that are affected by aspect deployment. Naturally, the AspectJ application performs much better on HotSpot than on the RVM, but it has to be noted that Steamloom performs even faster, unlike the "no aspect" and "static deployment" cases, where it had an overhead of 2.5 and 2.8 times, respectively. Obviously, `cflow` management is expensive.

We have also measured the performance of crosscuts not supported by dedicated pointcut designators. To do this, we have added an application-wide aspect that counts method invocations in the "db" benchmark from the SPECjvm98 suite. The set of methods whose invocations are counted is, however, not determined statically but at run-time. In the AspectJ case, we have used an aspect with an advice that checks, using a `HashSet` containing method names, at *every* method invocation whether the counting functionality applies to the current method. For the two extremes of counting *all* and *no* method invocations, we have executed the benchmark 20 times at problem size 100 and computed the average execution time. The "SPEC db benchmark" section of Table 2 shows the results. Regardless of being run on the RVM or HotSpot, the AspectJ implementation suffers from the high overhead of explicitly looking up methods in a data structure, while Steamloom's performance advantage is due to the fact that only those methods whose invocations are to be counted are actually decorated with advice code.

The cost of cloning TIBs and `VM_Method`s in the case of instance-local aspect deployment was not measured. It can, however, be expected to be very low as both TIBs and method representations are very small when, e. g., compared to a class constant pool. In the standard case, a TIB exists exactly once per class, just like a `VM_Method` instance exists once per method. Cloning such objects does not create a considerable memory overhead.

To weave advice into the base code Steamloom modifies the bytecodes of only those methods that include join points selected by the corresponding pointcut. This happens just before the bytecode is passed to the JIT compiler. If no advice are active for a method the unmodified bytecode is passed to the compiler. As the compiler is not modified in Steamloom the size of generated native code for an application run on the RVM and on Steamloom is the same.

We have not compared the bytecode modifications to AspectJ's output in detail. From a short evaluation, however, we expect that the bytecode produced by Steamloom's weaving mechanism is very similar to that produced by AspectJ.

3.2 Discussion and Future Work

Steamloom currently can decorate method executions with `before()` and `after()` advice that can be registered for a whole class or for single instances, local to certain threads or spanning all threads of the application. Steamloom's performance impact on running applications is low, whether aspects are deployed or not. The performance measurements we have presented underpin our claim that a tight integration of support for dynamic join points with the underlying virtual machine allows for efficiently supporting dynamic join points.

Performance measurements have also shown that an even tighter integration of such support is strongly desirable. We have increased the heap size because of the additional memory required by the bytecode toolkit BAT that Steamloom uses to modify method bytecodes. Because BAT is not as tightly integrated with the RVM as would be desirable, a copy of each class's constant pool must be built when the class is loaded to create a BAT constant pool representation from it (BAT works on its own representation of class files). If the bytecode manipulation framework was instead tightly integrated with the VM, the partly-redundant storage of class representations would become unnecessary. We are convinced that integrating dynamic join point logic more

	Fibonacci application			SPEC db benchmark	
	no aspect	static	`cflow`/dynamic	count all	count none
AspectJ (on Sun VM)	m1: 1,271 ms m2: 1,259 ms	m1: 1,293 ms m2: 1,283 ms	m1: 7,086 ms m2: 15,091 ms	16.42 s	16.112 s
AspectJ (on RVM)	m1: 3,128 ms m2: 3,129 ms	m1: 3,362 ms m2: 3,275 ms	m1: 51,822 ms m2: 51,824 ms	19.935 s	19.456 s
Steamloom	m1: 3,153 ms m2: 3,116 ms	(deploy: 165 ms) m1: 3,636 ms m2: 3,579 ms	(stat. deploy: 134 ms) m1: 3,145 ms m2: 3,693 ms (dyn. deploy: 74 ms) (dyn. undeploy: 22 ms)	12.629 s	12.733 s

Table 2: Results of the comparison of weaving with Steamloom and AspectJ.

closely with the actual VM execution logic—e. g., just-in-time compilers—will eliminate the need for bytecode manipulation. This will be the main focus of our future work.

Steamloom currently does not support `around()` advice. We will certainly strive to support the full power and flexibility this kind of advice offers. However, the important `proceed()` statement known from AspectJ, being no part of the Java language, has no representation in bytecode. To add native support for `around()` advice, we are thinking about entirely replacing methods with dynamically linked versions of them that contain the `around()` advice code.

To implement Steamloom, we have mostly exploited two concepts: selective recompilation of methods and modification of virtual method tables in the form of TIBs. To actually introduce support for dynamic join points in production-grade VMs, we will approach an implementation of Steamloom's features in the HotSpot VM which also allows for dynamic recompilation of methods [5]. We expect the performance of dynamic join point support to be very high.

4. RELATED WORK

In this section, we will discuss other approaches to dynamic crosscutting, especially those that enable unbound dynamic crosscuts. The discussed approaches are classified into three categories. Systems that modify application code at compile-time or load-time fall into the first category. The second category is made up of systems that monitor and intercept execution at run-time, but do not modify any code, while systems that actually *interweave* application and aspect code at run-time form the third category. We will present examples for each category[2] and briefly discuss each approach. Finally, a short discussion of the recently introduced notion of continuous weaving in relation to programmatic deployment is given.

4.1 Pre-Run-Time Instrumentation

There is a class of dynamic AOP implementations for Java like EAOP [7, 6], JAC [22, 15], JBoss AOP [16] and PROSE 2 [24] that modify the application's classes either before compilation using a preprocessor (EAOP), as they are loaded into the VM (JAC, JBoss AOP) or as their bytecode is about to be compiled by the just-in-time compiler (PROSE) so that the classes meet the requirements of the underlying AOP infrastructure. All four examples insert hooks and/or wrappers at join points, thus making the AOP infrastructure aware of them.

[2]We do not claim the list to be complete.

EAOP's model is different from that of JAC and JBoss AOP: it is based on regarding a running application as a sequence of events that are fired whenever execution reaches a join point [7, 6, 9]. EAOP features event composition, allowing for the definition of "composite" pointcuts that consist of join points occurring at different points in execution time.

PROSE 2 [24] uses a modified version of the RVM's baseline compiler to insert code that checks for the presence of advice at every possible join point. There is actually not much of a conceptual difference between this approach and EAOP's eager event generation: hooks are inserted and called at every point that *may be* a join point regardless of whether there is advice code associated with it or not. Eagerly inserting hook calls into native code introduces a considerable performance overhead and is a conceptually unsatisfying approach because it does not avoid unnecessary checking operations. While having a low overhead as long as no aspects are woven, PROSE 2 suffers from large performance penalties when aspects are activated: e. g., decorated virtual method calls are slowed down up to 8.8 times [24]. Also, it produces application code twice the size of code produced by an unmodified build of the RVM in any case [24], which is due to the eager decoration of every possible join point with calls to the weaving logic.

Instrumenting classes before their code is actually run and leaving them in that state afterwards, as JAC and JBoss AOP do, also leads to some disadvantages. There are basically two possible ways to perform this instrumentation: either *all* possible join points are decorated with hook method calls, or only a *specified set* of join points is. In the first case, comparatively large performance overheads are the result. In the second case, it is impossible to extend or reduce the set of "aware" join points at run-time—although single given join points may be riddened of their decoration, no new join points can be activated.

4.2 Run-Time Event Monitoring

Another class of dynamic AOP systems, represented by PROSE 1 [23], is conceptually a close relative to the EAOP system mentioned above. PROSE 1 also regards a running application as a sequence of events. The main difference lies in that PROSE 1 does not instrument any code but instead intercepts execution and branches to advice code whenever an "activated" join point is encountered. The system utilises the JVM's debugging facilities [18, 19] to generate events at certain points during application execution and intercept execution there.

This implementation—and probably any other implemen-

tation treating events as first-class entities of the run-time environment—suffers from overheads owed to event generation and processing logic. The running application has to be permanently monitored by the run-time environment.

4.3 Run-Time Weaving

The class of dynamic weaving systems that perform actual weaving operations entirely at run-time is here represented by Wool [26], a .NET-based approach by Schult and Polze [27] and AspectS [13].

Wool [26] uses a hybrid approach. It monitors an application's execution using the JPDA like PROSE 1, but as soon as an activated join point has been reached a sufficient number of times, calls to advice code are directly woven into the affected application code for performance reasons. This is done using the HotSpot VM's HotSwap technique [5] which allows for changing methods' bytecodes while an application is running in a debugger. Basically, Wool has a lot in common with PROSE 1, yet it enhances the performance of advice code invocation: for example, it slows down the execution of virtual method invocations only four times [26] in the presence of activated aspects.

Schult and Polze's system [27] allows for defining aspects to be interwoven with an object's code at instantiation time. This is done by dynamically creating a subclass of the class the object originally belongs to, and letting the newly created object be an instance thereof. The subclass represents the original class with interwoven aspect functionality. Conceptually, this approach is close to our implementation using TIB clones. Although instance decoration is supported, there is one major drawback: decoration happens at instantiation time and is not revertible. It is not possible to undeploy an aspect while the object is "alive" or to deploy an aspect on an already "living" object.

AspectS [13, 3] is an extension of Squeak [14, 28] with dynamic weaving capabilities. Due to the ease of accessing the language's meta level in Smalltalk, it is comparatively simple to implement dynamic aspect-orientation support for such a system. In AspectS, aspects are deployed and undeployed by sending `install` and `uninstall` messages to instances of aspect classes. The technique that is used is again that of wrapping decorated methods in dynamically added methods that check for the presence of applicable advice code and execute the latter where necessary. AspectS even supports instance decoration by using so-called *advice qualifiers* that determine if a piece of advice code must be applied to a given instance. This construct is just a wrapper for an explicit object identity check. Although AspectS obviously supports dynamics to a degree that goes beyond the capabilities of the other presented systems, its features are still implemented at language-level and are not an inherent feature of the underlying VM.

4.4 Continuous Weaving

Approaches discussed so far exploit a *complete weaving* model. Complete weaving, as it is e.g., implemented in AspectJ, creates shadows for all potentially matching join points at once before run-time. In [11], the notion of *continuous weaving* is introduced. With continuous weaving, dependencies between join points are taken into consideration to postpone creating shadows for some join points only after other join points the former depend on are reached. For example, shadows for the `otherPCD` part of `cflow(aPCD) && otherPCD` are created on the fly only after `aPCD` has matched.

A comparison of two versions of AspectS [13, 3], one exploiting the complete weaving model and the other exploiting continuous weaving, indicates the performance advantage of the latter [11]. However, in that implementation, the continuous weaving logic is part of the application code, which is why the performance gain is lost when compared to AspectJ complete weaving. Here, we describe support for continuous weaving as enabled by programmatic deployment of CAESAR at the VM level and evaluates it by a direct comparison to AspectJ—the most efficient approach today.

	aspect granularity		thread	dynamic
	class	instance	locality	crosscuts
AspectS	X	(X)		yes
EAOP	X			quasi
JAC	X			quasi
JBoss AOP	X			quasi
PROSE 1	X			yes
PROSE 2	X			quasi
Schult/Polze	(X)	(X)		see text
Wool	X			yes

Table 3: Overview of dynamic weaving systems.

4.5 Concluding Remarks

Table 3 shows how the presented systems support dynamic crosscuts. An X stands for "supported", an "(X)" denotes a reduced form of support for the respective feature. "Quasi" in the column "dynamic crosscuts" refers to emulation of dynamic crosscuts by intercepting the running program at every possible join point.

None of the systems fully support both class and instance decoration, although the latter can be simulated by explicitly checking object identity at join points; AspectS even provides a wrapper for this approach. However, this is unsatisfying as instance decoration should be possible by means of the system itself, not as a workaround. The only system that directly supports instance decoration, Schult and Polze's, has a major problem: aspect decoration is specified at object creation time and can not be reversed.

None of the systems directly support thread-local activation of aspects. In all of them, it is possible to emulate this feature by explicitly checking for thread identity in advice code. That is, however, unsatisfying as it is neither integrated with the execution model nor supported by the run-time environment but must be done by the advice programmer. Steamloom shifts support for thread-local deployment into its API, just like AspectS does for instance-local decoration. That way, the overhead of dealing with thread locality is not imposed on the advice programmer but can easily be dealt with at deployment time.

Systems that support unbound dynamic crosscuts by inserting checks at every possible join point (EAOP, PROSE 2, JAC, JBoss AOP) have the inherent disadvantage that a big performance penalty has to be paid, mostly even if no aspect is in use at all. On the other hand, the systems that do so while preserving modular structure beyond compilation either use the JPDA API (PROSE 1, Wool) or the Smalltalk meta-object protocol (AspectS), both of which cannot deliver industry-strength performance.

5. SUMMARY

We have presented Steamloom, an implementation of dynamic join point support as an extension to IBM's Jikes Research Virtual Machine. Steamloom supports the RVM's adaptive optimisation system, allowing for efficient execution of programs decorated with aspects, which is shown by our performance measurements. Comparing Steamloom to other implementations of dynamic crosscuts has shown that a tight integration of dynamic AOP support with the underlying VM itself boosts performance: Steamloom's execution speed reveals no major performance overheads when compared to equivalent static aspect-oriented software, it executes considerably faster in case of dynamic join points, especially those that are statically unbound.

Future work will focus on extending the set of supported primitive join point types, which is fairly restricted by now. Furthermore, we will work on an even tighter integration of dynamic weaving with the VM. Another interesting path will be to approach an implementation of Steamloom's features in the HotSpot VM which also allows for dynamic recompilation of methods [5]. We expect the performance of dynamic join point support to be very high.

Acknowledgements

We would like to thank Jörg Baumgart and Jan Vitek for their valuable comments concerning thread safety. Moreover, we wish to thank the anonymous reviewers for their suggestions.

6. REFERENCES

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and F. Sweeney P.˙Adaptive Optimization in the Jalapeño JVM. In *OOPSLA 2000 Proceedings*, pages 47–65. ACM Press, 2000.

[2] M. Arnold, M. Hind, and G. Ryder B.˙Online Feedback-Directed Optimization of Java. In *OOPSLA 2002 Proceedings*, pages 111–129. ACM Press, 2002.

[3] AspectS Home Page. `http://www-ia.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS/`.

[4] BAT Home Page. `http://www.st.informatik.tu-darmstadt.de/pages/projects/BAT/`.

[5] M. Dmitriev. Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications. In *Workshop on Engineering Complex Object-Oriented Systems for Evolution, Proceedings (at OOPSLA 2001)*, 2001.

[6] R. Douence, O. Motelet, and M. Südholt. A Formal Definition of Crosscuts. Technical Report 01/3/INFO, École des Mines de Nantes, 4 rue Alfred Kastler, 44307 Nantes cedex 3, France, 2001.

[7] R. Douence and Mario Südholt. A Model and a Tool for Event-Based Aspect-Oriented Programming (EAOP). Technical Report 02/11/INFO, Ecole des Mines de Nantes, 2002.

[8] EAOP Home Page. `http://www.emn.fr/x-info/eaop/`.

[9] R. E. Filman and K. Havelund. Source-Code Instrumentation and Quantification of Events. In G. T. Leavens and R. Cytron, editors, *FOAL 2002 Workshop (at AOSD 2002)*, pages 45–49, 2002.

[10] J. Hallpap. Towards Caesar: Dynamic Deployment and Aspectual Polymorphism. Diploma thesis, Software Technology Group, Darmstadt University of Technology, 2003. `http://www.st.informatik.tu-darmstadt.de/database/theses/thesis/DiplomaThesis.pdf?id=15`.

[11] S. Hanenberg, R. Hirschfeld, and R. Unland. Morphing Aspects: Incompletely Woven Aspects and Continuous Weaving. In *AOSD 2004 Proceedings*. ACM Press, 2004.

[12] E. Hilsdale and J. Hugunin. Advice Weaving in AspectJ. In *AOSD 2004 Proceedings*. ACM Press, 2004.

[13] R. Hirschfeld. AspectS - Aspect-Oriented Programming with Squeak. In M. Aksit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, volume 2591 of *LNCS*, pages 216–232. Springer, 2003.

[14] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future: the Story of Squeak, a Practical Smalltalk Written in Itself. In *OOPSLA 1997 Proceedings*, pages 318–326. ACM Press, 1997.

[15] JAC Home Page. `http://jac.aopsys.com/`.

[16] JBoss AOP Home Page. `http://www.jboss.org/developers/projects/jboss/aop.jsp`.

[17] The Jikes Research Virtual Machine. `http://www-124.ibm.com/developerworks/oss/jikesrvm/`.

[18] Java Platform Debugger Architecture Home Page. `http://java.sun.com/j2se/1.4.1/docs/guide/jpda/index.html`.

[19] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[20] G. Burke M. J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, J. Serrano M. C. Sreedhar V. H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *Java Grande 1999 Proceedings*, pages 129–141. ACM Press, 1999.

[21] M. Mezini and K. Ostermann. Conquering Aspects with Caesar. In *AOSD 2003 Proceedings*. ACM Press, 2003.

[22] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In A. Yonezawa and S. Matsuoka, editors, *Reflection 2001 Proceedings*, volume 2192 of *LNCS*, pages 1–24. Springer, 2001.

[23] A. Popovici, T. Gross, and G. Alonso. Dynamic Weaving for Aspect-Oriented Programming. In G. Kiczales, editor, *AOSD 2002 Proceedings*. ACM Press, 2002.

[24] A. Popovici, T. Gross, and G. Alonso. Just-in-Time Aspects. In *AOSD 2003 Proceedings*. ACM Press, 2003.

[25] PROSE Home Page. `http://ikplab11.inf.ethz.ch:9000/prose/`.

[26] Y. Sato, S. Chiba, and M. Tatsubori. A Selective, Just-in-Time Aspect Weaver. In F. Pfenning and Y. Smaragdakis, editors, *GPCE 2003 Proceedings*, volume 2830 of *LNCS*, pages 189–208. Springer, 2003.

[27] W. Schult and A. Polze. Dynamic Aspect Weaving with .NET. `http://www.dcl.hpi.uni-potsdam.de/`

dcl/papers/GI2002.ps.
[28] Squeak Home Page. http://www.squeak.org/.