Virtual Machine Support for Aspects with Advice Instance Tables

Michael Haupt — Mira Mezini
Software Technology Group, Darmstadt University of Technology
Hochschulstrasse 10
D-64289 Darmstadt
Germany
{haupt,mezini}@informatik.tu-darmstadt.de

ABSTRACT. The recent adoption of dynamic aspects in middleware containers calls for investigating the cost of introducing such support. With regard to weaving, most approaches to dynamic join points either employ compile-time or load-time insertion of hooks and wrappers at potential join point shadows, or exploit interfaces of run-time environments. With regard to advice instance management, frequently language-level constructs are used to store advice instances. While support for dynamic join points using these techniques is easily implemented, it is likely to be inefficient when dynamic aspects ask for dedicated support from the VM. We have developed the concept of advice instance tables that allow for a very efficient and highly flexible lookup of advice instances. We present the concept and discuss its performance as compared to that of advice instance lookup in other approaches.

RÉSUMÉ. L’adoption récente de la programmation par aspects dans les conteneurs intergiciels exige d’étudier le coût de cette introduction. En ce qui concerne l’opération de tissage, la plupart des approches utilisant des points de jonction dynamiques, opèrent par l’insertion de crochets ou d’encapsulateurs dans le code de base au moment de la compilation ou du chargement des classes ou exploitent directement les interfaces de l’environnement d’exécution. D’autre part, des constructions fournies par le langage de programmation sont utilisées pour enregistrer les instances d’un advice. Bien que la mise en œuvre de points de jonction dynamiques par les techniques sus-citées est simple, elles s’avèrent inefficaces lorsque les aspects nécessitent un support de la part de la machine virtuelle. Nous avons développé le concept de tables d’instances des advices qui permettent une recherche très efficace et fortement flexible des instances d’un advice donné. Dans cet article, nous présentons le concept des tables d’instance d’un advice et discutons ses performances par rapport à celles de la recherche d’advice utilisée dans d’autres approches.

KEYWORDS: dynamic aspects, advice instance management, VM integration, performance.

MOTS-CLÉS : aspects dynamiques, gestion d’instances d’advices, intégration d’AOP à la machine virtuelle, performance.
1. Introduction

The popularity of aspect-oriented programming (AOP) is growing quickly. For example, in middleware containers, aspects are used to implement interceptor infrastructures or container services (JBo, n.d.b; Bea, n.d.). Companies like IBM announce to employ and support AOP (Sabbah, 2004). Still, now that AOP is beginning to be employed in large-scale applications, more attention has to be paid to its tools’ implementation approaches, and a careful review of implementation techniques is at hand. In doing that, there are two central facets to be considered, namely weaving and advice instance management. We use the term “weaving” to denote the process of integrating the functionality of aspects at join points, without making any commitment on the technique used to implement this process. We use the term “advice instance” to refer to an object advice methods are executed on, while “advice instance management” concerns the association of such objects with the classes and objects affected by aspects.

An important question to ask with regard to implementation techniques for AOP languages is whether native support of the aforementioned facets needs to be integrated into run-time environments. We observe that, with regard to weaving, the majority of existing AOP systems for the Java programming language are implemented as preprocessors of source- or bytecode or as plug-ins or add-ons to the JVM. With regard to advice instance management, existing AOP implementations frequently use language-level constructs to associate objects or entire classes with advice instances. In the following, we will elaborate more on these observations.

With respect to weaving one category of approaches which is represented, e.g., by AspectJ 1.2 (Asp, n.d.a; Kiczales et al., 2001) employs compile-time or load-time weaving, simulating the semantics of dynamic join points by means of conditional logic added at so-called join point shadows, locations in code whose execution might yield a join point at run-time (Hilsdale et al., 2004; Masuhara et al., 2003). This approach to simulating dynamic join points is taken to its extreme by some of the so-called dynamic AOP approaches, which characterise a “dynamic” aspect by its ability to be deployed and undeployed while the application it cuts across is running, without the need for shutting the application down for recompilation. These systems support cases where the set of potentially affected locations in code cannot be determined until the application is actually running.

Dynamic weaving is currently mostly implemented by either decorating potential join point shadows with hooks or wrappers or by selective recompilation. Hooks and wrappers are method calls to some dynamic AOP infrastructure that checks a repository for the presence of advice that have to be invoked. Those calls are normally inserted into application classes at load-time. Selective recompilation is done after the bytecodes of methods affected by a weaving operation – i.e., methods that, after aspect deployment, have to call advice functionality – have been modified accordingly.

JBoss AOP (JBo, n.d.a), JAC (JAC, n.d.; Pawlak et al., 2001) and AspectWerkz (Asp, n.d.b) are representatives of approaches that insert hooks or wrappers at po-
tential join point shadows at load-time. PROSE (PRO, n.d.) also has the flavour of a wrapper-based approach, with the important difference that no instrumentation of join point shadows is done; instead, the execution is intercepted at join point shadows through debugger breakpoints. Each of these systems is implemented as an extension to a run-time environment, utilising pre-existing tool interfaces: PROSE relies on debugger breakpoints, AspectWerkz modifies classes through Hot-Swap (JPD, n.d.; Dmitriev, 2001), and JBoss AOP and JAC employ modified class loaders. We therefore classify these systems as plug-ins or add-ons.

Reflex (Tanter et al., 2003; Tanter et al., 2004) also is to be classified among the three aforementioned systems. It applies modifications to the base application’s code at load-time like AspectWerkz and JBoss AOP, but its entire approach is different from those of the latter two. Reflex is based on the partial behavioural reflection (Tanter et al., 2003) flavour of meta-programming, which aims at saving both memory and execution time by reifying meta-objects only when needed. Reflex weaves into the base application’s code calls to its meta-level, where dedicated meta-objects control the execution of the base application. Advice are implemented as methods in meta-object classes. Reflex allows for controlling meta-object reification through so-called activation conditions that are evaluated before a link from the base application to its meta-level is actually established.

Regarding advice instance management, the standard case in most existing AOP implementations is that there is exactly one instance of the advice (or of the aspect class, respectively)—the aspect is a singleton. This approach is satisfactory for static environments, where the aspect’s state is of global interest. However, it is often the case that an aspect’s state needs to exist once per single instance of the affected class. In this case, multiple instances of the aspect (multiple advice instances, respectively) need to exist; one for each instance of the affected class. For example, AspectJ allows for controlling aspect instantiation via its perthis and pertarget constructs. We call such aspects per-instance aspects. Next, an aspect may even be intended to affect single instances only, while entirely leaving all other instances of the same class alone. This is imaginable in contexts where single objects dynamically adopt and/or abandon certain roles or are subject to special treatment (e.g., in trusted environments).

In the three cases mentioned above, we observe different degrees of granularity in the association of aspect instances to application objects. In the first two cases, aspects affect entire classes. However, the second case asks for a more fine-grained association control in that advice affecting a certain method are executed in a different context (advice instance) depending on the sender/receiver of the affected join points. In the third case, the aspect is applied to individual instances only. In systems that support such advanced aspect instantiation mechanisms, it is not enough to associate advice instances with entire classes; it must be possible to associate them with single objects as well.

As we will show in sections 4 and 5, existing AOP implementations frequently use language-level constructs to associate single objects or classes with advice instances, introducing a performance penalty due to explicit checks for object identity at join
point shadows or in advice code. Selectively compiling dedicated versions of affected methods for decorated objects is a possible solution that is however hard to achieve using HotSwap or related techniques. After all, the underlying run-time environment itself – the JVM – does not inherently support method versions for single objects.

It is our conviction that, for efficiency reasons but also for better support of debugging and separate compilation, support for AOP needs to be tightly integrated with execution environments such as JVMs. In previous work (Bockisch et al., 2004), we have shown that a VM-integrated approach employing selective recompilation is not only feasible but efficient as well. The “Steamloom” VM (Bockisch et al., 2004) is an extension of IBM’s Jikes Research Virtual Machine (Alpern et al., 2000; Jik, n.d.). Extending a VM is a demanding task, yet the modifications applied to the Jikes RVM are well-localised and exploit features that are common to Java virtual machines (Haupt et al., 2005). Therefore, the AOP support we have implemented on top of the Jikes RVM is likely to be achievable in other VMs, like the Sun HotSpot VM, as well.

The previously published versions of Steamloom supported instance-local aspect deployment, but advice instance management was still done at language level. In our ongoing work, we develop and implement new concepts for a more tightly integrated native virtual machine support for dynamic join points. One field of enhancement is the way how advice instances are stored and mapped to classes and objects. This paper presents a novel concept called advice instance tables, which we have implemented in Steamloom. An advice instance table (AIT) is associated with a class or object and maps them to appropriate advice instances.

Because of the tight integration of AITs with the virtual machine on top of which Steamloom was built, advice instances can be retrieved at the speed of directly in-lined array access operations, even in the context of instance-local aspect deployment. Moreover, dynamic weaving and JIT compilation performance both benefit from the AIT concept. These claims are backed by the results of our performance evaluations comparing other dynamic join point approaches to Steamloom.

This paper is organised as follows. In the next section, we present the Steamloom VM. In section 3, we introduce advice instance tables as an enhancement of Steamloom. In section 4, we briefly discuss existing AOP implementations’ strategies for implementing advice instance retrieval, before presenting comparative performance measurements in section 5. Section 6 concludes the paper.

2. Steamloom

In this section, we will introduce our own implementation of dynamic AOP, Steamloom. Before we describe its workings, we will first give an overview of the virtual machine on top of which we have implemented Steamloom. For a more detailed introduction to Steamloom, we refer to previous work (Bockisch et al., 2004).
2.1. The Jikes Research Virtual Machine

The Jikes RVM is an open-source JVM built by IBM (Jik, n.d.) that provides a platform for experimental implementations of JIT compilers, garbage collectors, and so forth. It is almost entirely implemented in Java, and its openness makes it very valuable for extensions. The VM has no interpreter; it is completely based on just-in-time compilation, offering the choice among three compiler systems: a simple “baseline” compiler, an optimising compiler (Burke et al., 1999), and an adaptive optimisation system (AOS) built on top of the other compilers that performs online profiling and optimisation (Arnold et al., 2000; Arnold et al., 2002). The compiler architecture is chosen at VM building time.

Every class loaded in the Jikes RVM is represented by an instance of `VM_Class`. An instance is represented in memory by a concatenation of slots for header and attributes. 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, inter alia, the class’s virtual method dispatch table. Every 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 Figure 1 (index 1). The first time a method is invoked, the stub is executed and inspects the call stack to retrieve the callee object, its class, and the called method. Using this information, the stub retrieves the bytecode from the corresponding `VM_Method`, compiles the method, and updates the according TIB (or JTOC, in case of a static method) entry to point to the compiled code and executes the method (Figure 1, index 2). Next time the method is called, the compiled code is executed.

In the presence of the AOS, every method is initially baseline-compiled. As soon as the AOS controller decides – based on gathered profiling data – that an optimised version of that method will yield a performance improvement, it is recompiled and reinstalled while the application is running (Figure 1, index 3).
2.2. Dynamic Aspects in Steamloom

Steamloom, as a VM extension, works entirely at run-time. No compile-time or load-time steps are involved when decorating an application with aspects. Instead, the programmer can manipulate the application through an API (section 2.3) that provides access to aspect building and deploying functionality. Pointcuts, advice and aspects are modelled as first-class entities. A pointcut is 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 is called on an instance of the `Aspect` class. To Steamloom, an aspect is a container that maps pointcuts to advice. Global aspect deployment takes place by sending `deploy()` to an 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 instance-locally. Steamloom currently does not support per-instance aspect instantiation as provided by AspectJ’s `perthis`.

Steamloom currently supports `before()` and `after()` advice for method execution join points; `around()` advice are being implemented\(^1\). Advice are arbitrary methods that can retrieve information from the join point’s context, *e.g.*, the caller, callee or method parameters. They are encapsulated in `Advice` instances, along with parameter passing information that is set by calling special methods at advice construction time.

Aspect weaving in Steamloom is done by first modifying affected methods’ bytecodes and then recompiling them. For baseline-compiled methods, the lazy compilation stub is reinstalled by having the corresponding JTOC and TIB entries point to it. Thus, such methods are invalidated and marked for recompilation to take place automatically the next time they are invoked. This is illustrated in Figure 2. At first (Figure 2, 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). When the method is invoked after that, the stub is executed (index 3). If a method has been compiled by the optimising compiler due to an AOS controller decision, aspect deployment logic immediately recompiles the method at the optimisation level it was previously compiled at (Figure 2, index 2b): performance improvements are retained. Recompilation of optimised methods is not done via the lazy compilation stub because the latter does not directly support the optimising compiler. Moreover, recompilation has to be done immediately if the affected method was inlined somewhere, because the call site of an inlined method relies on its native code being present.

The case is different with instance-local aspects: when an aspect is deployed that affects only a particular instance of a class, simply modifying the method is not feas-

---

\(^1\) They are a more demanding challenge than `before` and `after` advice because they cannot simply be inserted into existing method bytecodes. They rather replace portions of the code that have to be moved elsewhere while their original context has to be kept up.
ible since it would affect the whole class. Instead, Steamloom exploits the fact that every object carries a TIB reference in its header. 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. Steamloom clones the affected object’s TIB and lets the object reference the clone. The VM Method object in question is also cloned and the clone’s code is modified, so that two versions of the method in question exist: one for unaffected objects, and one for those objects that are in the scope of the deployed aspect. Recompilation takes place as described above.

In Figure 3, instance-local aspect deployment is illustrated. Initially, the TIB pointers of both objects o1 and o2, instances of the same class, reference the same TIB (index 1). Upon deployment of the aspect on o2, both the TIB and the decorated method are cloned and the TIB’s respective entry is changed to point to the lazy compilation stub (in case the method was baseline-compiled: indices 2, 2a, 3) or to an immediately recompiled version of the optimised method (indices 2, 2b).

If a method decorated by an instance-local aspect was inlined in some place, it is not any longer looked up via the TIB: its native code is directly executed in place. In the context of instance-local aspects, at least two versions of a given method exist—the original one and the one that was cloned for the affected instance. Since the optimising JIT compiler cannot decide which version of the method in question is to be inlined (the original, or the instance-local one), inlining is prohibited for such methods. For details, we refer to our previous work on Steamloom (Bockisch et al., 2004).

The case that a method is advised by both class-wide and instance-local aspects is not fully covered in our current implementation. While it is possible to add instance-local advice to a method that is already subject to class-wide advice, support for arbitrarily adding and removing all kinds of (instance-local or class-wide) advice is not yet implemented. Since AIT slots are recycled to avoid unnecessary array growth, mixing class-wide and instance-local advice instance references in one single array per class leads to unclear slot allocations. We envision a solution using object-local AITs for storing advice instances that affect only single objects.

To support thread locality, a 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 belonging to a thread-locally deployed aspect.

2.3. **Steamloom Usage Example**

Assume that, before the execution of the method `C.m()`, an advice shall be executed. This advice is implemented in the method `A.adv()`. In AspectJ syntax, the following code inside an aspect would lead to the dedicated behaviour (the object `a` is meant to be an instance of the class `A`):

```java
before(): execution(void C.m()) {
    a.adv();
}
```

In Steamloom, the code in Figure 4 can be used to achieve the same effect. In the first two lines, the advice is assembled: the method to be called as an advice is retrieved via Java's reflection mechanism, and then an instance of `BeforeAdvice` is created that represents a call to the advice method on the newly created instance of `A`.

The pointcut designator expressing to which join points the advice call is to be applied is created by passing an expression in AspectJ syntax to a dedicated parser that is part of Steamloom. Normally, pointcuts can be created by directly assembling them from single `PointcutDesignator` instances, but using a parser as in the example is more convenient.

In the last three lines, the aspect is actually assembled: first, an instance of the `Aspect` class is created, and then, this aspect is added a mapping of the pointcut to the advice as intended. Finally, the aspect is installed into the application by sending it the `deploy()` message. This code can be placed anywhere in a running application and will immediately install the aspect due to the call to `deploy()`.
Method adviceMethod = A.class.getDeclaredMethod("adv", null);
Advice advice = new BeforeAdvice(adviceMethod, new A());
PointcutDesignator pcd =
    SimpleParser.getPointcut("execution(void C.m())");
Aspect a = new Aspect();
a.associate(advice, pcd);
a.deploy();

Figure 4. Assembling and deploying an aspect in Steamloom

In case the aspect is to affect a single instance of C only, the following code can be used, leading to the advice being executed only when `affectedInstance.m()` is invoked:

```java
C affectedInstance = new C();
a.deploy(affectedInstance);
```

3. Advice Instance Tables

In this section, we will introduce the concept and implementation of advice instance tables (AITs) which we have developed and integrated with Steamloom. AITs provide an extremely efficient lookup mechanism for advice instances for both class-wide and instance-locally deployed aspects. Thread-local aspect deployment is not affected because it takes place at bytecode level, through conditionals.

AITs are an integral part of Steamloom’s execution model and are not apparent at language level. This is the main reason for their superior speed: complex language-level data structures, being part of other dynamic AOP tools’ infrastructures, are implemented at bytecode level, and thus access to them is subject to execution by the virtual machine. AITs, being part of the VM itself, do not suffer from that overhead.

3.1. The Structure of Advice Instance Tables

Advice instances have to be associated with every class (for class-wide aspects) and with single instances (for instance-locally deployed aspects). Recall that, for every single class in the Jikes RVM, there exists a TIB. AITs are arrays of objects to which references are stored in the TIB. This was achieved by extending the TIB data structure by one word containing the AIT reference. AIT creation takes place lazily, i.e., as soon as advice instances are registered for one particular class. Thus, most of the AIT references in the various TIBs are null most of the time. The elements contained in the AIT are references to the various advice instances that are relevant to the respective class in whose TIB the AIT is stored. The AIT is created with a default size and is dynamically expanded if necessary.
Figure 5. Deployment of a class-wide aspect with AITs

Figure 5 displays the situation after the deployment of a class-wide aspect. The aspect affects the implementation of the method \( m() \) of a given class. The AIT field in the class’s TIB references an array of Objects, the first entry of which is the advice instance on which the actual advice method is to be invoked. The new code of the method \( m() \) now contains instructions that load the advice instance from offset 0 of the AIT and invoke the advice method thereon (for details on the new code see below).

If an aspect is deployed instance-locally, a deep copy of the class’s AIT is made for the affected instance during the process of cloning the TIB (section 2.2). That way, advice instances belonging to class-wide aspects can still be looked up, while advice instances local to the object in question are not stored in the class-wide AIT.

Assume that one single instance named \( o2 \) of the class known from Figure 5 is now decorated with an additional aspect that affects the implementation of the method \( n() \) of that class—but only for the instance \( o2 \). The situation after deploying this instance-local aspect is shown in Figure 6. The class’s TIB was cloned and made \( o2 \)’s local TIB. Furthermore, a deep copy of the class’s AIT was created that is now referenced from \( o2 \)’s TIB. Note that the version of the method \( m() \) referenced from the cloned TIB is that of the class, and that the first entry in \( o2 \)’s AIT refers the appropriate advice instance. Thus, the class-wide aspect is still active for the method \( m() \). For the instance-locally decorated method \( n() \), however, a new version was compiled and a new entry in the instance-local AIT was created.

3.2. Compiling Advice Instance Lookups

A previous version of Steamloom used a global static array to store advice instances. Every advice was, at creation time, given a unique ID that served as its array index. During weaving, bytecode sequences were added to the affected methods that called a static method \( \text{getAdviceInstance}() \) and thus retrieved the appropriate advice instance from the array. For instance-local deployment, the array elements contained hash tables mapping the affected objects to advice instances.
With AITs, weaving logic still operates at bytecode level, having to insert bytecodes to look up advice instances in the AIT. To implement such lookups, a new bytecode named \texttt{ait\_load} was added to Steamloom. This bytecode has one parameter, namely the AIT index from which an advice instance has to be loaded to be placed on top of the stack. After that, the object can be subject to method invocation in the usual way. The \texttt{ait\_load} bytecode is only used Steamloom-internally.

Of course, there still has to be some kind of mapping from aspect units to AIT indices. This is achieved through a hash table that uses \textit{aspect units} as keys. An aspect unit is a part of the Steamloom data structures, namely a container associating a given pointcut with an advice and, in case of instance-local or thread-local deployment, with the respective instance or thread. The hash table maps aspect units to AIT offsets. The expensive hash table lookup is performed at JIT compile-time only. This is especially important for instance-local aspects, for which the old Steamloom version had to perform a hash table lookup every time the advice was to be executed.

Both the baseline and optimising compiler were modified to generate appropriate native or intermediate code from the \texttt{ait\_load} bytecode. The baseline compiler generates (for an x86 processor) exactly two machine instructions per \texttt{ait\_load}: one to look up the AIT reference in the TIB, and one to load the advice instance from the AIT. For the optimising compiler, two high-level intermediate instructions (Burke \textit{et al.}, 1999; Whaley, 1999) with the same semantics are generated. Because the \texttt{ait\_load} bytecode is directly transformed into high-level intermediate code, and no further modifications were made to the optimising compiler, all subsequent optimisations that the compiler performs – \textit{e. g.}, copy propagation or instruction reordering (Whaley, 1999; Burke \textit{et al.}, 1999) – are applied to the resulting code.

The brief bytecode snippets in Figure 7 show the different approaches to weaving in Steamloom without (Figure 7a) and with AITs (Figure 7b). In both cases, the code shown is the code needed to invoke a \texttt{void} virtual advice method that
takes no parameters. Steamloom without AITs had to call the static method `getAdviceInstance()`, which was part of the Steamloom infrastructure, to retrieve the appropriate instance for a given advice and, in case of instance-local deployment, an object. The method `getAdviceInstance()` retrieves the advice instance from the global array of advice instances. After that, the advice method can be invoked.

Using AITs, the woven bytecode sequence is significantly shorter, consisting of only two instructions. A considerable advantage of the dedicated `ait_load` bytecode is that its translation is simple as it is a mere shortcut for some machine instructions that are (by the baseline compiler, for an x86 processor) generated as follows:

MOV ECX, [A/IT reference address]
PUSH [ECX + <index>]

Contrariwise, expressing the advice instance lookup in application-level byte codes results in the compilers generating various type checks, guards and array bounds checks that can be avoided for AIT lookups because the Steamloom environment guarantees type safety and array integrity.

The native code first retrieves the AIT reference from the TIB slot where it is stored. In the second step, an array lookup, taking the AIT reference as base address, is performed to retrieve the advice instance.

The address of the AIT reference slot is “hard-wired” into the generated code. This can safely be done because the Jikes RVM (and therefore Steamloom) stores TIBs in a memory area that is called “immortal space”. Objects stored in immortal space are never subject to garbage collection and thus do not change their address, which is why references to such objects can be hard-wired. Note that it is not the AIT which is placed in immortal space; it is the class’ TIB. AITs are, from the memory management perspective, ordinary objects that live on the heap.

4. Related Work

In this section, we will, for various AOP implementations, present an analysis of their approaches to advice instance management. We have chosen some typical representatives of AOP implementations and we do not claim that the list of systems
presented here is complete. For the discussion below, we have investigated those versions of the particular systems that were available at the time of writing the respective analyses.

In describing the systems’ aspect models, we restrict ourselves to describing how pointcuts and advice are associated; other features like introductions are not in the scope of this work. We will briefly outline the cases of class-wide and per-instance aspects and that of instance-locally deployed aspects. Some of the systems do not themselves allow for instance-local deployment. For them, we will outline an implementation that matches the intended semantics. A performance analysis of the systems will be presented in section 5.

4.1. AspectJ

In AspectJ (Asp, n.d.a; Kiczales et al., 2001), class-wide deployment is the standard case. Still, advice instance creation is controllable using the language’s perthis and pertarget modifiers. This way of associating advice (aspect) instances with objects is not instance-local aspect deployment: advice functionality defined by per-instance aspects takes effect at all matching join point shadows, regardless of the currently executing object. The important feature of the mechanism is that all advice code is executed in a context that is specific to particular application objects—namely the aspect instance the respective advice method is invoked on. This allows for creating and controlling aspect state on a per-application-object basis. In contrast to that, in the context of an instance-local aspect, an application class’s methods are decorated with advice functionality only if the methods are invoked on the object(s) affected by the aspect.

AspectJ follows a static weaving approach and does not take dynamic evolution of applications – as can be, e.g., achieved by dynamic weaving – into account. More specifically, the dynamic evolution of single objects is not in the design scope of AspectJ, so this language does not offer instance-local deployment of aspects.

AspectJ aspects are represented as Java classes. Each aspect class has a static method aspectOf() returning the advice instance in the standard case where the aspect is a singleton. For per-instance aspects, the aspectOf(Object) method is generated that returns, for a passed object, the advice instance associated with that object. Internally, this is achieved by storing advice instances directly in instances of the affected class and looking them up at run-time.

Instance-local aspect deployment can be emulated in AspectJ. Figure 8 shows an aspect that executes an advice before every execution of a method C.m(). The aspect decorates single objects or groups thereof, that have to be stored in the internal hash set through the aspect’s addInstance() method. This approach naturally imposes a certain overhead on advice execution, as the set containment check has to be done at

2. AspectJ 1.2, AspectWerkz 1.0, JBoss AOP 1.0, PROSE 1.2.1, and Reflex 3.0 alpha 11.
public aspect InstanceLocal {
    private static Set instances = new HashSet();
    public static void addInstance(C c) {
        instances.add(c);
    }

    pointcut affected(C c) : target(c) && if(instances.contains(c));

    before(C c) : execution(public void C.m()) && affected(c) {
        // advice functionality
    }
}

Figure 8. Instance-local aspects in AspectJ

every matching join point. Moreover, aspect code is executed regardless of whether
the advice will be actually invoked or not.

4.2. AspectWerkz

An AspectWerkz aspect is a collection of definitions of pointcuts and advice. The
latter are methods of arbitrary classes that have to match a given signature. Aspect-
Werkz supports singleton, per-class and per-instance aspects. Weaving takes place
at load-time. A JoinPointManager member is added to each affected class, and
affected methods are replaced by stubs that, at run-time, call the AOP infrastructure
through the manager object. The manager stores unique JoinPoint instances reifying
“activated” join points (i.e., join points that have advice attached to them). For each
join point, there exists exactly one JoinPoint instance that is created lazily when the
join point is first executed. JoinPoint instances contain so-called advice executors
that are responsible for associating a join point with corresponding advice. Advice
executors have an aspect container invoke the advice methods on the appropriate ad-
vice instances. Aspect containers are stored in a global aspect manager, exist once
per aspect and store the aspect’s advice methods.

Advice instances are stored in the aspect container. For singleton aspects, they
are simply held as a member. For per-instance aspects, the aspect container holds
a hash map mapping objects to advice instances. Whenever a per-instance advice
method is to be invoked, the current application object is used as the key to look up
the corresponding advice instance in the hash map. After that, the advice method is
invoked on that instance.

Instance-local decoration with aspects is not directly supported by AspectWerkz,
so it was emulated using an approach similar to the one that was described above for
AspectJ. In the advice code, a static hash set is queried for the respective instance, and if the instance is affected by the aspect, advice execution continues.

4.3. JBoss AOP

In JBoss AOP, aspect functionality is implemented by means of so-called interceptors that decorate join points. An interceptor is a class implementing the Interceptor interface. Instances of these interceptor classes are JBoss AOP’s advice instances.

JBoss AOP modifies classes as they are loaded into the JVM, adding calls to wrappers at possible join point shadows. Advice instances are stored in Advisor objects that are themselves added to the affected classes during the weaving process. For singleton aspects, the Advisors are static, for per-instance aspects they are ordinary class members, just like in AspectJ. Inside an Advisor, interceptors are stored in an array. When a join point shadow is reached, the corresponding Advisor structures’ advice instance array is iterated over, and all appropriate interceptors are invoked.

As JBoss AOP also does not have support for instance-local aspects, we have emulated it using the same approach as for AspectWerkz and AspectJ.

4.4. PROSE

In PROSE, aspects are containers defining a set of crosscuts. Each crosscut is a conglomerate of a pointcut and advice functionality attached to it. Each aspect carries its crosscut instances and advice instances with it. Upon aspect deployment (which process is called insertion in PROSE), the crosscut instances are directly associated with the corresponding join point shadows through registering a debugger breakpoint.

PROSE aspects can be singletons. The standard case is, however, to associate each newly created aspect instance with an object which can either be passed to the aspect constructor or is created transparently. These association objects serve as mere IDs for the aspect instances and are solely used for identity checks and related things. They are – especially if created transparently – in no way related to particular application objects in whose context join points may occur, so that PROSE’s association mechanism does not constitute a functionality like that of AspectJ’s perthis. PROSE does not support per-instance aspect instantiation, this feature would have to be emulated.

As the only one of the systems discussed in this section, PROSE offers explicit API support for instance-local aspects: it is possible to attach filters to pointcuts that look up “target” or “this” instances in a standard Java Collection passed to the filter. Internally, PROSE performs a containment check on this collection for every instance in whose scope a join point occurs. So, PROSE’s approach is merely an API-integrated variant of the emulation approach we have introduced for the other systems.
4.5. Reflex

Reflex modifies classes at load-time by means of a modified class loader that knows, from a configuration with which it was initialised, at which points in the application code links to the meta-level have to be established. At such hooks, Reflex inserts so-called “hooks” that invoke additional logic responsible for meta-object creation.

The objects that we refer to as advice instances are called meta-objects in Reflex. Their creation is controlled by the “link” to the meta-level that is inserted at a particular hook. The granularity of meta-object creation is controllable to a high degree: they can be assigned to a single hook, a set of hooks, a single instance or class, or they can be created per thread or globally. For example, assigning global scope to a link leads to there being only one meta-object being created for all occurrences of that link (i.e., for all hooks that the link connects to the meta-level). This is analogous to the singleton aspect instance in AspectJ. AspectJ’s per this semantics can be implemented in Reflex by assigning instance scope to the link.

Instance-local aspect decoration can be implemented by assigning an activation condition to the link. The link is only established – i.e., the meta-object is only created and control branched to it – if the condition evaluates to true. Basically, the logic inside the instance-local activation condition looks like the AspectJ solution presented in section 4.1.

4.6. Discussion

Most of the aforementioned approaches bind advice instances to affected classes or objects by modifying their structure through adding fields that hold references to advice instances during the weaving process. Advice instance lookup is then a simple field access. We refer to this approach as one that is using language-level data structures, as the entire task of advice instance management is actually done at bytecode level, using constructs that have exact counterparts in Java source code. Such code is therefore subject to normal execution by the VM, which cannot distinguish between original application code and aspect management code. All operations, such as array bounds checking, have to be applied to aspect management code, even if they might be unnecessary due to the used AOP system’s capabilities of ensuring consistent state.

Among the presented systems, PROSE is an exception. It does not at all modify the object structure of affected classes at language level. Instead, it binds advice instance containers to join points (reified through debugger breakpoints) and thus provides a clean separation of application and aspect management level. Still, in spite of the clarity of this approach, it suffers from the use of the debugger infrastructure (section 5). However, such an approach allows for more easily telling apart code constituting base application, advice, and aspect or advice instance management.
None of the systems discussed above supports truly instance-local aspects. It is however, in all cases, possible to restrict the scopes of class-wide aspects to single objects or groups thereof through the use of dedicated data structures. Still, in all of the approaches, aspects affect entire classes, and to restrict their scopes to single instances, affected instances have to be explicitly looked up in language-level data structures from the standard Java API.

5. Performance Measurements

In this section, we will first outline the impact of using AITs on Steamloom’s structure as well as its weaving and JIT compilation performance. Next, we will discuss the actual performance of method invocations in the context of class-wide and instance-locally deployed aspects. All results were obtained by running the respective applications on a Dual Xeon (3 GHz each) with 2 GB of memory under Linux 2.4.23. Where we did not run Steamloom, we used Sun’s HotSpot VM (version 1.4.2_01) in client mode as run-time environment. The performance of an optimised build of the Jikes RVM (and, therefore, the performance of Steamloom) is roughly the same as that of the HotSpot VM in server mode for the SPECjvm98 benchmarks (SPE, n.d.).

5.1. Structural, Weaving and Compilation Impact

The performance impact of running dynamic AOP applications using AITs is considered in depth in section 5.2. In this section, we will briefly discuss the impact of AITs on the size of the VM boot image (Alpern et al., 2000), and on weaving and JIT compilation.

For boot image size comparison, we look at a production build of the Jikes RVM 2.2.1. For an unmodified build, the image is 37 MB large. Building Steamloom without AITs results in a size of 54 MB. With AITs, any TIB is lengthened by one additional word possibly holding an AIT reference. In addition to that, the compilers had to be extended and some additional AIT management logic was needed. The impact is small: the boot image of Steamloom with AITs is only about 66 kB larger than that of Steamloom without AITs.

Weaving logic is responsible for storing and looking up advice instance indices and for passing them along to the JIT compiler in the form of bytecode parameters. Thus, we have measured the total time spent for dynamic (un)weaving and JIT compilation. The application for which we have measured these properties is a small program computing Fibonacci numbers recursively that was decorated with an aspect counting the number of invocations of one of the application’s methods.

Steamloom without AITs spent 35 ms on (un)weaving and 21 ms on JIT compilation, whereas the AIT-based Steamloom needed 31 ms and 18 ms, respectively. Both weaving and compilation benefit from the increased simplicity and decreased size of the woven bytecode that is generated when AITs are used.
5.2. Measurement Results

To obtain detailed performance data, we have run a measurement application on various platforms, namely AspectJ 1.2, AspectWerkz 1.0, JBoss AOP 1.0, PROSE 1.2.1, Reflex 3.0 alpha 11, and Steamloom without and with AITs. All systems but Steamloom were run on the HotSpot VM, version 1.4.2_05.

The measurement application is part of a suite of micro-measurements for dynamic AOP (Haupt et al., 2004). The micro-measurements count the number of given operations, e.g., virtual method invocations, per second. The suite is implemented using the JavaGrande framework (Jav, n.d.).

Micro-measurements for virtual method invocations work in the following way. The VM executes as many method invocations as possible, until a certain threshold is reached, or until a designated time is spent. Then, the average number of method calls per second is computed as the benchmark result. To hinder the VM from optimising away empty method calls, the measured method contains a simple counter increment operation.

In the aspect-oriented context, the measurement application attaches advice to the measured method invocations. The advice methods also contain a simple counter increment. For further details on the measurement suite, we refer to (Haupt et al., 2004).

For the results presented here, we have measured virtual method invocations. Measurements were made for plain method invocations (i.e., with no advice attached to the method, but with the respective AOP environment activated), for method invocations in the context of a class-wide aspect, and for method invocations on unaffected and affected instances in the presence of an instance-locally deployed aspect. Only PROSE and Steamloom directly support this feature, so it had to be emulated in the other systems. For AspectJ, the approach displayed in Figure 8 (section 4.1) was used. It was taken analogously for AspectWerkz and JBoss AOP. In Reflex, we have used an activation condition to implement instance-local decoration. The activation condition however followed the same approach as the AspectJ solution.

The measurement results are displayed in Figure 9. Please note that the $y$ axis is logarithmic.

For plain method invocations, all systems perform almost equally well, executing over $10^8$ method calls per second. As soon as advice invocations are involved, all systems exhibit a certain performance loss. Part of this loss is of course due to the cost of invoking the advice themselves, but nevertheless, the performances of the various approaches differ due to their respective implementation techniques. In case of a class-wide aspect, AspectJ clearly benefits from its static weaving approach and the fact that the singleton aspect instance is returned by a static method, which is a good opportunity for the JIT to inline such calls. AspectWerkz, JBoss AOP perform poorer since wrapper executions inflict a certain overhead. Reflex also exhibits some reduction due to its strategy of creating meta-objects during execution. PROSE suffers
Figure 9. Results of performance measurements

from expensive context switches associated with debugger breakpoint interception. The high cost of the debugger approach eliminates the efficient direct association of advice instances to join points described in section 4.4. Steamloom without AITs performs better than the four aforementioned systems but suffers from the synchronised access to the advice instance data structure. Steamloom with AIT support performs extremely well, albeit slower than AspectJ, because AIT lookups are slightly more expensive than AspectJ’s lookup strategy (section 3.2).

The most interesting cases are those in which instance-local aspects are deployed. Let us first compare the performance of AspectJ, AspectWerkz, JBoss AOP, Reflex and PROSE. Of these systems, only AspectJ exhibits a significant performance loss as compared to the class-wide aspect. This is because the comparatively high cost of looking up the affected instances in a hash set is more significant in an approach that is otherwise extremely efficient. The generally lower performance of AspectWerkz, JBoss AOP and PROSE is not significantly reduced by the additional checking operations. Reflex benefits from its use of activation conditions: in the case of sending a method call to an unaffected instance, the condition evaluates to false and the expensive meta-object creation is avoided.

Steamloom without AITs again exhibits a significant performance loss for calls to affected objects as compared to plain method invocations, which is again due to the synchronised lookup. However, method calls to unaffected instances are not slowed down since the way instance-local aspect deployment is managed avoids expensive checking operations. The performance of Steamloom with AITs shows that such VM-
level structures have clear benefits: calls to the affected instance are still expensive, but the overall performance is even better than that of AspectJ.

In the affected instance case, Steamloom with or without AITs performs slightly less well than in the class-wide case. This may be irritating at first, since there should not be an additional overhead for dispatching a virtual method call to an affected instance. The observed small performance gap is however to be expected as inlining is forbidden for methods affected by an instance-locally deployed aspect (section 2.2).

6. Conclusion

As we have shown in section 4, existing AOP implementations mostly address the binding of advice instances to affected classes and objects by modifying the latters’ structure at language level: weaving logic adds fields to the classes that hold either references to advice instances, or to controller objects serving as entry points to the AOP infrastructure. Directly storing advice instances in affected classes (at language level, i.e., like ordinary Java class members) harms the classes’ structural integrity. Moreover, all of the code responsible for aspect management in general and advice instance management in particular looks, to the VM, like ordinary application code and is therefore executed in no different way. This makes the resulting (interwoven) code hard to debug. PROSE follows a minimalistic approach by using debugger breakpoints to reify join points and leaving all classes of the application intact. Still, PROSE suffers from the employed technology (the JVM debugger infrastructure), and its infrastructural code is still subject to normal execution by the VM.

We have presented the Steamloom VM, an implementation of a virtual machine with native support for aspect-oriented programming. Steamloom clearly separates aspect management code from application code by tightly integrating all aspect and advice instance management tasks with the VM itself. As an enhancement of the Steamloom VM, we have introduced the novel concept of advice instance tables (AITs). Advice instance tables represent a means to access and manage advice instances at very high speed and low memory overhead, as no language-level data structures or operations are employed. Accessing language-level data structures comprises all kinds of checking operations and plausibility checks, such as array bounds and cast validity checking. Such operations are completely avoided when AITs are used as the Steamloom environment guarantees the integrity of the tables.

Our claim that such integration would bring performance improvements was underpinned by results from performance measurements which we have obtained by running a micro-measurement suite on various AOP platforms. First of all, for class-wide aspects, Steamloom generally exhibits the best performance of all dynamic AOP systems in scope. Having been enhanced with AITs, Steamloom even gets very close to the performance of AspectJ’s statically woven code. In the special case of instance-local aspects, Steamloom’s performance is, while being good without AITs, superior when fast advice instance lookup through AITs is employed. These results clearly
show that performance benefits from VM-level support for dynamic join points. In a nutshell, Steamloom provides – at roughly the same speed as a static weaving approach yields – support for both class-wide and instance-locally deployed aspects.

Dynamic AOP support as presented in this paper is not suited for direct use on the Sun standard VM or any other unmodified virtual machine. Advice instance tables cannot straightforwardly be integrated with other dynamic AOP implementations as, e.g., JBoss AOP or even PROSE. This is to be expected since concepts like AITs are explicitly meant to be supported by the virtual machine itself, not by some application- or language-level infrastructure. On the one hand, this may be considered a drawback since the use of a modified VM is not always acceptable. On the other hand, none of the elements found in Jikes that were exploited to build Steamloom are missing in other virtual machine implementations; with some effort, it is possible to integrate Steamloom-like support for dynamic aspects with any virtual machine.

Our ongoing work focuses on providing a sound platform for AOP languages targetable from arbitrary compilers. The main design goal is to provide efficient solutions that support AOP implicitly and avoid language-level constructs as much as possible.

Acknowledgements

We are grateful for Christoph Bockisch’s invaluable work on the implementation of Steamloom, and for his contributions to discussions affecting its development. The same holds for Tom Dinkelaker, who has worked on the implementation of AITs. We also thank Anis Charfi for preparing the French version of this paper’s abstract.

7. References


