Proceedings of the First workshop on Virtual Machines and Intermediate Languages for emerging modularization mechanisms (VMIL 2007), held at the Sixth International Conference on Aspect-Oriented Software Development, March 12-16, Vancouver, British Columbia, Canada

VMIL '07

Workshop Chairs:
Hridesh Rajan and Mira Mezini

ACM International Conference Proceedings Series
ACM Press

The Association for Computing Machinery
1515 Broadway
New York New York 10036

ACM ISBN: 978-1-59593-661-5

Workshop Organization

Program Committee:

- Samik Basu (Iowa State University, USA)
- Lodewijk Bergmans (University of Twente, The Netherlands)
- Christoph Bockisch (Darmstadt University of Technology, Germany)
- Bruce Childers (University of Pittsburgh, USA)
- Yvonne Coady (University of Victoria, Canada)
- Michael Haupt (Hasso Plattner Institute, University of Potsdam, Germany)
- Hidehiko Masuhara (University of Tokyo, Japan)
- Mira Mezini (Darmstadt University of Technology, Germany) [Co-Chair]
- Oege de Moor (University of Oxford, UK)
- Klaus Ostermann (Darmstadt University of Technology, Germany)
- Hridesh Rajan (Iowa State University, USA) [Co-Chair]
- Therapon Skotiniotis (Northeastern University, USA)
- Eric Tanter (University of Chile, Chile)
- Eric Van Wyk (University of Minnesota, USA)
- Eric Wohlstadter (University of British Columbia, Canada)

Organization Committee:

- Hridesh Rajan, Iowa State University
- Mira Mezini, Darmstadt University of Technology
- Christoph Bockisch, Darmstadt University of Technology

Table of Contents

A Flexible Architecture for Pointcut-Advice Language Implementations

Christoph Bockisch Mira Mezini

Darmstadt University of Technology
Hochschulstr. 10, 64289 Darmstadt, Germany
{bockisch, mezini}@informatik.tu-darmstadt.de

Abstract

Current implementations for aspect-oriented programming languages map the aspect-oriented concepts of source programs to object-oriented bytecode. This hinders execution environments with dedicated support for such concepts in applying their optimizations, as they have to recover the original aspect definition from bytecode. To address this representational gap we propose an architecture for implementations of pointcut-advice languages where aspect-oriented concepts are preserved as first-class entities. In this architecture, compilers generate a model of the crosscutting which is executed by virtual machines.

In this paper we discuss a meta-model for aspect-oriented concepts and a virtual machine-integrated weaver for the meta-model. As a proof of concept, we also provide an instantiation of the meta-model with the AspectJ language and an AspectJ compiler complying with the proposed architecture. We also discuss how preexisting high-performance optimizations of aspect-oriented concepts benefit from the proposed architecture.

1. Introduction

This paper is concerned with the implementation of the pointcut-advice sub-family of aspect-oriented languages, PA languages [1] for short. Most aspect-oriented languages belong to this family as they provide the pointcut and advice constructs to support the modularization of behavioral crosscutting. For simplicity, we will use the term "aspect-oriented languages" to refer to the pointcut-advice sub-family where no distinction is necessary.

The most prevalent implementation strategies of PA languages share the following conceptual workflow [2, 3]. First, the aspect-oriented program is parsed to retrieve pointcuts and advice from aspect modules. Next, join point shadows [2, 4] are searched for. Finally, bytecode instructions for dispatching advice functionality are generated and inserted at these shadows. The latter two steps are generally referred to as the weaving phase.

During this phase aspect-oriented (AO) concepts, which are expressed by special language constructs in the source code, drive code generation and transformations in the compiler. One implication of this approach is that code originally modularized in aspects is merged with code of the base language's modules. An obvious problem with this is the weakening of the continuity property of incremental compilation: Modifying an aspect potentially requires re-weaving in multiple other modules [5]. Moreover, the aspect-oriented concepts become implicit instead of staying first-class in the generated bytecode.

This is different from object-oriented programming languages, e.g., Java, where concepts like classes, methods, fields and even polymorphism are also reflected in the bytecode. In Java bytecode, polymorphic method calls can be immediately identified because they are represented by a special instruction rather than general-purpose instructions encoding the resolution logic. In contrast, current implementations of aspect-oriented languages generate sequences of general-purpose instructions to realize concepts like the cflow pointcut designator [6].

This representational gap hinders a range of optimizations by the execution environment [7, 8]. It is very difficult task for the just-in-time compiler of a virtual machine to recognize the intention of a sequence of instructions generated by the weaver, e.g., for checking whether a control flow is active. On the contrary, a first-class representation of quantifications over the control-flow, can be exploited by the virtual machine's just-in-time compiler to perform optimizations during the native code generation [9]. Other optimizations of this kind are also outlined in [9]. Furthermore, with a first-class representation of aspect-oriented concepts, object layouts that better suit the needs of aspects can be designed in the virtual machine [10].

Approaches like the AspectBench Compiler framework (`abc`) [11, 12] apply static analyzes based on a first-class model of the aspects to achieve some performance optimizations. Yet, this first-class model also only exists during compile- and weaving-time and is abandoned afterwards. Optimized implementations that can rely on virtual machine features to achieve efficient runtime execution are out of reach.

Besides facilitating optimizations in the execution environment, a first-class representation of AO concepts also supports development of new language extensions. The `abc` framework already supports re-using and extending the implementation of some core concepts in static compilers. It exposes interfaces for join point shadow search and weaving, which can be used to realize new kinds of pointcut designators [11, 12]. However, other concepts, e.g., concerning the aspect instantiation strategy or advice precedence, are not first-class. Hence, language extensions that target these parts of an aspect-oriented language can not re-use implementation efforts provided by the `abc` framework. Aspect-oriented languages that, e.g., support runtime deployment of aspects, like JAsCo [13, 14] or AspectWerkz [15, 16], can not benefit from `abc` at all, because the latter lacks an abstraction over the aspect deployment strategy.

The work presented in this paper targets the problems outlined so far. We propose an architecture for aspect-oriented language

Workshop VMIL '07 March 12-13, 2007, Vancouver, British Columbia, Canada.
Copyright © 2007 ACM 1-59593-661-5/07/03...$5.00

implementations where the aspect-oriented concepts stay first-class until execution. Centric to this architecture is a meta-model of core aspect-oriented concepts which acts as interface between compilers and runtime environments[1]. Compilers *instantiate* the meta-model with language constructs while runtime environments *implement* the execution of the meta-model. The core concepts made explicit in the current meta-model are: advice, join point shadows, dynamic properties of matching join points, context reification at join points, strategies for implicitly instantiating aspects, advice ordering at shared join points and aspect deployment.

A concrete instantiation of the architecture is also presented, consisting of two parts. First, to show that the meta-model is appropriate to express state-of-the-art aspect-oriented languages, we provide a compiler that expresses the AspectJ language features as instantiations of the meta-model. Second, we provide an implementation of the meta-model by means of a Java 5 agent and Class Redefinition [17]. This implementation serves as an unoptimized default solution to execute the meta-model on any standard Java 5 virtual machine.

The default implementation targets only one of the problems identified with current implementations: It better supports the continuity of incremental compilation. To show how the proposal supports sophisticated optimizations of the AO concepts, we discuss a second optimized implementation of the model by a dedicated virtual machine. Thereby, we show how the first-class AO concepts can drive advanced optimizations by the just-in-time compilers of virtual machines. Finally, to show that the architecture and its meta-model supports language extensions, a mapping for some advanced AO concepts and optimized implementations thereof is outlined.

The remainder of this paper is organized as follows. In section 2, the proposed architecture of PA language implementations and its meta-model is presented. Section 3 discusses a concrete instantiation and the default implementation of the proposed meta-model. We discuss alternative instantiations of the architecture in section 4 and present related work in section 5. Finally, section 6 concludes the paper and presents areas of future work.

2. Architecture and Meta-Model

In this section, an overview of the proposed architecture is given and the underlying meta-model of pointcut-advice languages is presented.

2.1 Overview of the Architecture

The proposed architecture is schematically shown in figure 1. The central block of the architecture is the *meta-model of pointcut-advice languages* which has been designed to be generic enough to accommodate the concepts of most current PA languages. It is defined as a set of interfaces and classes in the Java language. A concrete language implementation has different connections to the meta-model.

Concrete AO language features are mapped to the meta-model by implementing the interfaces according to the feature's semantics, we speak of *instantiating the meta-model*. Mapping a concrete language to the meta-model results in the model for the respective language.

Compilers adhering to the architecture (in the front-end block of the figure) generate bytecode as usual for the non-aspect parts of the program, i.e., object-oriented modules and object-oriented parts of aspect-oriented modules. Additionally, they create bytecode, called the *preamble*, that instantiates and configures model entities according to the behavioral crosscutting definitions in the source code under compilation. The preamble is executed before running

[1] One could also call the meta-model an intermediate representation for aspect-oriented programming languages.

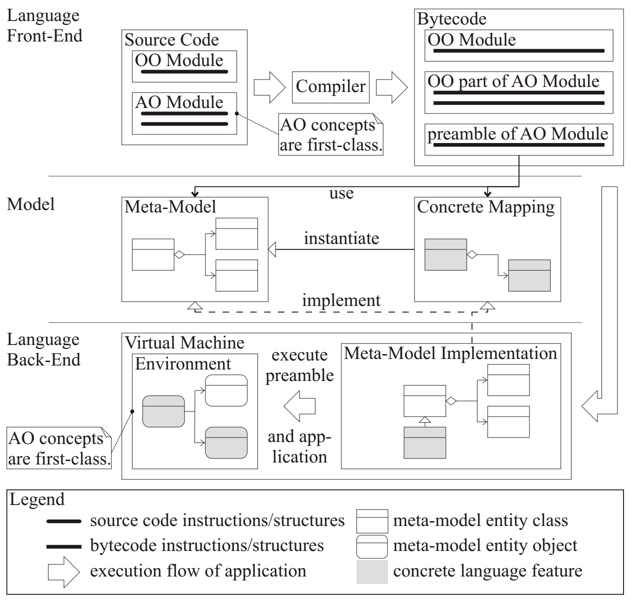


Figure 1. Schema of the proposed architecture for pointcut-advice language implementations.

the actual application. Its execution creates and possibly deploys the first-class representation of the crosscutting definitions.

Execution environments that adhere to the architecture (in the back-end block of the figure) *implement the meta-model*. Since all elements of the aspect-oriented programs are represented as objects at runtime, the execution environment has full access to a high-level description of the behavioral crosscutting defined by the AO program. This enables it to employ sophisticated optimizations. But also simple bytecode weaving as in existing implementations is possible. Beyond advanced optimizations, different implementations of the deployment interface can also provide advanced features such as synchronized or transactional aspect deployment.

In other words, the meta-model serves as the interface between AO compilers and execution environments, hence, decoupling the front-end from the back-end of an AO language. Besides providing the execution environment with first-class representations of AO concepts, this decoupling makes compilers and execution environments independently interchangeable: New implementations of the meta-model by new virtual machines can be used as the back-end for existing compilers, and the code produced by new compilers can execute on any execution environment that implements the meta-model.

2.2 A Meta-Model for Pointcut-Advice

The meta-model breaks down an aspect into small, differentiated units to improve re-usability and to avoid ambiguities. Figure 2 shows the elements of the model and their connections by means of a class diagram. The `Aspect` module provides a logical grouping of `AdviceUnits` – representing pointcut-advice pairs – to be deployed together. The other modules are concerned with expressing either pointcuts or advice functionality.

2.2.1 Expressing Pointcuts.

Pointcuts are represented as `JoinPointSet` data structures – the participating classes are marked by the box labeled *pointcut* in figure 2. A `JoinPointSet` corresponds to a simple clause in a pointcut expression of an aspect. A simple clause is one that consists of a single static pointcut designator, e.g., a call or a field access desig-

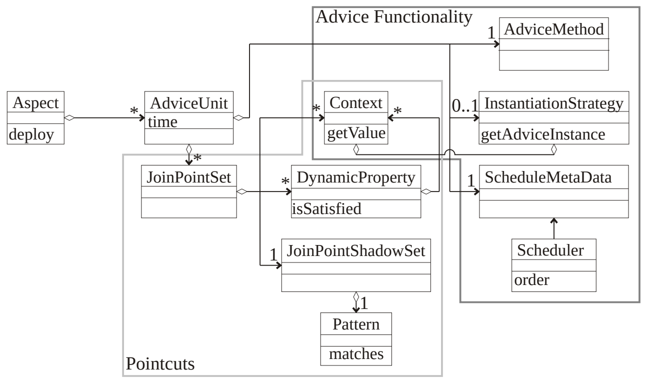


Figure 2. A meta-model for pointcut-advice languages.

nator, eventually combined with dynamic properties by an *and* (`&&`) operation. For instance, `call(pattern)&& dyn` is a simple pointcut clause where `call(pattern)` is a static pointcut designator selecting all calls to methods that match `pattern`, and `dyn` is some dynamic property that the method calls selected by the static designator must also fulfill.

Accordingly, each `JoinPointSet`, jps, has a static component, $jpss$ of type `JoinPointShadowSet`, representing its shadows [2], and a dynamic component, an set of `DynamicProperty` objects. A join point in jps occurs at runtime when an instruction in $jpss$ is executed and all dynamic properties are satisfied. In addition, a `JoinPointSet` may also refer to `Context` objects, representing values that are exposed to the advice at member join points.

`Context` objects represent some part of a join point's context – e.g., the **`this`** object or the source code location of a join point's shadow – and are used by several parts of the meta-model. Contexts can also be composed of other contexts, e.g., the **`thisJoinPoint`** value in AspectJ can be realized as a composed context.

A compound pointcut expression can be expressed in the meta-model by simple clauses combined with *or* (`||`) operators. An example of a pointcut in the *ored* form is `(call(pattern1)&& dyn1)|| (set(pattern2)&& dyn2)`. However, not all pointcuts in AspectJ-like pointcut languages are already in this form. For instance, the pointcut expression `call(pattern1)&& dyn && call( pattern2)` is not in the *ored* form. Yet such pointcut expressions can be transformed into the *ored* form. That is, the *ored* form does not constrain the valid pointcut expressions that can be defined.

The idea is that any pointcut expression that produces a non-empty join point set can be brought into an *ored* form. Given two pointcut designators, p_1 and p_2 with patterns pt_1 respectively pt_2, the pointcut p_1 && p_2 is equivalent to the pointcut p in *ored* form whose pattern is $pt_1 \wedge pt_2$.

For illustration, consider the following listing. It shows several pointcut definitions in a pseudo language, followed by an equivalent pointcut expression in *ored* form. The third example cannot be expressed in an *ored* form. However, this pointcut represents a family of pointcuts with empty join point sets. There can never be a join point which is a method call and a field set at the same time.

```
// 1.
// original pointcut definition :
call(pattern1) && dyn && call(pattern2)
// equivalent pointcut definition in ored form:
call(pattern1 && pattern2) && dyn

// 2.
// original pointcut definition :
(call(pattern1) || set(pattern2)) && dyn
// equivalent pointcut definition in ored form:
(call(pattern1) && dyn) || (set(pattern2) && dyn)

// 3.
// original pointcut definition :
(call(pattern1)) && (set(pattern2))
// this is an illegal pointcut
```

A pointcut in our meta-model is specified as an array of `JoinPointSets` which corresponds to a list of simple pointcut clauses combined with an *or* (`||`) operation.

An instance of `JoinPointShadowSet` refers to a `Pattern` object, which describes lexical properties of join point actions, e.g., the name or signature of the called method. A join point shadow set is, conceptually, evaluated by matching the lexical context of all join point shadows in the application against its pattern object.

Dynamic properties model the program's state at the time a join point is executed, e.g., the active control flow or the receiver object. A `DynamicProperty` can specify which values from the context of a join point are required for its evaluation by referring to respective `Context` objects.

2.2.2 Expressing Advice Functionality.

An `AdviceUnit` defines crosscutting functionality by specifying its *what*, *where*, and *when*. The *what* is defined in an `AdviceMethod` [2] and the *where* by a set of `JoinPointSets`. The `time` property of `AdviceUnit` (representing the *when*) determines whether the advice method is executed *before*, *after*[3] or *around* a join point. In the following, we will discuss the meta-model entities concerned with defining the *what*, marked in figure 2 by a box labled *advice functionality*.

Advice methods may need an object on which to execute – an *advice instance*. Generally, an advice may execute on different advice instances at different join points matched by its pointcut. For illustration, consider an AspectJ aspect declared with a **`pertarget`** clause and consisting of a call pointcut and an advice. The **`pertarget`** clause states that the advice of this aspect executes on different advice instances for different target objects at call join points matched by the pointcut.

The strategy for retrieving an advice instance is captured by `InstantiationStrategy`. To specify the values that it may need from a join point's context to retrieve an appropriate advice instance an instantiation strategy may refer to an arbitrary number of `Context` objects. When an advice method is executed at some join point, the required context values are determined and passed to the `InstantiationStrategy` which returns the advice instance on which to execute the advice method.

An advice unit also has `ScheduleMetaData` associated with it. `ScheduleMetaData` is used to determine the order of execution when multiple advice apply at the same join point. A simple form of schedule meta-data is a priority level associated with each advice unit. When two or more advice units are executed at the same join point, the one with the highest priority gets executed first. The `Scheduler` interface – which is responsible for generating a concrete order given some schedule meta-data – must be co-implemented with the schedule meta-data.

When multiple advice units must be executed at a join point shadow, the meta-model implementation will pass their schedule meta-data to the scheduler, which determines an ordering of the advice. The ordering is encoded as a chain of `AdviceOrderElement` objects. The structure of such a chain is defined by the class diagram in figure 3, whereby *action* is a reference to either an advice unit or the actual join point action.

[2] In the default implementation normal methods are used as advice methods.

[3] It is possible to specify that the advice executes after normal or exceptional execution of the join point.

Each `AdviceOrderElement` object stores a (possibly empty) list of *before* advice in the order of their execution, followed by an *around* advice, which, in turn, is followed by a (possibly empty) list of *after* advice to be executed after the *around* advice has finished. Any `AdviceOrderElement`, aoe_i, may be linked to a following element, aoe_{i+1}, thus, specifying the advice to execute, when the *around* advice of aoe_i **`proceed`**s. Once the execution of the last *after* advice of aoe_{i+1} is over, the *around* advice of aoe_i continues after its **`proceed`**, followed by the *after* advice of aoe_i, if any. The *around* of the last element of an `AdviceOrderElement` chain refers to the join point action rather than to an advice.

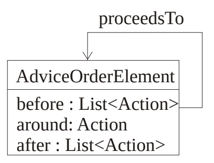


Figure 3. Data structure for representing the order of advice.

For illustration, consider a priority-based scheduler and three advice units - *A*, *B*, *C* - to be executed at the same join point shadow. *A* and *B* are *before* advice units with priorities 100, respectively 80; *C* is an *around* advice with priority 90. Further, assume that the scheduling strategy is such that a *before* advice with a lower priority than an *around* advice is executed only when the latter **`proceed`**s. Figure 4 shows the order structure returned by the scheduler for this example.

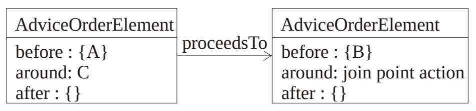


Figure 4. Example `AdviceOrderElement` data structure.

3. Default Instantiation of the Architecture

To show that the meta-model is appropriate for expressing aspect-oriented features of current languages and to illustrate the proposed architecture we will discuss an instantiation of the architecture, here. We have instantiated the meta-model with the AspectJ language features and implemented both, a compiler and an execution environment for the meta-model.

In the following subsection, we discuss how the AspectJ features are expressed as instantiations of the meta-model as well as the compiler for the AspectJ language that obeys the proposed architecture. In subsection 3.2 we present an implementation of an execution environment to execute the meta-model based on byte-code weaving. Dynamic aspect deployment is facilitated by means of a Java 5 agent and Class Redefinition [17].

3.1 Mapping AspectJ

In the following, we discuss how AspectJ features described in appendix B *Language Semantics* of the AspectJ Programming Guide [6] can be mapped to our meta-model. We also discuss the code generated by a compiler for building model entities that represent the crosscutting in the source code. The AspectJ compiler conforming to our architecture is built using the JastAdd compiler framework [18] which allows for flexibly extending the processed language features.

The AspectJ aspect in listing 1 will be used to illustrate the discussion in this section. Listing 2 shows the preamble code that expresses the same aspect in terms of our meta-model, using implementations of meta-model interfaces discussed in this subsection. Figure 5 shows an object diagram created by the preamble.

```
1  aspect BoundPoint issingleton() {
2
3    before(Point p):
4        call(void Point.set*(*)) &&
5        target(p)
6    {
7      // ...
8    }
9
10 }
```

Listing 1. Aspect definition in the AspectJ language.

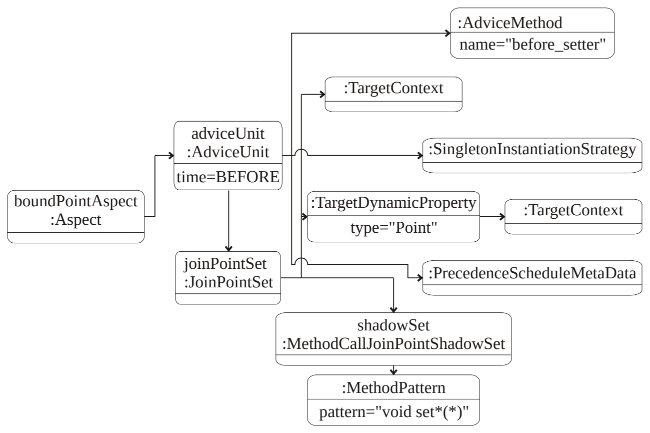


Figure 5. Meta-object structure modeling the AspectJ aspect.

To start with, the compiler creates a class for each aspect (lines 1 to 5 in listing 2) that contains a method (lines 2 to 4, listing 2) for each advice (lines 6 to 8, listing 1).

Next, the compiler generates the preamble code, shown in lines 8 to 30 in listing 2. Our AspectJ compiler generates this code into the static initializer of the main class. The preamble contains code for setting up the pointcut (lines 8 to 17) and the advice functionality (lines 19 to 26), finally, the preamble also contains the deployment of the defined aspects (lines 28 to 30).

3.1.1 Pointcut.

Join points described in [6], *method call*, *field set*, etc., are mapped to implementations of the classes `JoinPointShadowSet` and `Pattern` [4]. For illustration, consider the lines 8 to 17 in listing 2, where a `MethodPattern` and a `MethodCallJoinPointShadowSet` object are created to express the **`call`** pointcut from line 4 in listing 1.

`Pattern` classes are provided for matching the fully qualified signature of a method, field, and type against a pattern specified in the AspectJ wildcard notation. To evaluate type patterns containing the "+" operator, the `Pattern` implementation needs information about the type hierarchy. The required information can be gathered by intercepting class loading.

AspectJ also defines pointcut designators (PCD) that select join points based on dynamic properties. The pointcut designators **`target`**, **`this`**, and **`args`**, specify the dynamic type that the receiver object, the executing object, respectively the argument objects, must have in order for an execution point to classify as a join point.

[4] The current implementation does not support *Handler execution*, *Advice execution*, *Constructor execution*, *Object initialization* and *Object pre-initialization*; implementations of these constructs are subject to future work. As for object creation join points, we currently only provide an equivalent to *Constructor call*.

```
 1 class BoundPoint {
 2   before_setter(Point p) {
 3     // ...
 4   }
 5 }
 6
 7 ...
 8 MethodCallPattern setterPattern = Patterns.ajMethodPattern("call(void Point.set*(*))");
 9
10 JoinPointShadowSet shadowSet = AspectModelFactory.createMethodCallJoinPointShadowSet(setterPattern);
11
12 Set<DynamicProperty> dynamicProperties =
13   Collections.singleton(AspectModelFactory.createTargetDynamicProperty(Point.class));
14
15 List<Context> contexts = Collections.singleton(AspectModelFactory.createTargetContext());
16
17 JoinPointSet joinPointSet = AspectModelFactory.createJoinPointSet(shadowSet, dynamicProperties, contexts);
18
19 Class boundPointClass = BoundPoint.class;
20 Method adviceMethod = boundPointClass.getDeclaredMethod("before_setter", new Class[]{Point.class});
21 AdviceUnit adviceUnit = new AdviceUnit(
22   BEFORE,
23   Collections.singleton(joinPointSet),
24   AspectModelFactory.createSingletonInstantiationStrategy(boundPointClass),
25   adviceMethod,
26   new PrecedenceScheduleMetaData());
27
28 Aspect boundPointAspect = AspectModelFactory.createAspect(Collections.singleton(adviceUnit));
29
30 AspectModelFactory.deploy(boundPointAspect);
```

Listing 2. Aspect from listing 1 in our model.

We provide implementations of the interface `DynamicProperty` for these designators. These implementations are configured with the type to which the respective context value must conform. For illustration, consider the code in line 13 in listing 2, which expresses the **`target`** pointcut designator in line 5 of listing 1.

In our model, the pointcut designators **`within`**, **`withincode`**, **`cflow`** and **`cflowbelow`**, are also mapped to implementations of `DynamicProperty`. In AspectJ, **`within`** and **`withincode`** select join points based on their lexical scope and are statically resolved by the `ajc` and `abc` weaver [2, 19]. At the conceptual level, we consider them more generally as dynamic properties that select join points based on the topmost frame in the call stack. This allows to keep the mapping independent of a certain weaving strategy. For instance, in the efficient weaving technique presented in [20, 21], it is not possible to evaluate **`within`** statically.

When a mapping of concrete language features to the meta-model is provided, the feature must be realized in terms of the interface of the meta-model. This interface is very general – for dynamic properties it is simply the method `isSatisfied` – which allows (a) a uniform treatment by meta-model implementations (e.g., a weaver) and (b) an implementation of the feature using Java's full computational power. A weaver that optimizes certain features will not use this general interface, but instead directly generate code driven by the special knowledge about the feature.

Listing 3 sketches the default implementation of the **`cflow`** pointcut designator of AspectJ, which basically follows the implementation scheme of the AspectJ compilers [19].

The constructor of `CflowDynamicProperty`'s default implementation receives an array of `JoinPointSets`, corresponding to the pointcut of a **`cflow`** in AspectJ. From these join point sets a *before* `AdviceUnit` is created, whose advice method is the `increase` method defined in the class `CflowDynamicProperty`. Similarly, an *after* advice unit is created that decreases the counter. The `ExplicitInstantiationStrategy` used in line 19 always returns the specified object as the advice instance. The `isSatisfied` method of a `CflowDynamicProperty` returns **`true`** if the counter is greater than zero.

Because we use the abstract factory design pattern [22] to create the model entities, it is easily possible to replace their concrete implementations. A virtual machine with dedicated optimizations for cflow, e.g., can overwrite the factory and construct an appropriate object for representing the cflow dynamic property in a way transparent to the user. The factory method receives all information that describes the cflow dynamic property on an abstract level, i.e., a description of the join points constituting the control flow in question. An alternative implementation would not use this description as in the example above, but store it and make it available to the virtual machine. Execution environments that do not overrise the factory will override the default implementation. This topic will be discussed further in section 4.2.

AspectJ defines the pointcut designators **`target`**, **`this`** and **`args`** to bind values from the dynamic context of a join point and to make them accessible to the advice. An example is given in line 5, listing 1. These pointcut designators are modeled as subclasses of `Context`[5]. In the default implementation, the `getValue` method of these `Context` subclasses exploit the Java Virtual Machine Tools Interface (JVMTI) [23] to access local values in the join point's context. An optimized implementation is discussed in section 3.2. Lines 15 to 15 in listing 2 show an example that uses these context implementations.

The special forms **`thisJoinPoint`**, **`thisJoinPointStaticPart`** and **`thisEnclosingJoinPointStaticPart`** available in AspectJ are also mapped to `Context` implementations. These implementa-

[5] It is also possible to bind values at entry points of a control flow in AspectJ. For these designators a default implementation can be provided similar to the cflow dynamic property. As for all parts of the meta-model, an optimized implementation is also possible.

```
public class CflowDynamicProperty
    extends DynamicProperty {

  private int counter;

  public void increase() {
    counter++;
  }

  // ...

  public CflowDynamicProperty
      (Set<JoinPointSet> joinPointSets) {
    AdviceUnit increaseAdviceUnit =
      AspectModelFactory.createAdviceUnit(
        BEFORE,
        joinPointSets,
        AspcetModelFactory.
          createExplicitInstantiationStrategy(this),
        CflowDynamicProperty.class.getDeclaredMethod(
          "increase", new Class[0]),
        new PrecedenceScheduleMetaData());

    // ...
  }

  public boolean isSatisfied(Object[] contextValues)
       {
    return counter > 0;
  }
}
```

Listing 3. Implementation of *cflow* in our model.

tions require context values such as the target object or the signature of the join point, which are used to create an instance of `JoinPoint`.

3.1.2 Advice Functionality.

The *per clause* in AspectJ controlls the retrieval of advice instances. The keyword **issingleton** in listing 1, line 1, specifies that all advice are executed on the same advice instance. Line 24 in listing 2 illustrates the use of the `SingletonInstantiationStrategy`, whose implementation is sketched in listing 4. The first time an advice instance is needed, a new instance of the specified class is created and used for all subsequent advice method executions. Respective `InstantiationStrategy` implementations are also provided for **perthis**, **pertarget**, **percflow** and **percflowbelow**.

```
public class SingletonInstantiationStrategy
extends InstantiationStrategy {

  private Class type;
  private Object singleton;

  public SingletonInstantiationStrategy (Class type) {
    this.type = type;
  }

  public Object getAdviceInstance
      (Object[] contextValues) {
    if(singleton == null) {
      singleton = type.newInstance();
    }
    return singleton;
  }
}
```

Listing 4. Singleton instantiation strategy.

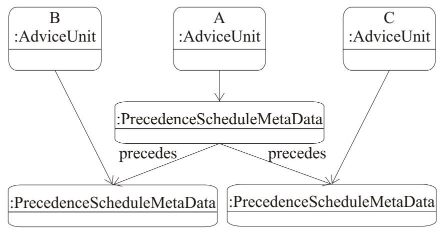

B
:AdviceUnit
A
:AdviceUnit
C
:AdviceUnit
:PrecedenceScheduleMetaData
precedes
precedes
:PrecedenceScheduleMetaData
:PrecedenceScheduleMetaData

Figure 6. An example for the `PrecedenceScheduleMetaData`.

Advice precedence is specified in AspectJ either implicitly by the order in which advice are defined in an aspect, or explicitly by `declare precedence`. To map this feature, we provide the classes `PrecedenceScheduler` and `PrecedenceScheduleMetaData`, which implement the interfaces `Scheduler`, respectively `ScheduleMetaData` . In listing 2, line 26, the advice unit is initialized with an empty `PrecedenceScheduleMetaData`, because it is the only advice unit and does not precede another one.

For a more sophisticated example consider three *before* advice units *A*, *B* and *C*, where *A* precedes both *B* and *C*. The precedence schedule meta-data of *A* stores a reference to the precedence schedule meta-data of *B* and *C* (figure 6 shows a corresponding object diagram). When advice unit *A* and *B* have to be executed at the same join point shadow, the `PrecedenceScheduler` implementation is passed the schedule meta data of *A* and *B*; it returns that *A* must be executed before *B*.

It might seem naive to model aspects at this level of granularity risking poor performance. However, the meta-model was designed to preserve all the concepts that have been present in the source code in a way independent from the weaver implementation. In our architecture, the implementation of the meta-model by components of the virtual machine is responsible for performing optimizations to the model, as discussed in the following subsection and in section 4.

The AspectJ compiler presented here as well as the corresponding execution environment implementation discussed in the following subsection have been integrated in a modified version of the AJDT, called the AJDT-EM (EM stands for Execution Model). Documentation about the AJDT-EM can be found in [24], installation and usage instructions are available at [25].

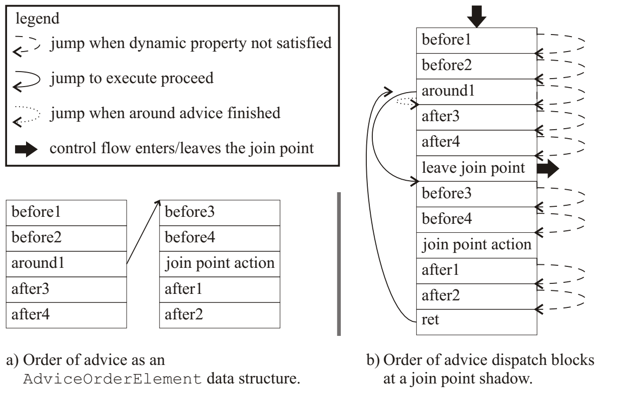
3.2 Default Weaver for the Meta-Model

The factory for our meta-model provides the `deploy` operation for aspects. When an aspect is deployed by invoking this operation, the execution environment must take care that the aspect is active during the subsequent execution. As the default implementation of such an execution environment, we provide a Java agent as an extension to a standard Java 5 virtual machine (JVM). The default weaver implementation is also discussed in [26, 27].

The agent uses the bytecode instrumentation package to intercept class loading: When a class is loaded by the virtual machine, the agent processes the class data and stores meta-information about the class to be used for join point shadow search and weaving. Upon aspect deployment, the agent performs bytecode weaving similar to existing aspect weavers and uses the Class Redefinition facility (also part of the bytecode instrumentation package) of JVMs to replace the old bytecode of classes with the bytecode where the aspect is woven in. When a class is loaded the contained join point shadows are matched against the join point shadow sets of the aspects currently deployed. When a shadow is matched all corrsponding advice units are deployed.

Upon aspect deployment, join point shadow search is performed. The `JoinPointShadowSets` of all `JoinPointSets` associated with the advice units in the aspect are evaluated. The result of the evaluation is a set of join point shadow meta-objects. The latter store information about advice units applying to the corresponding shadow: the `AdviceMethod`, the `ScheduleMetaData` and the `InstantiationStrategy` of the `AdviceUnit` as well as the `Contexts` and the `DynamicPropertys` of the matched `JoinPointSet`. After all advice units are processed in this way, bytecode is generated for all affected join point shadows and the Class Redefinition facility is used to replace the bytecode of affected methods in the virtual machine.

The code that is generated by the weaver for checking the dynamic properties and executing the advice method is called *advice dispatch block*. The execution order of the advice dispatch blocks is determined by the scheduler based on the `ScheduleMetaData` objects of the corresponding advice units. Figure 7 shows (a) an advice order structure and (b) how it is mapped onto a sequence of advice dispatch blocks.



legend
jump when dynamic property not satisfied
jump to execute proceed
jump when around advice finished
control flow enters/leaves the join point
before1
before2
around1
after3
after4
before3
before4
join point action
after1
after2
leave join point
ret
a) Order of advice as an AdviceOrderElement data structure.
b) Order of advice dispatch blocks at a join point shadow.

Figure 7. Instruction sequence generated for ordered advice units.

A dispatch block for **before** and **after** advice consists of code to:

1. invoke the method `isSatisfied` on each of the dynamic properties,
2. (for non-static advice methods) invoke `getAdviceInstance` on the instantiation strategy,
3. invoke `getValue` on all context values,
4. invoke the advice method.

Each call to `isSatisfied` on a dynamic property is followed by a conditional jump (if the call returns **false**) behind the last instruction of the advice dispatch block. In this case, the execution continues with the next advice dispatch block.

In order to support **proceed**, the dispatch block for **around** advice is slightly different. Instead of generating code for invoking the advice method (item 4 in the structure of dispatch blocks for **before** and **after** advice), the default weaver copies the **around** advice method's implementation into the advice dispatch block.

Inlining of **around** advice can not be performed when the **proceed** occurs in an anonymous nested type in the **around** advice as observed in [19, 2]. There, an implementation based on closures is described that can be used in all situations. A similar implementation in the default weaver is subject to future work.

The inlined code is modified in two ways. First, local variables are re-numbered to avoid interference with local variables of the method containing the join point shadow or other inlined around advice. Second, return instructions are replaced by instructions that jump behind the inlined advice method's code to ensure that the execution continues with the next advice dispatch block after the inlined advice method.

Our AspectJ compiler produces bytecode for around advice methods in a way to facilitate the necessary rewrites. A **proceed** is represented as an invocation to a static method with the same signature as the around advice's method. This method will never be called, though. Whenever such a call is discovered, the weaver replaces it with a "jump to subroutine" (`jsr`) to the first advice dispatch block of the next `AdviceOrderElement`. Before this, the current program counter is stored to enable resuming execution immediately behind the `jsr` instruction. Correspondingly, a `ret` instruction is inserted right behind the last advice dispatch block of an `AdviceOrderElement` for returning to the `jsr`.

The default weaver presented here conforms to the meta-model. But in our implementation, we already treat some of the AspectJ-instantiations of the meta-model specially, thereby showing how a meta-model implementation benefits from the AO concepts being first-class. This implementation provides optimized implementations for two kinds of context values: those that are locally available at the join point, and those that can be evaluated at weaving time.

An example of the first kind is the `Target` context: The weaver simply generates instructions that load it from the call frame instead of generating a call to the `Target` objects `getValue` method. An example of the second kind is the join point signature (e.g., exposed as part of **`thisJoinPointStaticPart`** in AspectJ): A string constant and an instruction for loading it are generated by the weaver.

With these optimizations, the default implementation of the meta-model is comparable to conventional aspect weaving in terms of the runtime performance. Advanced optimizations are also possible as will be discussed in the following section.

4. Alternative Instantiations of the Architecture

In this section, we evaluate the proposed architecture in terms of its flexibility in supporting variability with respect to both, different instantiations of the meta-model (i.e., compilers and language features) and different implementations of the meta-model (i.e., execution environments).

4.1 Different Instantiations of the Meta-Model

Currently, there is a variety of aspect-oriented programming languages offering additional features to AspectJ [28]. We will discuss how some advanced features can be realized as instantiations of our meta-model; more discussion of mapping languages to the meta-model can be found in [27, 26].

To start with, several languages support dynamic scoping of aspects, e.g., an aspect can be active only within certain threads or only for certain base objects. Examples are the languages CaesarJ [29, 30] and JAsCo [13, 14]. In our meta-model, the scope of an aspect can be mapped to a dynamic property: When an aspect is deployed with a specific scope, the aspect is copied and all join point sets of its copied advice units get an additional dynamic property implementing checks for the scope. Afterwards, the copied aspect is deployed.

Aspect-oriented languages also differ with respect to their pointcut languages. We have observed that, for the most part, the join point shadows, i.e., the statically resolvable part, is identical; differences exist in the dynamic properties that can be specified. Several languages allow to define pointcuts in terms of the execution history beyond the abilities of the **`cflow`** pointcut designator of AspectJ. The respective pointcut languages provide expressions to describe "interesting" sequences of join points, e.g., by means of regular expressions [11, 31, 32] or linear temporal logic [12]. When such a sequence is detected at runtime, the pointcut matches.

Join point sequences can be realized as dynamic properties in our meta-model, in a similar way as the **`cflow`** realization discussed in subsection 3.1. `AdviceUnits` are generated and deployed that get notified at the join points participating in the sequence. Internally, the advice units keep track of the already encountered join points, e.g., by updating an automaton. When the dynamic property's `isSatisfied` method is called, it checks the automaton's state.

Another dimension of variability concerns advice ordering. In section 3.1, we discussed how the proposed approach enables to determine the order of advice execution at a given join point using aspect precedence. More advanced strategies can be realized as well. For example, `ScheduleMetaData` objects can also specify conditional inter-advice dependencies, such as, *if advice a and b but not c apply then execute a before b; if, however, a, b and c apply then the order is b before c before a*. Given such specifications, constraint solving techniques, e.g., discussed in [33], could be used to determine the advice' order.

4.2 Different Implementations of the Meta-Model

We already discussed optimizations that can be performed by the weaver, e.g., for the `Target` context. Besides generating more efficient code for accessing context values, alternative weaver implementations may also be able to evaluate dynamic properties statically. For instance, in contrast to our default implementation, the **`within`** and **`withincode`** dynamic properties could be statically evaluated.

There are other kinds of optimizations which are only possible within a virtual machine. In previous work [20], we have implemented virtual machine techniques that speed up dynamic aspect deployment. The idea is to treat join point shadows similar to method calls and use well established speculative optimization techniques for virtual methods [34, 35]. The idea of these techniques is to perform no virtual method dispatch when the just-in-time compiler (JIT) can determine the concrete type of the receiver object. This determination is based on the assumption that certain properties of the application, e.g., the type hierarchy, will not change. The virtual machine can detect changes to these properties, class loading for example, and efficiently replace the non-virtual method dispatch with a full virtual method dispatch [36].

Similarly, these techniques are used in [20] for deploying aspects. The assumption when compiling a join point shadow is that the set of deployed aspects will not change. At the event of aspect deployment, the code of join point shadows is replaced. By using these techniques an efficient implementation of the deployment for the meta-model can be provided.

The evaluation in [20] measured the time for deploying an aspect affecting all calls to public application methods. Deploying this aspect on the SPEC JVM98 benchmark applications [37] with the VM integrated deployment only took 3 ms on average. Other implementations of dynamic deployment averagely took between 229 ms and 3360 ms in the same scenario.

In the Steamloom virtual machine [7], we have experimented with optimized implementations of control flow checks [9] and of advice instance retrieval [10].

For the optimized control flow check, an identifier is assigned to each **`cflow`** pointcut designator defined in the application. Instead of weaving in instructions that execute the control flow check, the weaver flags the place where the check has to be executed and inserts the **`cflow`**'s identifier as a reference to the first-class entity. When the JIT compiler encounters a place where a control flow check is to be executed, it can access the **`cflow`**'s full definition because it is a first-class entity. This allows the JIT compiler to check whether the control flow is always **`true`** or always **`false`** in the currently compiled context. In [9], this optimization as well as others are described in more detail.

In [9] we present a worst case evaluation of the effect of **`cflow`** pointcut designators on single operations. We measured how much the performance of method calls degrades for methods that (a) constitute the control flow referred to by **`cflow`** and (b) only contains a check of the current control flow. The integrated **`cflow`** implementation imposed an overhead of 166% for case (a) and 67% for case (b). The performance loss of other investigated implementations ranged from 498%[6] to 7370% for case (a) and from 687% to 4240% for case (b) in single-threaded applications. For multi-threaded applications our implementation exposed the same performance as in the single-threaded case while the other implementations expose a performance loss of at least 1856 % (case (a)) and 2108% (case (b)).

When using this optimization in a meta-model implementation, the optimized control flow check implementation can use the control flow based `DynamicProperty` implementations as first-class representations of the check. When the JIT compiles an advice dispatch block that contains such a check, it can use the associated join point sets to determine if the check will, e.g., always succeed and omit instructions calling the dynamic property's `isSatisfied` method. If the check can not be omitted, the JIT can generate code for a more efficient check that is executed instead of the call to the default implementation of `isSatisfied`, as has been shown in [9].

In other work, the object model of the virtual machine has been modified to store a table of advice instances to realize optimized access to them [10]. This way, the lookup costs are reduced. The extended virtual machine provides a special bytecode instruction for loading the instance from its storage location in the extended object layout.

The enhanced object model for virtual machines to support per instance aspects can play its strength for **`pertarget`** aspects. In the approach with advice instance tables, the performance of executing an advice from a **`pertarget`** aspect is at least circa one order of magnitude faster than in other investigated approaches [10].

While all these optimizations are promising, they currently lack a common interface to make them available to a wide range of language implementations. The meta-model presented in this paper can act as such an interface.

5. Related Work

The Nu project [5] also aims at providing an interface between compilers and execution environments. For this purpose, two new instructions are provided in the intermediate language (e.g., the bytecode) for associating and disassociating a *pattern of join points* with a *delegate*. The model of this approach is less differentiated and less complete than our meta-model. For example, dynamic properties of join points such as **`cflow`** can not be expressed [38].

In [39], a meta-model to capture the semantics of features in aspect-oriented languages is defined. The meta-model is implemented as an interpreter in the Smalltalk language, called *metaspin*, whereby each computational step is represented as a closure and can be a join point. The aspect sand-box project [40] follows similar goals. The semantics of aspect-oriented languages are expressed as interpreters in the Scheme language. Similar to our approach, both aforementioned approaches represent aspect-oriented concepts as first-class entities. However, these approaches only target language design and the connection to optimized implementations are not considered.

The Reflex project [41, 42] aims at providing an extensible kernel for aspect-oriented programming based on behavioral reflection

[6] We leave the results for the stack-walking implementation out of this discussion, as the performance at **`cflow`** checks is probibitively bad.

by means of a Java embedded language. This kernel has some similarities with the meta-model proposed in this paper. What we call an advice unit, is a *link* in their terminology; join point shadow sets are called *hooksets* and Dynamic properties are called *activation condition* in Reflex and are also first-class objects. However, in the proposed way of using Reflex, i.e., by means of the Reflex kernel language [41], activation conditions have to be specified as blocks of code which hinders re-use. What is more important, Reflex covers only the meta-model part of our proposed architecture. Questions related to exploiting the reflective aspect definitions in the execution environment, including different implementations of aspect deployment, e.g., transactional aspect deployment, are not considered.

The AspectBench Compiler (`abc`) [43, 19] offers a workbench for implementing compilers for aspect-oriented languages. It provides an extensible parser based on the Polyglot framework. Furthermore, interfaces for *shadow types* and *shadow matcher* are provided, which are used for join point shadow search and weaving. Bytecode analyzes and optimizations of the Soot framework are also part of the workbench. Language extensions implement the `ShadowType` and `ShadowMatch` interfaces for new kinds of pointcut designators and extend the parser such that it creates instances of these classes when it encounters a pointcut definition. These objects are passed as first-class entities to the weaver which generates an intermediate code. The analyzes and optimizations provided by Soot work on this intermediate code and can, thus, be reused by language extensions.

The `abc` model is less flexible than our meta-model, e.g., with regard to instantiation strategies, as the methods for loading the receiver object for advice calls are hard-coded for the per clauses defined in the AspectJ language. Also, as the compiler weaves the aspects into the bytecode, the concepts are not first-class in the execution environment, preventing it from making additional optimizations that are not possible to perform at compile time.

The `ajc` compiler from the AspectJ distribution [44] is split into a front-end which parses the AspectJ code and generates pure Java bytecode from it and a back-end which weaves in the aspects. In the first step, aspects and advice are transformed into classes and methods. Non-java constructs such as pointcut declarations are stored as Java bytecode attributes. In a second step, the back-end of the compiler reads these attributes and performs the weaving [2]. Thus, similarly to `abc`, pointcut declarations are passed as first-class entities to the weaver but lose this state after weaving, i.e., before the execution. Also, the interface between the front-end and the back-end is not officially documented and also not extensible.

6. Summary and Future Work

In this paper, we presented an architecture for implementations of aspect-oriented programming languages. Central to this architecture is a meta-model of aspect-oriented language features that decouples the definition of language features from their implementation in a virtual machine.

The meta-model is generic in a sense that a wide variety of current aspect-oriented language concepts can be mapped onto it. This mapping can be expressed in a way that abstracts from optimized implementation issues. This enables language designers to concentrate exclusively on the semantics of language features and yet profit from optimization techniques implemented in virtual machines that adhere to the architecture.

The proposed architecture requires that the front-end of a language implementation, i.e., the compiler, produces code that creates runtime objects which define the aspects according to the meta-model. The back-end, i.e., the execution environment, recognizes these runtime objects and weaves the program accordingly. Since aspect-oriented concepts are expressed as first-class entities via the runtime objects, execution environments are enabled to make sophisticated optimizations.

The architecture is flexible in the sense that compilers and execution environments adhering to it can be flexibly exchanged. Optimizations that are made in special execution environments adhering to the architecture can be be exploited by programming languages also adhering to the architecture. More information and downloads can be found at the project's home page [25].

In future work, we will target three different areas. One area is to improve the default implementation of the model. Currently not all join point shadows provided by AspectJ are supported, namely, exception handlers as well as different variants of object initialization (e.g., **`preinitialization`**). Support for these join point shadows and an implementation of **`around`** and **`proceed`** based on closures will be provided in future versions of the default weaver.

A second area of future work will target other concrete instantiations of the architecture as discussed in section 4. On the one side, we will provide virtual machine optimizations presented in earlier work, as a special implementation of the meta-model. This will include a comprehensive performance evaluation and comparison. On the other side, other aspect-oriented languages will be mapped to the meta-model and respective compilers will be implemented. In this process, the meta-model might need to be refined.

Finally, we we will increase the expressiveness of our model. In concrete we will research possibilities to also capture structural crosscutting in the model. Further, we will investigate support for more advanced join point models and more expressive pointcut languages, e.g., similar to Prolog queries.

Acknowledgments

We would like to thank Michael Eichberg who contributed to the discussion of advice ordering, Michael Hausl for implementing the AspectJ compiler discussed in the paper, and Markus Maus for contributing to the AspectJ instantiation of the meta-model.

This work was supported by the AOSD-Europe Network of Excellence, European Union grant no. FP6-2003-IST-2-004349.

References

[1] Masuhara, H., Kiczales, G.: Modeling crosscutting in aspect-oriented mechanisms. In: 17th European Conference on Object-Oriented Programming (ECOOP). (2003) 2–28

[2] Hilsdale, E., Hugunin, J.: Advice weaving in AspectJ. In: AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development, New York, NY, USA, ACM Press (2004) 26–35

[3] Masuhara, H., Kiczales, G., Dutchyn, C.: Compilation semantics of aspect-oriented programs. In: Foundations Of Aspect-Oriented Languages (Workshop at AOSD 2002). (April 2002)

[4] Hidehiko Masuhara, G.K., Dutchyn, C.: Compilation semantics of aspect-oriented programs. In: Foundations Of Aspect-Oriented Languages (Workshop at AOSD 2002). (April 2002)

[5] Rajan, H., Dyer, R., Hanna, Y., Narayanappa, H.: Preserving Separation of Concerns through Compilation. Technical Report 405, Iowa State University (21 March 2006 2006)

[6] : The AspectJ Programming Guide. `http://www.eclipse.org/aspectj/doc/released/progguide/index.html` (2006)

[7] : The Steamloom Homepage. `http://www.st.informatik.tu-darmstadt.de/static/pages/projects/AORTA/Steamloom.jsp`

[8] : Homepage of the Envelope-Aware Jikes RVM. `http://www.st.informatik.tu-darmstadt.de/EBW-aware`

[9] Bockisch, C., Kanthak, S., Haupt, M., Arnold, M., Mezini, M.: Efficient Control Flow Quantification. In: ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2006). (2006)

[10] Haupt, M., Mezini, M.: Virtual Machine Support for Aspects with Advice Instance Tables. In: First French Workshop on Aspect-Oriented Programming, Paris, France (September 2004)

[11] Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In: OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (2005) 345–364

[12] Stolz, V., Bodden, E.: Temporal Assertions using AspectJ. In: Fifth Workshop on Runtime Verification (RV'05), Electronic Notes in Theoretical Computer Science., Elsevier Science Publishers (2005)

[13] : Jasco homepage. `http://ssel.vub.ac.be/jasco/`

[14] Suvée, D., Vanderperren, W., Jonckers, V.: JAsCo: an Aspect-Oriented Approach Tailored for Component Based Software Development. In: Proc. AOSD 2003. (2003) 21–29

[15] : AspectWerkz homepage. `http://aspectwerkz.codehaus.org/`

[16] Bonér, J.: AspectWerkz - Dynamic AOP for Java. `http://codehaus.org/~jboner/papers/aosd2004_aspectwerkz.pdf` (2003)

[17] : Api specification for package java.lang.instrument. `http://java.sun.com/j2se/1.5.0/docs/api/java/lang/instrument/package-summary.html`

[18] Ekman, T., Hedin, G.: Rewritable Reference Attributed Grammars. Lecture Notes in Computer Science **3086** (January 2004) 147–171

[19] Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Optimising AspectJ. In Hall, M., ed.: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM Press (2005)

[20] Bockisch, C., Arnold, M., Dinkelaker, T., Mezini, M.: Adapting Virtual Machine Techniques for Seamless Aspect Support. In: ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2006). (2006)

[21] Bockisch, C., Haupt, M., Mezini, M., Mitschke, R.: Evenelope-based Weaving for Faster Aspect Compilers. In: In Net.ObjectDays. (2005)

[22] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Reading, Massachusetts (1994)

[23] : Jvmti (java virtual machine tool interface) homepage. `http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/`

[24] Jackson, A., Clarke, S., Bockisch, M.C.C.: Deliver preliminary support for next-priority use cases. Technical Report AOSD-Europe deliverable D80, AOSD-Europe-IBM-80, IBM UK (26 February 2007 2007)

[25] : Homepage of the aspect language implementation architecture (alia). `http://www.st.informatik.tu-darmstadt.de/static/pages/projects/ALIA/alia.html`

[26] Bockisch, C., Mezini, M., Gybels, K., Fabry, J.: Initial definition of the aspect language reference model and prototype implementation adhering to the language implementation toolkit architecture. Technical Report AOSD-Europe Deliverable D72, AOSD-Europe-TUD-7, Technische Universität Darmstadt (27 February 2007 2007)

[27] Jackson, A., Clarke, S., Chapman, M., Dean, A., Bockisch, C.: Deliver Preliminary Support For Top Priority Use Cases. Technical Report AOSD-Europe deliverable D64, AOSD-Europe-IBM-64, IBM UK (30 October 2006 2006)

[28] Dinkelaker, T., Haupt, M., Pawlak, R., Navarro, L.D.B., Gasiunas, V.: Inventory of Aspect-Oriented Execution Models. Technical Report AOSD-Europe Deliverable D40, AOSD-Europe-TUD-4, Technische Universität Darmstadt (28 February 2006 2006)

[29] Aracic, I., Gasiunas, V., Mezini, M., Ostermann, K.: Overview of caesarj. Transactions on AOSD I, LNCS **3880** (2006) 135 – 173

[30] : CaesarJ homepage. `http://caesarj.org/`

[31] Douence, R., Fritz, T., Loriant, N., Menaud, J.M., Ségura-Devillechaise, M., Südholt, M.: An expressive aspect language for system applications with Arachne. In: AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development, New York, NY, USA, ACM Press (2005) 27–38

[32] Vanderperren, W., Suvée, D., Cibrán, M.A., Fraine, B.D.: Stateful Aspects in JAsCo. `http://ssel.vub.ac.be/jasco/media/sc2005.pdf`

[33] Eichberg, M., Mezini, M., Kloppenburg, S., Ostermann, K., Rank, B.: Integrating and Scheduling an Open Set of Static Analyses. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE). (2006)

[34] Arnold, M., Fink, S.J., Grove, D., Hind, M., Sweeney, P.F.: A survey of adaptive optimization in virtual machines

[35] Detlefs, D., Agesen, O.: Inlining of virtual methods. In: 13th European Conference on Object-Oriented Programming (ECOOP). Volume 1628 of LNCS. (June 1999) 258–278

[36] Fink, S.J., Qian, F.: Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In: International Symposium on Code Generation and Optimization (CGO). (2003) 241–252

[37] : SPEC JVM98 homepage. `http://www.spec.org/osg/jvm98/`

[38] Rajan, H., Dyer, R., Narayanappa, H., Hanna, Y.: Nu: Towards an AspectOriented Invocation Mechanism. Technical report, Iowa State University (26 March 2006 2006)

[39] Brichau, J., Mezini, M., Noyé, J., Havinga, W., Bergmans, L., Gasiunas, V., Bockisch, C., Fabry, J., DHondt, T.: An Initial Metamodel for Aspect-Oriented Programming Languages. Technical Report AOSD-Europe Deliverable D39, AOSD-Europe-VUB-12, Vrije Universiteit Brussel (27 February 2006 2006)

[40] Dutchyn, C., Kiczales, G., Masuhara, H.: Aop language exploration using the aspect sand box. In: Aspect-Oriented Software Development (AOSD 2002). (April 2002)

[41] Tanter, É.: An Extensible Kernel Language for AOP. In: Proceedings of AOSD Workshop on Open and Dynamic Aspect Languages, Bonn, Germany (2006)

[42] Tanter, É., Noyé, J.: Motivation and Requirements for a Versatile AOP Kernel. In: 1st European Interactive Workshop on Aspects in Software (EIWAS 2004), Berlin, Germany (September 2004)

[43] Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: abc: an extensible AspectJ compiler. In: AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development, New York, NY, USA, ACM Press (2005) 87–98

[44] : AspectJ homepage. `http://www.eclipse.org/aspectj/`

A Machine Code Model for Efficient Advice Dispatch

Ryan M. Golbeck   Gregor Kiczales
University of British Columbia
{rmgolbec,gregor}@cs.ubc.ca

Abstract

The primary implementations of AspectJ to date are based on a compile- or load-time weaving process that produces Java byte code. Although this implementation strategy has been crucial to the adoption of AspectJ, it faces inherent performance constraints that stem from a mismatch between Java byte code and AspectJ semantics. We discuss these mismatches and show their performance impact on advice dispatch, and we present a machine code model that can be targeted by virtual machine JIT compilers to alleviate this inefficiency. We also present an implementation based on the Jikes RVM which targets this machine code model. Performance evaluation with a set of micro benchmarks shows that our machine code model provides improved performance over translation of advice dispatch to Java byte code.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors–Code Generation, Optimization, Run-time environments

General Terms Languages, Performance

Keywords AspectJ, Aspect-oriented programming

1. Introduction

One of the design goals of ajc [16], a mainstream AspectJ [19] compiler, is to produce no performance overhead over a straight forward hand-coded implementation of the same functionality, as demonstrated by this quote from the AspectJ frequently asked questions:

> We aim for the performance of our implementation of AspectJ to be on par with the same functionality hand-coded in Java. Anything significantly less should be considered a bug [11].

ajc and other byte code rewriting based implementations [22, 23] face inherent performance limitations that stem from the semantics of Java byte code. There are a number of static invariants that are provably true in the AspectJ semantics that cannot be expressed in Java byte code.

Although byte code rewriting implementations have been critical to the adoption of AspectJ, and were the right strategy to employ at the time, the inherit expressibility limitations prevent them from surpassing the design goal quoted above.

In this paper we present a runtime architecture for the AspectJ language which addresses these performance problems by binding of advice to join point shadows in machine code. We refer to our implementation as the AJVM. The contributions of this work are:

- a discussion of the limitations of Java byte code semantics in efficiently expressing AspectJ language constructs;
- a layout for a runtime architecture supporting late-binding of advice to join point shadows;
- a target machine code model for efficient execution of advice; and
- an implementation of this target model together with an evaluation and demonstration of the performance benefits of late-binding advice to join point shadows.

Our implementation achieves better performance than existing implementations of the AspectJ language semantics [6, 24, 16, 22]. It also meets or exceeds the performance of hand-coded comparable functionality, thereby satisfying the performance goals outlined in the AspectJ FAQ quoted above.

This paper is organized as follows. Section 2 discusses byte code rewriting based AspectJ implementations and briefly touches on the problems this strategy causes. Section 3 presents work on virtual machines supporting object-oriented programming and where in the architecture of these systems we can add functionality to enhance support for aspect-oriented (AO) programming [18].

Section 4 presents the primary results of the paper: a high level overview of the target machine code model together with the optimizations available to Just-in-Time (JIT) compilers, but not to byte code translators, that allow us to efficiently express AspectJ advice semantics.

This discussion is followed by the particulars of our implementation and the target machine code model for Intel i386 based processors in section 5. The results of our evaluation and benchmarks are presented in section 6. In sections, 7, 8 and 9 we present related work, future work and conclusions.

2. Byte Code Rewriting

The primary implementations of AspectJ to date are based on a compile- or load-time weaving process which operates on and produces Java byte code.

Compile- and load-time weaving in ajc are essentially the same. Compile-time weaving occurs during translation from AspectJ code to Java byte code, and load-time weaving is a byte code to byte code translation during the loading of the class. Both of these approaches are essentially byte code rewriting techniques. Load-time weaving has more conceptual similarities to the VM-based AO architecture we are developing, so we will draw our parallels to ajc's load-time weaver.

When using ajc's load time weaver, program compilation translates aspects into annotated class files and produces an XML aspect configuration file which describes the aspects that need to be loaded. When a program is executed, the ajc runtime is responsible for consulting the aspect definition file and loading the relevant aspects before classes are loaded; a customized class loader then weaves defined advice into the class byte code during loading.

Operating in this fashion, the static compilation phase of the program compiles aspects, but does not weave them. It produces an appropriate external representation of aspects, together with relevant meta-data, to be loaded by the runtime architecture. When the program is executed, the ajc runtime loads the external format into an internal representation and weaves class methods as they are loaded for advice dispatch.

These byte code rewriting implementations typically compile aspects into Java classes, compile advice bodies into methods associated with the aspect class, and arrange for advice to be executed by weaving method calls at advised join point shadows. This translation throws away information unique to aspects; after weaving, an advice body execution is no longer distinguishable from a normal method call. This loss of semantic information has a performance impact for advice execution, because, as we show below, we lose the ability to prove some static properties of the code.

3. Virtual Machine Architecture

An efficient VM based architecture supporting AO programming is analogous to studied architectures supporting object-oriented (OO) programming. Advice dispatch in an AO program parallels method dispatch in an OO program, and can be supported by a similar sort of architecture.

Like AO programming, some OO programming language implementations started out as pre-processing techniques. As OO techniques became more studied and accepted, these implementations were integrated into their own language compilers and eventually into virtual machine based runtime architectures.

Early VM-based OO architectures [12, 10] had five basic parts: an external format for representing classes that can be loaded into the runtime environment (external representation), an internal format for representing classes in the environment (internal representation), a dispatcher which determines the appropriate methods to be executed at a call-site (dispatcher), a format for laying out compiled code produced by a JIT compiler (target model), and a memory management scheme (memory manager). Note, however, that both the target model and memory manager are optional components. The target model can be omitted in an interpreter based implementation, and the memory management burden can be left to the programmer.

A VM-based environment can integrate additional components into each of these 5 parts of the architecture to provide a VM-based AO environment: additional external and internal representations for aspects, additional dispatch logic for advice execution, a modified layout for compiled code, and special rules for handling aspect instances in memory.

However, there is a rich history of work in VM-supported OO architectures [20, 17] since this initial basic design. Improvements include dynamic optimization compilers with adaptive optimization systems which control the JIT and its decisions during optimization that trade-off compile time and runtime efficiency of produced code. These systems use profiling data accumulated during the running of the program. Some VMs also have the ability to store profiling data on disk for cross-run persistence [2].

We present a VM-based AO implementation architecture that moves AspectJ along the path from language compiler, byte code based implementations to integration into a 5 part VM-based architecture. Our architecture mirrors the progression of previous OO architectures and follows in the foot steps of other VM-based AOP implementations such as PROSE[24], AspectWerkz[6] and Steamloom[4]. Our approach is conceptually similar to ajc load-time weaving: AspectJ code is compiled to Java byte code statically but using a compiler which preserves aspect specific information, and aspects are loaded and their advice executions woven at runtime. The primary difference between our implementation and traditional approaches for the purposes of this paper is that weaving is deferred from load time until JIT time. Waiting until JIT time allows improved performance by making it possible to exploit optimizations that are expressible in machine code, but not in Java byte code.

This approach also appears to enable a revised language semantics in which aspects can be loaded and unloaded dynamically. This paper does not advance AspectJ and AO programming all the way to a full VM-based implementation, it is one step along the path to realising such an implementation. Specifically, we do not integrate advice dispatch into the dynamic optimization system – instead, we weave advice as method calls, and rely on the existing profiling system to continue work as if the running program were purely OO-based.

4. Target Machine Code Model for Dispatch

In this section we elaborate on the mismatches between AspectJ and Java byte code semantics, and present the optimizations available in the JIT that alleviate the resulting inefficiency. The target machine code model[1] presented has few and tightly controlled coupling to the runtime architecture, thereby placing few restrictions on the design space of the supporting infrastructure. This model provides the flexibility required to design the remainder of the architecture with respect to other trade-offs in the VM, such as pointcut matching, class and aspect loading and unloading, and memory management techniques (garbage collection). These are factors that affect the overall runtime performance of a program and the runtime environment without specifically affecting the execution speed of the actual program code itself.

4.1 Common Case Optimizations

Because ajc compiles advice dispatch into Java byte code, it cannot take advantage of static invariants that are true during execution of the program. Consider a simple `before` advice on a dynamic call join point[2] defined in an `issingleton` aspect. At the Java Language Specification [13] level, the semantics of this advice dispatch could be implemented by adding a call such as the following to the join point shadow:

```
A.aspectOf().before$0();
```

ajc produces this equivalent byte code

```
//Method Aspect.aspectOf:()LAspect;
9:  invokestatic    #31;

//Method Aspect.before$0()V
12:  invokevirtual   #34;
```

Walking through this byte code we can see that it does more work than is required, and there are several possibilities for optimization if the target model is native machine code.

The first task is retrieving the aspect instance (byte code 9). This look-up, together with its implicit semantics, is a primary source of overhead in advice execution in ajc because the only ways to get an instance of an object in Java byte code are via `new`, to retrieve the

[1] In the remainder of this paper we use target model and target machine code model interchangeably.

[2] In the remainder of the paper we will use the more concise *join point* to mean dynamic join point.

instance from a variable, or to have it returned from some other method call. In this case, ajc retrieves the aspect using a static method call.

The body of the `aspectOf()` method is a reference to a static variable. We can reasonably expect a JIT compiler to inline the `aspectOf()` call because it is static and short. However, in the best case, inlining `aspectOf()` still results in a static variable reference which for typical VM memory management approaches becomes a constant load from memory in machine code.

However, because of the semantics of `issingleton` aspects, it is often possible to know the exact location in memory where the aspect instance is stored. This depends on the particulars of the memory management component of the architecture, but memory managers typically have a region of dedicated unmovable objects whose lifetimes correspond to the lifetime of the program [3]. By arranging to have the aspect instance allocated in this region, we can load it with one native instruction which loads the address of the instance as an immediate constant.

The next instruction, byte code 12, is composed of three different operations. The first performs a null check on the instance that was returned by `aspectOf()`. This is necessary because there is no way to express the fact that the aspect is guaranteed to exist. An aspect-aware JIT can eliminate this test because the aspect loader can guarantee that the aspect instance has been created before any advice is invoked.

Second, it causes a load from the aspect class' virtual function table. This load can also be eliminated by an aspect-aware JIT, because an aspect-aware VM can guarantee the exact type of the aspect corresponding to a specific advice call. This guarantee is sound because it is the aspect that arranges for its advice to be called, it is not a normal virtual method call.

The last operation is the actual call instruction which invokes the synthetic method that holds the body of the applicable advice. This call is necessary in any implementation for the advice body to be executed at all. Since advice bodies are compiled into standard methods internally, this advice call can be inlined like any standard method call. However, the runtime type of an instance cannot be proven at compile time; hence, when inlining the advice call a standard JIT compiler must place guards around the inlined method to check that the runtime type corresponds to the method that was inlined. But, in the case of advice body execution, an aspect-aware JIT compiler can prove the exact runtime type during code generation, and the guards can be eliminated for the same reason that the virtual function table need not be consulted.

All three of these optimizations are based on a combination of semantic invariants of AspectJ and specific information that, although inexpressible in byte code, is expressible in machine code and can reasonably be known inside the JIT.

4.2 Dynamic Instances and Residues

The discussion above covers the common case for advice execution, but AspectJ also supports more complicated aspect instantiation rules and pointcut semantics.

AspectJ supports the `perthis`, `pertarget`, and `percflow` clauses for aspect instantiation. In ajc, and similar byte code rewriting implementations, this difference is contained behind the static `aspectOf()`, or similar, call. In these cases, the location in memory of the aspect instance is not statically determinable, even at JIT time. Therefore, the aspect aware JIT can elect to produce essentially the same code that a byte code weaver would produce: a static call to a runtime method which will retrieve the proper aspect instance. However, it is still possible for the VM to guarantee the precise type of the aspect, and that the result of the static call is not null. So, avoiding the aspect instance null check and virtual function look-up is still possible.

Note, however, that the benefits of these optimizations are likely to be over-shadowed by the dynamic cost of the more expensive aspect look-up. It is possible to implement `perthis` and `pertarget` more efficiently by storing a reference to the aspect instance directly with the affected object in memory. This approach is similar to ajc which uses inter-type declarations to introduce a field into the object.

Dynamic residues are either automatically generated, as in the case of `cflow` conditionals, and `args`, `this`, and `target` type checks, or they are user-generated, as in the case of the `if` pointcut. In both cases, there is no additional static information during JIT compilation that provides any additional opportunities for optimization over the standard Java byte code optimizations thus these concepts are already cleanly expressible in byte code.

So, these optimizations work for common case advice dispatch, but more dynamic look-ups can overshadow the improvements. However, these checks can be optimized in other ways such as the way abc optimizes `cflow`, or using other efficient VM-based approaches such as in [5]. Further, these dispatch optimizations do not hinder the efficient implementation of the dynamic features of the language, they are in fact orthogonal, and other optimizations that perform non-local analysis can still be applied.

4.3 VM Architecture Inter-Dependencies

As mentioned at the beginning of section 4, the optimizations presented tie the compiled code to the rest of the architecture in limited and tightly controllable ways. Since the static part of each pointcut is compiled directly into native code, and the dynamic residues of the pointcut have been inlined, we have only two dependencies.

The direct dependency is on the runtime representation of the aspect; some of the optimizations above require that the aspect is instantiated and stored in a known location before the advice runs, so this optimization is only available when the runtime architecture will not move the aspect instance for the purposes of garbage collection. In practice, this is a reasonable constraint, because this optimization is performed on `issingleton` aspects only, which are created once and have a lifetime matching the lifetime of the program. Aspects must also be represented internally as annotated classes; the aspect instances must be laid out like object instances. Maintaining this layout makes advice invocation the same operation as method invocation.

The indirect dependency exists because of the ability to load aspects during the execution of the program. Since the machine code for a method can be generated before an aspect is loaded, that machine code becomes stale if the new aspect contains advice that must weave into join point shadows in the compiled method. This dependency causes a restriction on the runtime architecture to ensure that stale methods are not executed; they must be either edited or recompiled if they are stale.

This is not an issue in the current static semantics of AspectJ because all aspects must be loaded before classes are loaded into the VM, and so there will never be stale methods. However, one of the potential advantages of a runtime architecture is to support more dynamic behaviour than is specified in the AspectJ language. Stale methods would result from aspect loading in any such runtime system, because it cannot be known when a method is compiled whether it will be advised by an aspect that is not yet loaded. So, this design constraint on the architecture will be present in any runtime system which employs a JIT compiler and provides dynamic aspect loading semantics. Further, there are semantic ambiguities raised by allowing dynamic deployment that need to be resolved. One possible resolution is the `deploy` construct in Caesar [21].

```
 9 ia32_call   AF CF OF PF SF ZF = <[edi(Lorg/vmmagic/unboxed/Offset;)]+1191182812>DW,
                                    static"Aspect.aspectOf ()LAspect;"
12 ia32_mov     ebp([Ljava/lang/Object;) = <[eax(LAspect;)]+-12>DW (t13sv(GUARD))
12 ia32_add     esp(I) AF CF OF PF SF ZF <-- -4
12 ia32_call    AF CF OF PF SF ZF = <[ebp([Ljava/lang/Object;)]
                                   +[edi(Lorg/vmmagic/unboxed/Offset;)]>DW (<TRUEGUARD>),
                                  virtual"Aspect.before$0()V", eax(LAspect;)
```

Figure 1: Machine code produced for AspectJ byte code weaving for a simple before call advice

```
-11 ia32_mov  ebp([Ljava/lang/Object;) = <0+1459722656>DW (<TRUEGUARD>)
-11 ia32_mov  eax(I) = 0x570199ac
-11 ia32_add  esp(I) AF CF OF PF SF ZF <-- -4
-11 ia32_call AF CF OF PF SF ZF = <[ebp([Ljava/lang/Object;)]+60>DW (<TRUEGUARD>),
                                 virtual_exact"Aspect.before$0()V", eax(LAspect;)
```

Figure 2: Machine code produced by AJVM for a simple before call advice

5. Implementation

Our work to date is focused on the generation of efficient woven machine code. To enable this generation we have developed initial, simple implementations of the rest of the architecture (external and internal representations, memory management and dispatch). These can be thought of as simply loading pre-compiled aspects in a way similar to ajc's load-time weaving support: we load aspects stored in class files annotated with meta-data representing the additional data that is unique to aspect definitions.

We have implemented our architecture in the JikesRVM [7]. This implementation serves to validate the target model and overall architecture presented above. Our basic architecture and types of optimizations are applicable to other VM-based architectures as outlined in the previous section, although there are, of course, numerous implementation details critical to performance that are tightly coupled to the JikesRVM.

The JikesRVM has no interpreter; it uses two JIT compilers to translate Java byte code directly to native machine instructions. One of these compilers is the baseline compiler; a very fast compiler that produces code that exactly simulates the Java byte code stack machine. When the VM has determined that the cost of optimizing a method is justified, it invokes the optimizing compiler [9] to recompile that method.

Our current implementation supports before advice on both execution and call join points, in both the baseline and optimizing compiler.

In the baseline compiler, our weaver is integrated into the direct translation to native code. It checks each join point shadow against the registered set of advice and pointcuts. If a match can be determined statically, then direct calls to the appropriate advice body are inserted as dictated by the advice type (before, after, etc); any dynamic residues (such as cflow testing), are implemented as a conditional branch based on runtime information. No attempt is made to inline advice calls at this point. However, aspect instance look-up is still optimized as discussed in the previous section, by hard-coding the singleton aspect addresses into the machine code. Since the baseline compiler has little effect on frequently executed methods, our optimizations to the baseline compiler have little effect on the steady-state, long-running performance of most programs in which advised shadows execute frequently because the methods will be recompiled by the optimizing compiler.

The weaver integration into the optimizing compiler is more complicated because the optimizing compiler has several rounds of optimization and translation and is controlled by an adaptive-optimization system (AOS) [1] within the VM. The optimizing compiler uses several levels of intermediate representations. Our weaver operates during the translation from Java byte code to a high-level intermediate representation (HIR). Operating at this level gives us sufficient expressibility to address memory directly and to remove null checks and virtual function table look-ups, but, at the same time, the weaving takes place before many optimizations, such as copy-propagation and register allocation. Additionally, the inlining of advice method calls is determined by the JikesRVM adaptive-optimization system (AOS) framework. The AOS framework uses profiling data accumulated during the run of a program, in both baseline and optimized code, to drive decisions during the optimization of a method. One of the decisions affected by the profiling data is whether or not a method gets inlined.

Consider, again, the byte code produced by ajc from the previous section of a simple before advice on a call join point.

```
//Method Aspect.aspectOf:()LAspect;
9:  invokestatic    #31;

//Method Aspect.before$0()V
12:  invokevirtual   #34;
```

The JikesRVM optimizing compiler translates this code into the machine intermediate representation (MIR) shown in figure 1. MIR is the last intermediate representation before native code, and it is translated to native instructions in a straightforward manner which includes the insertion of necessary null checks and constant computation.

Looking at the code we see that the call associated with byte-code 9 retrieves the aspect instance and stores it in eax. The first instruction associated with byte code 12 moves the aspect class TiB (type information block) into `ebp`. The TiB contains the virtual method table for the class associated with the aspect. This instruction also has associated with it a guard, `t13sv`. This guard is a null check guard which is causing a null check to be output before referencing eax, which could be null.

Secondly, notice that the call instruction computes an offset from `ebp` to locate the code array to execute. This computation is the virtual function table reference.

The MIR produced by the AJVM for the same advice is shown in figure 2.

There are no associated byte code indices with this advice call. Advice calls are annotated with the special byte code index -11 to denote that they are part of the runtime architecture, and not the program being run.

Secondly, note that both `ebp` and `eax` are loaded with immediate hard-coded values. These refer to the virtual function pointer table and the aspect instance address respectively. In the call to the before advice, the pointer to the virtual function table is a constant loaded into `ebp` with constant offset (60), because we have proven that the aspect type is exact it is not necessary for us to determine this pointer dynamically. So, when this call is translated into the final native code, these constants can be computed statically into a constant address for the method.

Finally, note that all the guards in these instructions are "true guards." They generate no additional runtime checks, including the instruction that populates `eax` with a constant value. So this advice call generates no runtime null check.

From these actual generated instructions, we can see that there are no dependencies on the rest of the runtime architecture other than those mentioned in the previous section. That is, hard-coding the aspect instance address directly into the machine code depends on the fact that the aspect instance is not moved; however, there are no further dependencies on how aspects, pointcuts, and look-up tables are managed. The machine code is an exact representation of the advice and pointcut; there are no additional data structures that must be referenced, and so the representation of this information in the rest of the architecture is unconstrained.

Further, although the specific details of the implementation are specific to the JikesRVM, any VM implementation of this architecture must have these essential pieces in its implementation: virtual function look-ups, instance look-ups, type check guards, and null checks. Therefore, our approach for weaving is not specific to the JikesRVM, and it could be integrated into any VM containing a JIT compiler.

6. Results and Evaluation

This section presents the setup of our experimental benchmark suite, and evaluates the results we get from them.

6.1 Experimental Setup

We use a number of micro benchmarks to measure the performance of our target model. The micro benchmarks are configured to run until the VM reaches a steady-state. That is, we run the benchmarks a number of times without measuring performance so that the optimizing compiler has sufficient opportunity to optimize the code that we are measuring.

We start from JikesRVM version 2.4.4. The VM is built using its production build configuration which enables all JIT optimizations, compiles the entire VM with the optimizing compiler, and disables runtime assertions. The AJVM is a modified version of JikesRVM v2.4.4 built the same way. We use the timing measurement utilities from the Java Grande suite [8] of benchmarks to measure the execution time of our tests.

Our benchmarks were run on a Intel Pentium 4 3Ghz machine with 1GB of memory running SuSE Linux 10.1 with kernel version 2.6.16.

We ran benchmarks using call and execution join points on an unadvised and advised Fibonacci computation, and the cross product of the call and execution join points of a trivial method with no advice, and trivial advice with no dynamic values, and with this, target, args, and cflow dynamic values.

Figures 3 and 4 show the class, AClass, and aspect, AnAspect, used in all the benchmarks aside from the Fibonacci benchmarks. Figure 4 shows advice from all the benchmarks, but only one of these is active at any given time; they are just shown together here for brevity. The individual differences in each benchmark are explained below.

```
class AClass {

  int counter = 0;

  public void m1(int i) {
    counter++;
  }

  public int fib(int i) {
    if(i <= 1)
      return 1;
    return fib(i - 1) + fib(i - 2);
  }

  public void cflowCrossing(int numRepeats) {
    for(int i = 0; i < numRepeats; i++) {
      m1(i);
    }
  }

  public void test(int numRepeats) {
    for(int i = 0; i < numRepeats ; i ++)
      m1(i);
    //fib(6);
  }

  public void BenchRun(int numRepeats,
                       double primingTime) {
    double time = 0.0;
    JGFInstrumentor.addTimer("BCTimer",
                             "jp executions");
    while(time < primingTime) {
      JGFInstrumentor.resetTimer("BCTimer");
      JGFInstrumentor.startTimer("BCTimer");
      test(numRepeats);
      JGFInstrumentor.stopTimer("BCTimer");
      time += JGFInstrumentor.readTimer("BCTimer");
      JGFInstrumentor.addOpsToTimer("BCTimer",
                                    (double)numRepeats);
    }
    JGFInstrumentor.printTimer("BCTimer");
  }
  public static void main(String [] argv) {
    int numRepeats = 10000000;
    int primeForS = 10;
    if(argv.length > 1) {
      numRepeats = Integer.parseInt(argv[0]);
      primeForS = Integer.parseInt(argv[1]);
    }
    (new AClass()).BenchRun(numRepeats, primeForS);
  }

}
```

Figure 3: The Class and Driver used in the micro benchmarks

The graphs in figure 5 show the results of all the benchmarks. Each graph measures the time in seconds that it took each implementation to finish executing the benchmark. All times are normalized to ajc compiler implementation.

The following abbreviations are used in the graph: TM for Trivial Method, TA for Trivial Advice, and Fib for Fibonacci.

The Trivial Method benchmark is a control benchmark to calibrate normal virtual method calls in the JikesRVM, on our platform. In this case, AnAspect is not loaded at all, and `AClass.m1()`, a method which increments a counter, is run repeatedly in a loop. This benchmark is used to ensure that all implementations perform equally well when there are no applicable advice in the system.

The second benchmark defines a trivial advice, which also increments a counter on the call or execution join point of the trivial

```
public aspect AnAspect {

  private int counter = 0;

  /* Trivial Advice (TA).
   * Used in TMTA and FibTA benchmarks
   */
  before ():
  call/execution(void AClass.m1/fib(..)) {
    counter++;
  }

  /* Trivial Value Advice (TAwVAL).
   * Used in this, target, args.
   */
  before (int x):
  args (x)
    && call/execution(void AClass.m1 (..)) {
    counter += x;
  }

  before(AClass obj):
  this/target (obj)
    && call/execution(void AClass.m1 (..)) {
    counter = obj.counter ;
  }

  /* Trivial Advice Dynamic residue dispatch (TADyn)
   * Used in perthis, inside cflow, outside cflow
   */
  before () :
  cflow (call/execution(void AClass.cflowCrossing()))
    && call/execution(void AClass.m1(..)) {
    counter++;
  }
}
```

Figure 4: The Aspect used in the micro benchmarks

method, and the time measurement of the loop is retaken. In this case, the Aspect is deployed in the system.

The Fibonacci benchmark measures the execution time of computing Fibonacci numbers in a loop without any advice. In this case, we call `AClass.fib()`, and AnAspect is not deployed in the system. This benchmark ensures that each implementation performs equally well where there are no aspects present. The second Fibonacci benchmarks introduces the aspect to the system and retakes the measurement.

Both the Trivial Method and Fibonacci use the Trivial Advice defined in the aspect in figure 4.

Micro-measuring both the trivial method and the Fibonacci advised methods allows the numbers to reflect different possible optimizations. Specifically, Fibonacci is not amenable to inlining because it is a recursive function.[3] On the other hand, calls to the trivial method are straight forward to inline, and we can reasonably expect a JIT to do so. Measuring both of these cases ensures we get an accurate reflection of the cost of the advice dispatch, not how well the optimizing compiler inlines method calls.

The second section of the benchmarks compares the efficiency of the different implementations by measuring the time it takes to execute join points where dynamic information is needed as part of advice dispatch. We show measurements that pick out the `this` object, the `target` object, and the `args` of the method call. In these cases, one of the advice from the Trivial Value Advice in figure 4 is used.

[3] It is amenable to loop-unrolling style optimization, however our benchmark runs it at a depth (6) that we believe is sufficient enough to discourage unrolling the entire recursion.

The last section of benchmarks shows two more complicated constructs in AspectJ: `perthis` aspect instantiation, and the `cflow` pointcut. The `perthis` benchmark is the same as the trivial method and trivial advice benchmark mentioned above except that the aspect is instantiated by the `perthis` instantiation rules.

In benchmarking the `cflow` pointcut we used two different benchmarks. The `cflow` pointcut is shown in the last section of figure 4 and is used in both cflow benchmarks, inside cflow and outside cflow. These tests repeatedly call the `AClass.m1()` method and use the same advice as in the previous cflow benchmark. In the first case, these calls are done within the control flow of the `cflowCrossing`, and in the second case they are not. These two benchmarks measure the cost of any sort of runtime checking required to determine the control flow context. Note, however, that this runtime checking is not specifically part of the advice dispatch, but it does demonstrate that our implementation does not hamper the efficient implementation of dynamic residues.

The full numbers from each benchmark have been included in the appendix A in table 1.

6.2 Evaluation

First we notice that there is little variation in the running time of all 4 of the benchmarks which are executed with no advice in the system: Trivial Method and Fibonacci for both call and execution join points. This fact is important because it demonstrates that integrating advice dispatch into the JIT does not slow down unadvised method dispatch.

Secondly, also in the Trivial Advice benchmarks, we see that the AJVM performs consistently better in all four cases that include advice in the program: both Trivial Method plus Trivial Advice and Fibonacci plus Trivial Advice. These results show that advice dispatch overhead is recovered whether or not the JIT can inline the advised virtual method call. Looking at Table 1 of absolute execution time in appendix A, we can see that the improvements range from 6% to 8.6% for Trivial Method plus Trivial Advice and 2.3% to 5.1% for the Fibonacci plus Trivial Advice.

The fact that the By Hand implementation does better than AJVM in the Trivial Method plus Trivial Advice (TM&TA) call and execution benchmarks is due to differences in the way the optimizing compiler unrolls the loops in `AClass.m1(..)`. The loops are unrolled differently because the code the compiler operates on is different between the two implementations, and hence different decisions are made. Most notably, AJVM does not output null checks for the aspect instance, but the By Hand implementation does.

However, our optimizations do improve on the advice dispatch for this benchmark demonstrated by Figure 6. This graph shows the running time of Trivial Method plus Trivial Advice for just the AJVM and By Hand implementations with the optimizing compiler turned off. We can see that our optimizations improve on the By Hand implementation when no other optimizations are applied. Therefore we can expect that integrating knowledge of advice dispatch into the dynamic optimizing compiler could further improve its performance by directing loop unrolling and other optimizations.

The second row of graphs Figure 5 show the costs of executing Trivial Advice that use a dynamic value as one of their arguments. These results show slightly better improvements of the AJVM over ajc than in the Trivial Advice cases for the `this` and `target` pointcuts. We do see more significant improvements in the `args` pointcut. This gain can be also be attributed to different loop unrolling strategies used by the optimizing compiler. In this case, the differences between the aspect instance look-up implementations caused the optimizing compiler to choose a more efficient loop unrolling layout for the AJVM.

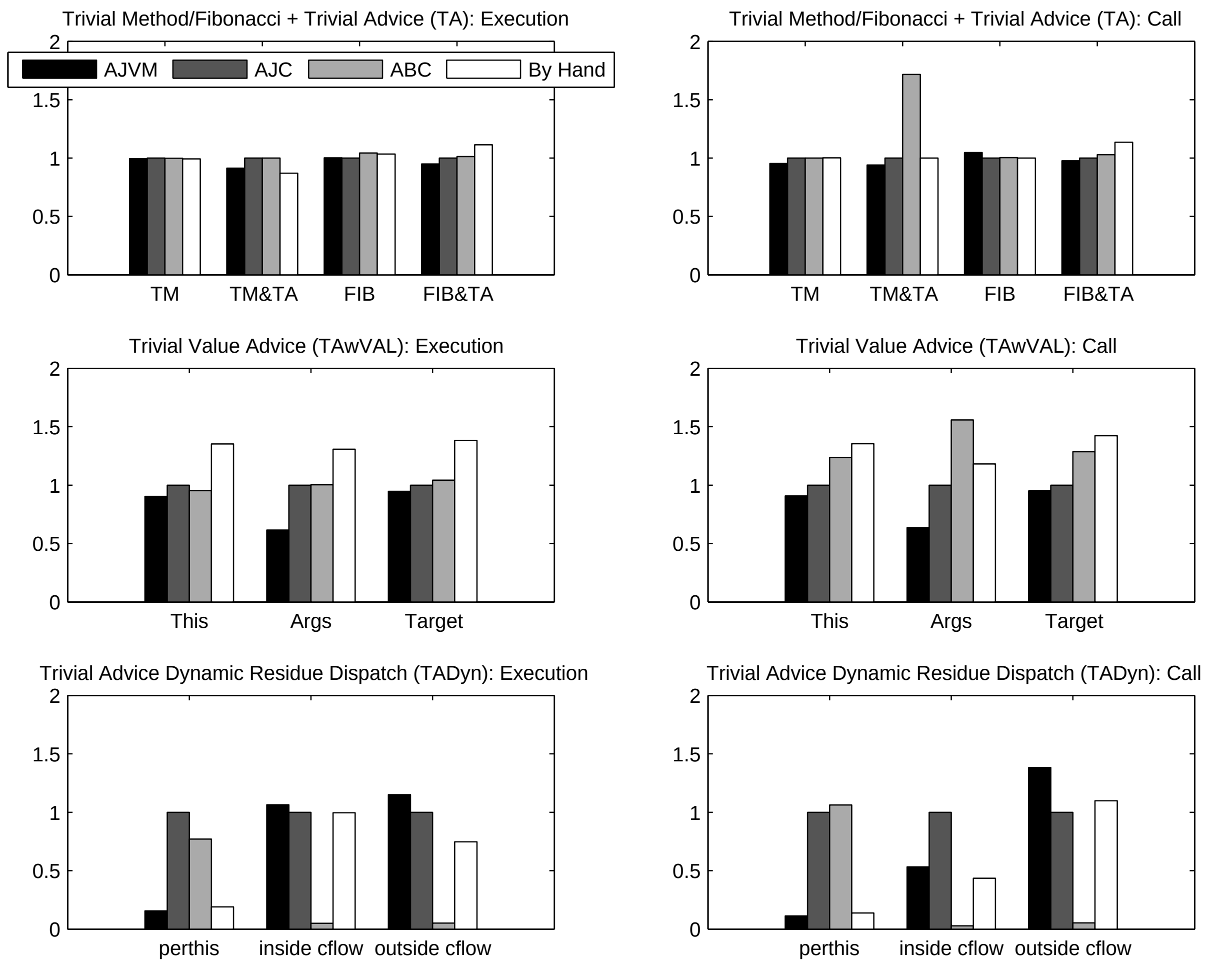


Figure 5: Execution time in seconds normalized to ajc TM=Trivial Method; TA = Trivial Advice; FIB = Fibonacci

The perthis benchmark shows that we can achieve much greater performance when the VM understands how aspect instances should be managed. The AJVM uses a private field in affected objects to store the aspect instance associated with that object in the case of `perthis` aspect instantiation. In this case the aspect-aware JIT outputs a simple instruction to retrieve directly the instance pointer that is stored with the object. A null check determines if the aspect has already been created. However, ajc uses inter-type declarations and an interface on the object to manage the aspect instance associated with the pointcut. This management adds considerable overhead that is avoided in the machine code model.

While looking at the cflow benchmarks, it is important to realise the difference between the strategy of the cflow implementation, and the actual advice dispatch. The AJVM implements optimizations that improve the advice dispatch, and in the case of cflow, it employs the same strategy (thread local counters) as ajc to determine at runtime whether the advice body needs to be executed. In this case, the cost of maintaining the counters overshadows the advice dispatch, so that the AJVM and ajc come out with similar, but varying results. This variability stems from the subtle interaction between the dynamic optimizing compiler and the cflow implementation strategy, rather than the cost of advice dispatch.

Furthermore, we see that abc's non-local strategy for cflow optimization has a very significant increase in the performance of the micro benchmarks. This optimization can prove, in some cases, that no thread local counter (or stack) need to be consulted or managed at all, and hence we can remove all runtime checks guarding the advice dispatch. There is no reason why the strategy used in [5] could not be integrated into AJVM to replace its naive thread local counter strategy. However, in this paper we focus on recovering overhead caused by the mismatch between AspectJ semantics and Java byte code specifically.

The cflow benchmarks, do, however demonstrate that the AJVM is comparable to the ajc implementation. Therefore, we can conclude that our advice dispatch optimizations do not preclude further orthogonal optimizations related to the dynamic features of the language.

7. Related Work

Object-Oriented runtime architectures are well studied. This work is discussed in detail in section 3, and so we will not repeat it here.

Runtime support for aspects is a natural progression as AOP becomes more widespread and adopted; AspectJ would not have been adopted if the first implementation involved a customized VM.

The first implementations of AspectJ were based on preprocessing; this was followed by byte code rewriting based ap-

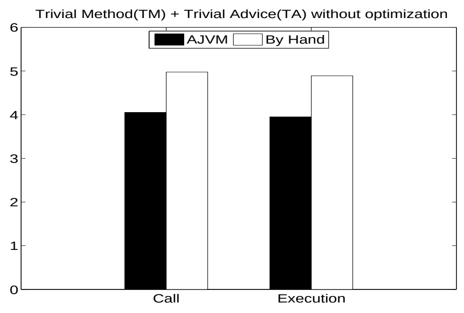


Figure 6: Trivial Method plus Trivial Advice execution time in seconds without dynamic recompilation and optimization

proaches [19, 22]. All of these sorts of implementations suffer from the problem that the output semantics are constrained to a language that is too high level to be optimally efficient.

Byte code rewriting implementations typically compile aspects into classes, advice bodies into methods, and handle advice execution by inserting appropriate dynamic checks and method calls into the program being rewritten. However, byte code rewriting implementations suffer from a similar problem to pre-processing techniques because the Java byte code semantics do not include instructions that can express cleanly AspectJ semantics.

AspectWerkz [6] implements an AspectJ-like language on top of traditional Java using a separate XML file, or embedded annotations to specify advice and pointcuts. AspectWerkz, like ajc, can use a static or load time weaving approach together with a runtime library. However, AspectWerkz makes heavy use of delegation and reflection on advised join points which make it simple to make changes to the aspect and advice (not, however, pointcuts) at runtime. This method of weaving introduces extra indirection costs associated with advice execution that significantly hamper its performance. Our work instead focuses on efficient advice dispatch which can be used to support an efficient dynamic weaving system like AspectWerkz.

Prose [24], unlike ajc or AspectWerkz which use a modified class loader together with a runtime framework on top of standard Java VMs, adds hooks into the VM so that an AOP system could be implemented efficiently on top of it. Prose works by instrumenting the JikesRVM baseline JIT compiler to weave stubs into each join point shadow as the code is compiled. Each stub calls out to the AOP engine which can arrange for advice to be executed or not as required. This approach differs from ours in that we instrument only the join point shadows which can have advice associated with them, and further, we do not weave stub method calls to these join points, but rather inline the code to execute the advice body or the dynamic residue required for any runtime checking.

Steamloom [4, 15] moves advice weaving into the virtual machine. However, it still implements advice dispatch using byte code rewriting. The advantage that Steamloom gains by doing the byte code rewriting in the virtual machine is the added dynamism available because the virtual machine understands what aspects and advice are. This allows Steamloom to achieve greater cflow performance in some cases. However, the limits of byte code rewriting still constrain Steamloom. As described in [14], Steamloom seeks to partially address this by adding a new internally used byte code to the JIT to improve the performance of aspect instance look-up.

AspectWerkz, Prose, and Steamloom are approaches for dynamic weaving AOP systems. The work we present in this paper continues on the path created by these implementations by moving advice one step further into the JIT compiler.

8. Future Work

Achieving efficient advice dispatch in a runtime environment raises many questions regarding the architecture and implementation of the rest of the runtime environment supporting AspectJ: the external and internal representations for aspects, the dispatch and weaver, and memory management. These questions parallel the basic design of VM-based OO architectures, and lead to two basic avenues of future work: the efficient design and implementation of the supporting architecture, and revising the semantics of AspectJ to accommodate the inherit dynamic properties enabled by this architecture.

The internal representation of aspects affect all other parts of the architecture, and it determines the primary trade-offs for space and time efficiency when looking up aspect-related data. The algorithms and representations used define how quickly pointcuts can be matched or searched, and these operations are critical to support the efficient operation of the dispatcher and JIT compiler. This overhead can be a primary concern in short lived programs that require low start-up times in which there is less time for pointcut related computations to be amortized over the life of the program, and for programs that dynamically load and unload aspects during the life of the program.

The internal representation of aspects further drives the design of the external representation so that aspects can be loaded quickly from the external format. It is conceivable that a static AspectJ compiler could pre-process and layout advice and pointcuts to minimize the work that needs to be done during aspect load time for conversion to the internal representation, and for determining which previously loaded methods must be modified because of the new advice.

Further, as mentioned in section 3, we have implemented an aspect-aware JIT compiler, but not an aspect-aware adaptive-optimization system. There could still be performance gains by integrating advice dispatch into the AOS. These gains are especially likely when optimizing dynamic residues from runtime type matching for `args`, `target` and `this` pointcuts, and cflow checking. Integrating advice dispatch further into the AOS moves aspect-aware runtime environments further along the path followed historically by OO runtime environments.

As further validation, we intend to work with a commercial JVM vendor to replicate our experimental results on a production quality JVM.

Additionally, a proper runtime architecture potentially enables more dynamic language semantics. This dynamism can include the ability to load and unload aspects, control the activation of advice, and possibly allow the ability to define pointcuts and advice at runtime. But, AspectJ's semantics do not wholly apply to the new environment, and several important semantic questions need to be addressed. These include dealing with concurrency issues and advice activation. In these cases, there are subtle issues that arise with atomicity of advice deployment, and how the deployment interacts with threads that have newly advised live join points on their stacks. Many of these questions have been addressed in Caesar [21] using their `deploy` construct.

There is work to be done in determining whether inter-type declarations, or other join point models, should be supported by a runtime environment, or if they should only be a feature of a static byte code compiler. These problems directly affect AspectJ semantics, and they will need to be addressed to maintain consistent language semantics when deployed in a runtime architecture.

Finally, our current implementation supports only before advice on call and execution join points. The remaining join points are not very conceptually different than call and execution. However, advice types like around are significantly different and could benefit significantly from integration into the JIT.

9. Conclusion

We have discussed the semantic mismatches between AspectJ and Java byte code. These mismatches stem from the inability to express the exact type of a runtime object, or to provide guarantees that an object exists to avoid null checks at runtime in Java byte code. Although the inability to express these concepts does not restrict the byte code rewriting based implementations of AspectJ functionally, we have shown that they do impose performance constraints.

We have presented a machine code model that can be targeted by virtual machine JIT compilers that alleviates these inefficiencies. We verified our claim that this model can be targeted by a JIT compiler by providing an implementation based on the JikesRVM in which we modify its baseline and optimizing JIT compilers to target our machine code model. Further, we have shown that both the target machine code model, and our implementation, rely on only two reasonable constraints on the supporting runtime architecture; this fact allows us to pursue future work on more fully integrating AspectJ into a runtime environment.

Finally, we have verified that these performance constraints are alleviated by our target model by showing the results of a suite of micro benchmarks which compare the running times of advice dispatch in different scenarios across different weaving implementations.

Our work progresses along the same path that OO runtime architectures progressed in the past 25 years. We hope that by using OO architecture work as a guide, we can quickly bring VM level support for languages like AspectJ to a similar state.

Acknowledgments

This research was partially supported by the IBM Centers for Advanced Studies and the Natural Sciences and Engineering Research Council (NSERC).

A. Results

Table 1 shows the absolute numbers produced from the benchmarks discussed in section 6. The results show the absolute running time in seconds of each of the benchmarks on each of the implementations.

References

[1] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 47–65, New York, NY, USA, 2000. ACM Press.

[2] Matthew Arnold, Adam Welc, and V. T. Rajan. Improving virtual machine performance using a cross-run profile repository. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 297–311, New York, NY, USA, 2005. ACM Press.

[3] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 137–146, Washington, DC, USA, 2004. IEEE Computer Society.

[4] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual Machine Support for Dynamic Join Points. In *AOSD '04: Proceedings of the 3rd International Conference on Aspect-oriented Software Development*, pages 83–92, New York, NY, USA, 2004. ACM Press.

[5] Christoph Bockisch, Sebastian Kanthak, Michael Haupt, Matthew Arnold, and Mira Mezini. Efficient control flow quantification. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 125–138, New York, NY, USA, 2006. ACM Press.

[6] Jonas Bonér and Alexandre Vasseur. AspectWerkz. `http://aspectwerkz.codehaus.org/index.html`.

[7] Bowen Alpern and C. R. Attanasio and Anthony Cocchi and Derek Lieber and Stephen Smith and Ton Ngo and John J. Barton and Susan Flynn Hummel and Janice C. Sheperd and Mark Mergen. Implementing jalapeo in Java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 314–324, New York, NY, USA, 1999. ACM Press.

[8] Bull. A benchmark suite for high performance Java. *Concurrency, practice and experience*, 12(6):375, 2000.

[9] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño dynamic optimizing compiler for Java. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 129–141, New York, NY, USA, 1999. ACM Press.

[10] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302, New York, NY, USA, 1984. ACM Press.

[11] Frequently Asked Questions about AspectJ. `http://www.eclipse.org/aspectj/doc/released/faq.html`, 2006.

[12] Adele Goldberg and David Robson. *Smalltalk-80: Ihe Language and its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[13] James Gosling, Gilad Bracha, Bill Joy, and Guy L Steele. *The Java Language Specification*. Addison-Wesley Professional, 2000.

[14] M. Haupt and M. Mezini. Virtual Machine Support for Aspects with Advice Instance Tables. *L'Objet*, 11(3):9–30, 2005.

[15] M. Haupt, M. Mezini, C. Bockisch, T. Dinkelaker, M. Eichberg, and M. Krebs. An Execution Layer for Aspect-Oriented Programming Languages. In J. Vitek, editor, *Proceedings of the First International Conference on Virtual Execution Environments (VEE'05)*, pages 142–152, Chicago, USA, June 2005. ACM Press.

[16] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35, New York, NY, USA, 2004. ACM Press.

[17] Urs Hölzle and David Ungar. A third-generation self implementation: reconciling responsiveness with performance. *SIGPLAN Not.*, 29(10):229–243, 1994.

[18] Gregor Kiczales and Erik Hilsdale. Aspect-oriented Programming. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, page 313, New York, NY, USA, 2001. ACM Press.

[19] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[20] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specificaiton*. Addison-Wesley Longman, Inc., 1997.

[21] Mira Mezini and Klaus Ostermann. Conquering aspects with Caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM Press.

[22] Pavel Avgustinov and Aske Simon Christensen and Laurie Hendren and Sascha Kuzins and Jennifer Lhoták and Ondřej Lhoták and Oege de Moor and Damien Sereni and Ganesh Sittampalam and Julian Tibble. abc: an extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented*

	AJVM	AJC	ABC	BH
execution join points				
Trivial Method	2.181	2.191	2.187	2.176
Trivial Method and Trivial Advice	3.562	3.899	3.902	3.391
Fibonacci	28.751	28.707	29.945	29.681
Fibonacci and Trivial Advice	29.975	31.575	31.976	35.151
this(x)	11.45	12.683	12.082	17.149
args(x)	11.237	18.28	18.321	23.903
target(x)	11.547	12.185	12.706	16.834
perthis	6.981	44.761	34.51	8.542
inside cflow	23.279	21.883	1.08	21.769
outside cflow	22.876	19.879	1.031	14.844
call join points				
Trivial Method	2.093	2.195	2.195	2.198
Trivial Method and Trivial Advice	3.534	3.761	6.457	3.761
Fibonacci	29.743	28.433	28.534	28.452
Fibonacci and Trivial Advice	30.403	31.133	32.017	35.356
this(x)	11.521	12.703	15.688	17.203
args(x)	11.52	18.116	28.211	21.406
target(x)	11.423	12.024	15.467	17.108
perthis	6.989	62.284	66.183	8.58
inside cflow	23.347	43.801	1.222	19.062
outside cflow	22.906	16.572	0.877	18.191

Table 1: Running time of micro benchmark tests in seconds

software development, pages 87–98, New York, NY, USA, 2005. ACM Press.

[23] Pavel Avgustinov and Aske Simon Christensen and Laurie Hendren and Sascha Kuzins and Jennifer Lhoták and Ondřej Lhoták and Oege de Moor and Damien Sereni and Ganesh Sittampalam and Julian Tibble. Optimising AspectJ. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 117–128, New York, NY, USA, 2005. ACM Press.

[24] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-in-time aspects: efficient dynamic weaving for Java. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 100–109, New York, NY, USA, 2003. ACM Press.

A Distributed Dynamic Aspect Machine for Scientific Software Development

Chanwit Kaewkasi
Centre for Novel Computing
School of Computer Science
University of Manchester
ckaewkasi@cs.man.ac.uk

John R. Gurd
Centre for Novel Computing
School of Computer Science
University of Manchester
jgurd@cs.man.ac.uk

Abstract

This position paper proposes the use of an event-based dynamic AOP machine as an infrastructure for interactive development of high performance scientific software. Advice codes in the proposed approach are similar to mobile agents that execute on distributed computational nodes. The key ideas underlying this approach are multi-level separation of parallelisation concerns and event-driven dynamic join points. The primary aim of the research is to use the AOP paradigm to improve productivity for scientific software development; the dynamic AOP machine is also expected to be further developed as an interactive computational grid.

Categories and Subject Descriptors D.3.4 [*Processors*]: Run-time environments

General Terms Run-time environments, Languages

Keywords Aspect-oriented Programming, Virtual Machine, High Performance Computing, Separation of Concerns

1. Introduction

Scientific software development often involves High Performance Computing (HPC) because many mathematical problems require considerable processing power. But performance is not the only requirement; productivity is becoming ever more important as scientific problems become more complex. In [13], there is a discussion of the characteristics of the High Productivity Computer System (HPCS) programme, under which it is sought to double productivity of Peta-flop systems every 18 months. The main target of the HPCS programme is to reduce the time-to-solution, rather than focusing only on hardware processing speed. The author suggests that the real cost for HPC codes consists of both the execution time and the development time. The execution time obviously relates to the performance of the executing code, including its parallelisation, while the development time may involve software engineering problems.

The work proposed in this paper focuses on applying an event-based dynamic aspect-oriented approach to reduce development time for HPC applications that run on distributed memory machines (DMMs). A DMM is a group of distributed computers that are connected together via a network in order to form a single computational resource. During the software development phase, interactivity is important for developers to gain productivity; fully compiling the whole program in every development cycle obviously impedes productivity.

This position paper proposes the concept and structure of a distributed dynamic aspect machine for scientific software to support interactive HPC software development with separation of concerns. The contributions of the proposed work are expected to be as follows. First, the use of an event-based dynamic approach [20] to aspect-oriented programming (AOP) [15] during the development phase may reduce the time to develop HPC programs. Second, the use of the proposed distributed advice code execution, which has the same characteristics as a mobile agent computing approach [6], may dramatically reduce communication overhead for data transfer, because the advice codes, in the form of mobile agents, will be mainly transferred among processors, rather than the processing data. It is also expected that the outcome of the proposed research, in the form of a distributed virtual machine, could be further developed to build an interactive computational grid for HPC software.

The remainder of the paper is organised as follows. Section 2 discusses related work dealing with parallelisation concerns and dynamic AOP, including the event-based approach. Section 3 introduces the concept of parallelisation sub-concerns, in particular Local Master Parallelisation and Remote Worker Parallelisation. Section 4 proposes the novel dynamic aspect machine for HPC applications. The paper ends in Section 5 with conclusions and ideas for future work.

2. Related Work

Two separate topics are pertinent, namely parallelisation concerns (and their limitations) and dynamic aspect-oriented programming.

2.1 Parallelisation Concerns

The proposed research follows on from the work of Harbulot and Gurd [7, 8], who have suggested that development of aspects for separating parallelisation concerns in HPC requires a new kind of join point model. They propose LoopsAJ, a join point model for loops, which has been implemented in `abc` [1] and is compatible with AspectJ's join point models. In their work, an analysis technique has been developed for finding loops in Java .class files. This technique detects back-edges to find appropriate weaving points for common kinds of advice, namely *before*, *after* and *around*.

The advantages of this technique are that it can be used for analysis of compiled Java programs and it is compiler-independent. However, in practice, parallelisation aspects actually occur at *application-level*. Application-level aspects are defined as a set of aspects that are specific for their base code. This kind of aspect is similar to a hyper slice, as described in [19]. Unlike infrastructure-level aspects, such as logging, transactional behaviour or security, parallelisation aspects in the context of the proposed research are *strategies* for the algorithms that they are woven into. Characteristics of parallelism over loops in other programming models, such as OpenMP [3] and HPF [17], suggest that it is necessary to know the semantics of the variables that are accessed in each loop in order to convert the loop into the proper parallel form. If there is insufficient information about these variables, it may be difficult for the compiler to control the parallel blocks efficiently.

The authors themselves identify several problems that can limit their approach for processing loops [8]. First, a whole method analysis may be required in order to identify data dependencies among variables in loops to ensure that the loop can be made parallel; such analysis is obviously complex and time-consuming. A possible solution is to develop a compiler that is able to embed additional structural information (e.g. block markers), that can be retrieved via reflection or similar techniques, into the compiled program for further processing by the machine; this is the approach that will be taken in the proposed research.

Second, loops in scientific programs often involve a large number of local variables. The nature of block statements is different from that of methods, which often have defined fixed arguments; block statements, such as loops, have no mandatory list of arguments. The approach introduced in [8], that manages some local variables as the loop's arguments, may not be flexible enough for complex HPC programs. For example, Figure 1 shows a parallel code written using OpenMP directives.[1] One can see that three array variables are being used in the parallel loop; in a more complex HPC code, a much larger number of local variables might be expected. Rather than passing these as arguments to the loop, the contextual programming style introduced in [16] is more appropriate for accessing them. Contextual programming utilises Java 5 annotations by assigning values to annotated variables at runtime. This technique could help the advice code to be more readable because it is not necessary to declare the potentially large number of local variables in the argument list of the join point context.

Third, bytecode transformation may change the original semantics of a loop. This problem arises because the weaving process makes direct modifications to the loop block. The loop modification techniques that are used in [8] are, for example, moving invariants out of the loop (in order to expose them in the join point context) and insertion of advice codes in several places within the loop. It is difficult to prove that the woven code has the same loop semantics as the original. The technique proposed here to solve this problem is event-driven dynamic weaving [20], which is described below and will be implemented in the proposed work. It is not necessary for this kind of weaving to modify the base code.

```
#include <omp.h>
#define N 1000

main () {
 int i;
 float a[N], b[N], c[N];

#pragma omp parallel shared(a,b,c) private(i)
 {
 #pragma omp sections nowait
  {

   #pragma omp section
   for (i=0; i < N/2; i++)
    c[i] = a[i] + b[i];

   #pragma omp section
   for (i=N/2; i < N; i++)
    c[i] = a[i] + b[i];
  }  /* end of sections */
 }  /* end of parallel section */
}
```

Figure 1. A simple OpenMP program.

Characteristic / Where	Weaving	
	Static	Dynamic
Compilation	Yes	No
Post-Compilation	Yes	No
Loading	Yes	No
Execution	No	Yes
Techniques	(1),(2)	(3),(4)
Redeployment	hard	easy

(1) source transformation
(2) bytecode transformation
(3) runtime bytecode insertion
(4) dynamic join point dispatch

Figure 2. Summary of characteristics of different weaving processes.

2.2 Dynamic Aspect-Oriented Programming

An AOP system uses a weaving process to compose separated concerns into the base code. Unfortunately, a static and load-time weaving process could be a time-consuming task, as it is in AspectJ. This weaving time problem is raised by Bockisch et al. [2], who introduce an approach, called envelope-based weaving, to make static weaving faster by using *envelopes* to reduce the number of potential join point shadows. However, the dynamic aspect machine will tackle this problem using a different approach.

Dynamic weaving is a weaving process in AOP that allows the system to redefine aspects at runtime. Current implementations of dynamic weaving are, for example, PROSE [18] and Steamloom [10, 9]; both are AOP systems for the Java virtual machine (JVM) execution environment. There are several models for describing the behaviour of dynamic weaving [4]. The model which will be used in the proposed research is an event model. As explained in [20], events will be triggered at join points to invoke the weaver and execute advice codes that are matched by pointcut designators during the execution flow. PROSE [18] is known to support this join point execution model. This model allows straightforward implementation for block-level join points which are difficult to weave by a static weaving process [8]. Figure 2 gives a summary of the characteristics of different weaving processes.

An AOP system that employs the event model for dynamic weaving has advantages over a static weaving system. A key characteristic of a flexible AOP system is that it allows re-definition and re-weaving of aspects at any time. The event-based dynamic weaving model offers a dynamic join point based on triggering of events during the flow of program execution. This means that the weaver

[1] This fragment of code is from the OpenMP tutorial at http://www.llnl.gov/computing/tutorials/openMP.

will be invoked by the interpreter using this triggering mechanism, and then the associated advice will be executed to modify the execution of the main program. In an implementation of this event-based model, the *static join point shadow* [11], a widely used concept in other weaving implementations [5, 12], is not needed because no transformations will be applied to the base code.

The machine proposed in this paper extends the event-based join point approach to encompass distributed computing. Events will trigger the execution of advice code on both local and remote machines, as will be described in Section 4.

3. Parallelisation Sub-concerns

Separation of concerns is a key idea to manage software complexity in software engineering. It forces programmers to deal with the software development concern-by-concern [19]. For example, in scientific computing, one typically starts by developing a mathematical model, next implementing the model as a computer algorithm, and then improving this to become a parallel code. Such development steps follow the concept of separation of concerns as they tackle the three concerns (mathematics, discretisation and parallelisation) one at-a-time.

Using parallelisation concerns that have been explicitly defined and separated out of the program as aspects is a good starting point for applying AOP to scientific software development [7, 8]. However, parallelisation might be more specific than other crosscutting concerns, such as logging or transaction behaviour, depending on the base algorithms and the intended machine environment(s). In other words, parallelisation is a parallelism strategy for this base algorithm to gain the best performance on a (set of) specific machine(s).

Parallelisation concerns can be further separated into sub-concerns. We call this kind of separation *multi-level separation* because our studies show that some kinds of concern, such as parallel 'task farming', contain related sub-concerns inside. The following subsections define two such parallelisation sub-concerns, namely Local Master Parallelisation (LMP) and Remote Worker Parallelisation (RWP).

3.1 Local Master Parallelisation

Local Master Parallelisation (LMP) is a sub-concern which exists only on a master machine in a DMM. The primary tasks of this sub-concern are as follows. Firstly, the code in LMP may pre-process data for worker machines. The pre-processing steps in LMP are, for example, dividing a large array to fit the number of processors, or initialising data or variables for remote machines. Secondly, the advice in LMP is responsible for distributing data to other machines. However, the mechanisms for distribution depend on the communication layers. For example, the advice may use an explicit API for sending data to the workers. Thirdly, the advice of LMP might call `proceed` to perform computing, or `parproceed` (parallel proceed) to perform the same task remotely. The definition of `proceed` and its parallel counterparts will be investigated later. Next, the advice receives or collects the computation results from remote machines. These steps also depend on the communication layer. Finally, post-processing steps might be performed on the received results; for example, all sub-results may be reduced or merged into a single array.

3.2 Remote Worker Parallelisation

Remote Worker Parallelisation (RWP) is a sub-concern that exists on worker machines. RWP is usually responsible for performing numerical computing tasks. An important characteristic of RWP is that it can enable nested parallelism, which allows another level of parallel computing inside the parallel code. For example, the

```
function find2nd(a) {
  x = sort(a)
  return x[2]
}
```

```
aspect find2ndParallel {
  sort@local(a) {
    aa[] = split(a, node)
    x = []
    aa.each { it ->
      x  += parproceed(it)
    }
    return proceed(x)
  }
  sort@remote(cond:{cpu==1}, aa) {
    return proceed(aa)[1..2]
  }
  sort@remote(cond:{cpu > 1}, aa) {
    aaa[] = split(aa, cpu)
    x = []
    aaa.forkEach { it ->
      r = proceed(it)
      x += r[1..2];
    }
    return proceed(x)[1..2]
  }
}
```

Figure 3. A sample algorithm and its aspect.

second `sort@remote` in Figure 3 is for performing nested parallel computation. In a heterogeneous HPC environment, some machines have a single processor, while other machines may contain multiple processors or multi-core processors. This means that they need different RWPs to achieve their best performance. Without AOP, multiple RWP concerns might make the algorithm difficult to read. More importantly, RWP can be the right solution for scalable heterogeneous DMMs in that the new RWP code can be easier to develop for future computing nodes.

Further, separating parallelisation into these two sub-concerns provides better modularity and it becomes easier to execute their associated advice codes with different processors. A pseudo code for an algorithm and its parallelisation aspect, which wraps LMP (`sort@local`) and RWPs (`sort@remote`) together, is shown in Figure 3.

4. Dynamic Aspect Machine

The concept of a dynamic aspect machine is introduced next. A dynamic aspect machine is a virtual machine that contains special steps in its fetch-execute cycle for executing programs that use dynamic aspects. Normally, a virtual machine executes a program by fetching a 'current' instruction from the program's image, pointing the program counter at the 'next' instruction and then executing the 'current' instruction. To work with this normal machine, static weaving mechanisms have traditionally been employed to insert advice codes into the base program [14, 12, 18, 10]. However, it is not necessary for advice code to be woven into the base code. The advice can instead be stored in a separated address space and, as long as it is able to execute correctly from there, a physical weaving process can be avoided. The associated execution mechanism might use an external referencing table to jump between the address spaces of the base program and its advice codes when *events* occur. This kind of execution model for AOP has been described in

[20, 4]. However, a new kind of machine is needed to support this model by adding extra steps into the machine's execution cycle.

The main advantage of storing advice codes in a separate space is that it is not necessary to perform physical insertion of these advice codes into the base program. So, several steps of bytecode transformation, which are significant parts of the static weaving process, can be eliminated. The advice code references can be kept as an external table, which, for example, store offsets of join points in the base program together with their associated advice addresses. Context information for the advice is assumed to be available on the operand stack.

It is our view that high productivity, interactive development of complex systems, such as HPC codes, requires the separation of concerns techniques of AOP together with an event-based dynamic weaving model. This model has a number of important characteristics. First, it is a general join point model for sequences of instructions; its event-driven nature can be applied to all kinds of statement, including loops and conditions. Second, the base code does not need to be transformed. This technique solves the problem that bytecode insertion might change the original semantics of the base code, especially for fine-grained statements. Third, this model fully supports separate execution of the advice code on one or more different machines.

4.1 Sequential Behaviour

The dynamic aspect machine operates by means of interception at a block join point. The definition of block in this context is similar to that of Java's synchronisation block; interception occurs at the entry and exit points to and from the block. The block join points are defined using entries in the *transparent join point table*. This model differs from Java's synchronisation block in that our approach does not need specific 'marker' instructions (e.g. JVM's `MORNITOR_ENTER` and `MORNITOR_EXIT`) in the base program; this implies that no change will be made to the program. The main purpose of the block join point model is to generalise the LoopsAJ join point model for loops [8] by adapting it to be event-driven.

Figure 4 shows how the block join point works with *around* advice code. The two black bars indicate that an event will be triggered at these points when (a) entering and (b) exiting the current block. At (1), the virtual machine fetches an advice instruction whose offset has been stored in the transparent join point table. This triggers the start of the dynamic weaving process. The machine pushes the current program counter (PC) onto the call stack. The before code is then executed using the same evaluation context as the base code. At (2), the `proceed` instruction has been reached. The current advice is stored onto the advice stack. The machine then switches back to the base code; the base code's PC is popped from the call stack and the program continues. If there is a jump to outside the block's region (PC $<$ block entry or PC $>$ maximum block exit), the virtual machine raises a runtime error. At (3), the machine reaches the instruction at the block exit and the PC is again pushed onto the call stack. The advice is popped from the advice stack and the machine then switches again to perform the advice code in the after region, starting from the advice's PC. Finally, at (4), the machine reaches the end of the after region and then returns to the base program at the end of the block.

4.2 Distributed Execution

As mentioned earlier, the advice can be executed in a distributed fashion. A distributed dynamic aspect machine consists of multiple advice execution units (AEUs) that are connected together using their own protocols. Each AEU is a processing unit which is responsible for executing advice codes that are triggered by dynamic join point events. The master machine contains a local AEU (LAEU), while worker machines contain remote AEUs (RAEUs).

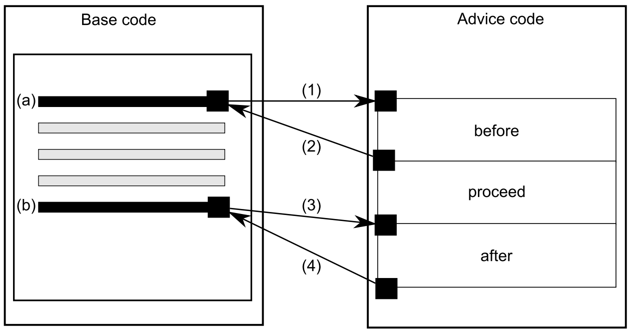


Figure 4. Behaviour of block join points.

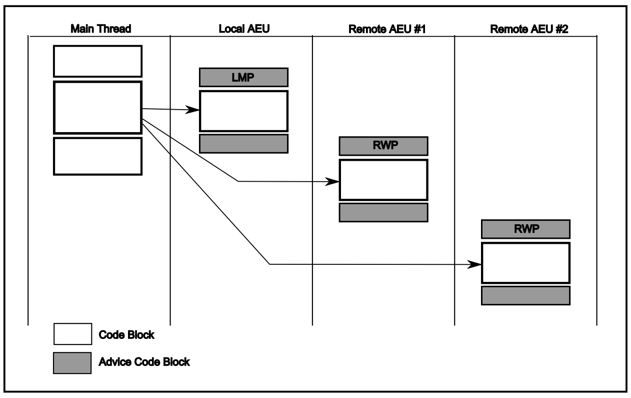


Figure 5. Sequence diagram showing the execution of a program in the distributed dynamic aspect machine.

An RAEU is a special case of an AEU which entails network communication. An RAEU receives the remote advice blocks, then it will be waiting for program fragments to-be-executed and associated data from the master machine.

For distributed execution, the basic execution steps from Section 4.1 need to be extended. The key actors are (1) the main thread, for executing the base program or algorithm, (2) the LAEU, for executing the advice codes for the LMP concerns, and (3) the RAEUs, for executing the advice codes for the RWP concerns. Figure 5 shows the execution steps in the proposed distributed machine. When the base program reaches a join point, it triggers the execution of all AEUs. At this point, the associated advice codes for each concern will be transferred to the appropriate AEUs. For example, from the pseudo code shown in Figure 3, the advice code `sort@local` will be sent to the LAEU, while the advice codes `sort@remote` will be sent to different RAEUs, according to their `cpu` pointcuts. After that, the appropriate base code will be sent to each AEU for use when the `proceed` instruction is encountered. Computational results can be returned from the RAEUs to be collected by the LAEU for a post-processing step. The final result will be returned to the main thread, which then continues by executing the next instruction.

Advice codes that are executed by RAEUs share some of the properties of mobile agents [6]. From the mobile agent point-of-view, each advice code will be serialized from the master machine in the form of a streaming JVM .class file. After receiving this advice code, an RAEU will instantiate and run it. When its task for this node finishes, the RAEU will serialize the advice code again with its current state and further migrate it to another AEU until the whole task is completed.

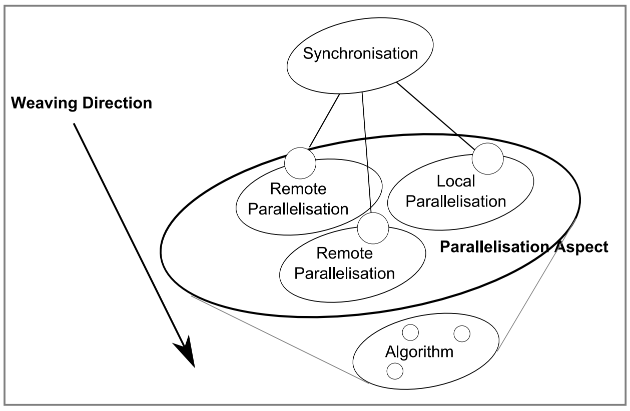


Figure 6. A possible way to further separate synchronisation concerns.

5. Conclusions and Future Work

This paper proposes a preliminary architecture for a distributed dynamic aspect machine that can aid high productivity, interactive, scientific software development. The machine has been specifically designed to cope with multi-level parallelisation concerns. The proposed machine is similar to the mobile agent computing model and is implemented using AEUs. Using this approach, it would also be possible to develop an interactive computational grid based on the proposed distributed dynamic aspect machine.

Figure 6 shows one possible way for further separating synchronisation concerns from an existing parallelisation aspect. After separating parallelisation concerns from the algorithm, one can see that synchronisation still cuts across a number of parallelisation concerns. An open question for multi-level separation of concerns in HPC and scientific software is whether synchronisation concerns should be separated out of the LMP and RWP sub-concerns. If yes, how should they be properly described? Is it possible to have another kind of structure to better describe synchronisation?

References

[1] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhotak, Ondrej Lhotak, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM Press.

[2] Christoph Bockisch, Michael Haupt, Mira Mezini, and Ralf Mitschke. Envelope-based weaving for faster aspect compilers. In *NODe/GSEM*, pages 3–18, 2005.

[3] Rohit Chandra, Dave Kohr, Leonardo Dagum, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000.

[4] C. Dutchyn, G. Kiczales, and H. Masuhara. Aspect Sand Box. http://www.cs.ubc.ca/labs/spl/projects/asb.html, 2002.

[5] Eclipse.org. AspectJ project. http://www.eclipse.org/aspectj, 2006.

[6] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Trans. Softw. Eng.*, 24(5):342–361, 1998.

[7] Bruno Harbulot and John R. Gurd. Using AspectJ to separate concerns in parallel scientific Java code. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 122–131, New York, NY, USA, 2004. ACM Press.

[8] Bruno Harbulot and John R. Gurd. A join point for loops in AspectJ. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, 2006.

[9] M. Haupt. *Virtual Machine Support for Aspect-Oriented Programming Languages*. PhD thesis, Software Technology Group, Darmstadt University of Technology, 2006.

[10] M. Haupt, M. Mezini, C. Bockisch, T. Dinkelaker, M. Eichberg, and M. Krebs. An Execution Layer for Aspect-Oriented Programming Languages. In J. Vitek, editor, *Proceedings of the First International Conference on Virtual Execution Environments (VEE'05)*, pages 142–152, Chicago, USA, June 2005. ACM Press.

[11] Erik Hilsdale and Jim Hugunin. Advice weaving in aspectj. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, New York, NY, USA, 2004. ACM Press.

[12] JBoss.org. JBoss AOP. http://labs.jboss.org/portal/jbossaop, 2006.

[13] J. Kepner. HPC productivity: An overarching view. *International Journal of High Performance Computing Applications*, 18(4):393–398, 2004.

[14] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science, ECOOP 2001 - Object-Oriented Programming: 15th European Conference*, 2072:327, June 2001.

[15] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[16] Gavin King. JSR 299: Web Beans. http://www.jcp.org/en/jsr/detail?id=299, June 2006.

[17] David B. Loveman. High performance fortran. *IEEE Parallel & Distributed Technology: Systems & Technology*, 1(1):25–42, February 1993.

[18] Angela Nicoara and Gustavo Alonso. Dynamic AOP with PROSE. In *Proceedings of International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA 2005) in conjunction with the 17th Conference on Advanced Information Systems Engineering (CAISE 2005)*, 2005.

[19] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.

[20] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, 2004.

A Case for Explicit Join Point Models for Aspect-Oriented Intermediate Languages

Hridesh Rajan

Iowa State University

hridesh@iastate.edu

Abstract

Aspect-oriented languages mostly employ implicit language-defined join point models, where *well-defined* points in the program are called join points and declarative predicates are used to quantify them. The primary motivation for using an implicit join point model is obliviousness and ease of quantification. A design choice for aspect-oriented intermediate languages is to mirror the source language model. In this position paper, I argue that an explicit join point model is better suited at the intermediate language level and sketch a preliminary solution.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features - Control structures; D.3.4 [*Programming Languages*]: Processors-runtime environments

General Terms Languages

Keywords Aspect-oriented intermediate languages, explicit join point models, implicit join point models, Nu AO intermediate language

1. Introduction

Aspect-oriented (AO) languages based on static compilation models such as AspectJ [17], Eos [23], etc have shown the potential to provide significant modularity benefits. Support for aspect-orientation in virtual machines and intermediate languages open new avenues. A key benefit is to preserve separation of concerns beyond compilation [21], which in turn promises to simplify development processes such as incremental compilation, debugging, etc. More optimization opportunities also open up as shown by Bockisch *et al* in their recent work [4].

The design of AO intermediate languages has also received some attention recently. Abstractions provided by enhanced intermediate language designs promise to preserve the separation of concerns achieved by AO source languages at the object code level. For example, one such language design discussed in our previous work [21,22] on *Nu* extends existing intermediate languages to include two new AO invocation primitives. We demonstrated that the *Nu* AO intermediate language allowed design modularity to maintained even at the object code level. All common advising structures in prevalent aspect-oriented (AO) mechanisms [9, 15] could be modeled as simple combinations of these two invocation primitives.

The purpose of this position paper is to direct attention at another important aspect of the intermediate language design: the join point model. A point in the execution of a program is called a join point in the popular terminology of aspect-oriented languages. A method for selecting a subset of these join points is called quantification. The notion of quantifiable join points [10, 12] is central to the notion of aspect-orientation [9, 16]. The AspectJ programming guide [3], for example, defines a join point as a new concept and explains that it is a *well-defined* point in the execution of the program. Others often define a join point as an implicit *language-defined* point in the execution of the program. I argue that an explicit join point model is more suitable for an AO intermediate language design, instead of an implicit language-defined model.

2. Rationale for a Language-Defined Implicit Join Point Model

The primary rationale for a language-defined join point model is *obliviousness* [10,11]. Obliviousness is a widely accepted tenet for aspect-oriented software development. In an oblivious AOSD process, the designers and developers of base code need not be aware of, anticipate or design code to be advised by aspects. This criterion, although attractive, has been questioned by others for various reasons and there is at least some consensus among researchers that complete obliviousness between base and aspect designers and developers may be a mirage [2, 5, 6, 8, 13, 18, 25]. Tools such as AspectJ Development Tools (AJDT) alleviate the problem [1] but do not completely solve it. Nevertheless, the notion of obliviousness appears to have significant influence on the design of aspect-oriented languages. A language-defined implicit join point model promotes obliviousness in that it allows aspect developers to quantify join points without requiring the base code developers to declare them.

The implicit language-defined join point model wins hands down with respect to the ease of the first time implementation. It is definitely much easier compared to manual join point selection (e.g. by placing annotation on join points) for the programmer to select join points for advising. By just writing simple declarative expression, they can select join points throughout the code base. In the next section, I discuss the design rationale for employing an explicit join point model in the intermediate language design.

3. Rationale for Explicit Join Point Models

In this section, I discuss the rationale for an explicit join point model in AO intermediate languages. In particular, I describe two important goals: extensibility, and the need for a mechanism to make reflective information available at a join point.

3.1 Extensibility

Virtual machines and intermediate language designs are often subject to standardization, which makes it extremely hard to change them. On the other hand, language designs often evolve to incorporate experimental ideas and constructs. For example, new join points are proposed to address use cases that the traditional join point model was not able to address [14, 24]. Other use cases are also documented, where desirable join points were not quantifiable in the current models: for example, in the context of the AO design of the Hypercast system, Sullivan *et al* [25] observed that *many join points that have to be advised in the same way cannot be captured by a quantified PCD, e.g., using wild-card notations. A separate PCD is required for each join point. There were about 180 places in the base code where logging was required. Most of the join points do not follow a common pattern. Not only is there a lack of meaningful naming conventions across the set of join points, but also variation in syntax: method calls, field setting, etc.* [25, pp. 170] These use cases will serve to fuel the evolution of the join point model at the source language level.

Adopting an implicit language model at the intermediate language level will restrict the design space of aspect-oriented source languages. Either the language designer will have to wait for the intermediate language designs to evolve to support new join points or they will emulate them using existing models thereby sacrificing the benefits of deeper support at the virtual machine and intermediate language level. Therefore, a key goal for an intermediate language design is extensibility. An explicit join point model, where join points in the intermediate code are precisely marked by the compiler, is likely to be more extensible compared to an implicit model. Such an implicit model will also need a precisely defined semantics and enforcement mechanism to rule out locations in the object code that may not be marked as join points.

3.2 Uniform Reflective Information

Current aspect languages provide an interface for accessing contextual (or reflective) information about a join point. An aspect can access the contextual information at the join point using pointcuts such as *this* to access the executing object (*this*), *target* to access the target object (such as the *target* of a call), *args* to access the arguments at a join point, etc. Alternatively, one can explicitly marshal this information from an *implicit* argument, often called *thisJoinPoint*, available to the advice, where other miscellaneous information such as source code location, name, etc, is also available.

This interface between the join point and the aspects is fixed in current AspectJ-like languages. There are rational reasons for such design decisions. This interface introduces coupling between the classes and the aspects. The thinner this interface is the lower the coupling will be, resulting in perhaps easier and independent evolution of classes and aspects. Extending the set of language constructs to include access to more primitives also takes away regularity from the language design [20]. As it is, current language constructs for retrieving contextual information are not completely regular, e.g. this, target, and arguments are not available at all join points [3].

In addition, others have shown that this rather limited interface does not satisfy all usage scenarios. For example, in some cases access to a local variable is needed [25, pp. 170] in others access to other information such as join point specific messages for logging is needed at the join points. Therefore, the source language design may evolve to include additional reflective information. It is therefore imperative that a more flexible method to access contextual information at the join point is provided at the intermediate language level that can support these evolutions.

```
ExplicitJP
   : .joinpoint modifier type
     identifier([arguments])block
block
   : {[instruction_list]}
```

Figure 1. Abstract Syntax of Explicit Join Points

```
class Point: FigureElement {
 ...
 public void SetX(int x) {
      if(this.x != x){
           this.x = x;
        }
     }
 }
```

Figure 2. An Example Code Snippet

```
1  .method public hidebysig instance
2      void SetX(int32 x) cil managed
3  {
4    // Code size       17 (0x11)
5    .maxstack  2
6    .joinpoint public void ExecutionSetX(int32 x)
7     {
8     IL_0000:  ldarg.0
9     IL_0001:  ldfld      int32 Point::x
10    IL_0006:  ldarg.1
11    IL_0007:  beq.s      IL_0010
12    IL_0009:  ldarg.0
13    IL_000a:  ldarg.1
14    IL_000b:  stfld      int32 Point::x
15    IL_0010:  ret
16    } // end of join point execution(public void Point.SetX(int x))
17 } // end of method Point::SetX
```

Figure 3. An Explicitly Declared Execution Join Point

4. An Explicit Join Point Model for Intermediate Languages

The proposed intermediate language design has two key characteristics that serve to satisfy the goals set in the previous section. First, it explicitly labels sections in the intermediate code that correspond to the join point shadows, and second, it explicitly defines the types of reflective information exposed at the join point. The view is similar to that of Ligatti *et al* [19] and Clifton and Leavens [7] in their semantics but has not appeared in language designs. Figure 1 shows the abstract syntax.

These labels will be generated by the compilers. To model language-defined implicit join points, the compiler would generate appropriate labels at all necessary locations (join point shadows) defined by the language semantics. For example, to model an `execution` join point shadow in AspectJ-like languages, the matched method code will be labeled as shown in Figure 3. The figure shows the intermediate code in Common Intermediate Language (CIL) for the source code shown in Figure 2. Here the intermediate code for the method `SetX` of class `Point` is labeled as the join point `ExecutionSetX` on line 6. Based on the reflective information being used in the advice, join point only exposes the value of the argument. It may also choose to expose reflective information as in AspectJ-like languages. The scope of the join point is identified by the block that encompasses instructions from line 8 to line 15.

Note that these labels are not visible to the programmer; therefore the source language-level obliviousness is still maintained. Moreover, explicit join point shadow makes the AO intermediate

```
.method public hidebysig instance
     void SetX(int32 x) cil managed
{
  // Code size       17 (0x11)
  .maxstack  2
   IL_0000:  ldarg.0
   IL_0001:  ldfld      int32 Point::x
   IL_0006:  ldarg.1
   IL_0007:  beq.s      IL_0010
  .joinpoint public void IfBlockInsideSetX(int32 x)
   {
     IL_0009:  ldarg.0
     IL_000a:  ldarg.1
     IL_000b:  stfld      int32 Point::x
   } // end of join point IfBlockInsideSetX
   IL_0010:  ret
} // end of method Point::SetX
```

Figure 4. Supporting Finer-grained Join Points

language extensible in that it may now support source languages with different join point models.

To demonstrate the extensibility of this AO intermediate language model, let us now consider an evolutionary scenario, where the join point model of the source language is enhanced to include conditional constructs (if, switch) as join points. Using this enhanced model, the aspect developer chooses to select the execution of the true block (line 5 in Figure 2) of the *if* statement inside the method `SetX`. This statement truly represents the state change of the Point class. The compiler for this enhanced language model may now generate the intermediate code as shown in Figure 4. In this modified version, only the intermediate code corresponding to line 5 in Figure 2 is within the scope of the new join point `IfBlockInsideSetX`.

5. Conclusion

In this position paper, I argued that explicitly declared join points are better suited for intermediate languages to support extensibility in source languages in two dimensions. First key dimension is evolution of join point models of source languages. Second dimension is extension of the reflective information that is available at the join point. A preliminary solution was proposed with the expectation that it will serve to generate exciting discussion during the workshop.

Acknowledgements

The author is supported in part by National Science Foundation grant CT-ISG 0627354 and by a generous startup grant provided by Iowa State University. This paper draws upon the discussions with my colleague Gary T. Leavens and my students Robert Dyer and Juri Memmert. Many thanks to the anonymous reviewers of the AOSD VMIL 2007 reviewers for helpful comments.

References

[1] AJDT:AspectJ development tools. http://www.eclipse.org/ajdt/.

[2] Jonathan Aldrich. Open modules: Modular reasoning about advice. In *Proc. 2005 European Conf. Object-Oriented Programming (ECOOP 05)*, pages 144–168, July 2005.

[3] AspectJ programming guide. http://www.eclipse.org/aspectj/.

[4] Christoph Bockisch, Sebastian Kanthak, Michael Haupt, Matthew Arnold, and Mira Mezini. Efficient control flow quantification. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, 2006.

[5] Curtis Clifton and Gary T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. Technical Report TR03-01a, Iowa State University, January 2003.

[6] Curtis Clifton and Gary T. Leavens. A design discipline and language features for modular reasoning in aspect-oriented programs. Technical Report 05-23, Department of Computer Science, Iowa State University, December 2005.

[7] Curtis Clifton and Gary T. Leavens. MiniMAO$_1$: Investigating the semantics of proceed. *Science of Computer Programming*, 2006.

[8] C. Constantinides and T. Skotiniotis. Reasoning about a classification of cross-cutting concerns in object-oriented systems. In Pascal Costanza, Günter Kniesel, Katharina Mehner, Elke Pulvermüller, and Andreas Speck, editors, *Second Workshop on Aspect-Oriented Software Development of the German Information Society*. Institut für Informatik III, Universität Bonn, February 2002. Technical report IAI-TR-2002-1.

[9] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001.

[10] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*, October 2000.

[11] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Aspect-oriented Software Development*, pages 21–35. Addison-Wesley Professional, 2004.

[12] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. pages 21–35. Addison-Wesley, Boston, 2005.

[13] William G. Griswold, Kevin J. Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular software design with crosscutting interfaces. *IEEE Software, Special Issue on Aspect-Oriented Programming*, Jan/Feb 2006.

[14] Bruno Harbulot and John R. Gurd. A join point for loops in aspectj. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 63–74, New York, NY, USA, 2006. ACM Press.

[15] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming 15th European Conference*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, Budapest, Hungary, June 2001.

[16] Gregor Kiczales, John Lamping, Cristina Videira Lopes, Chris Maeda, Anurag Mendhekar, and Gail Murphy. Open implementation design guidelines. In *Proceedings of the 19th International Conference on Software Engineering*, pages 481–90, Boston, Massachusetts, 17–23 May 1997. IEEE.

[17] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland, June 1997. Springer-Verlag.

[18] Gregor Kiczales and Mira Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, volume 3586 of *Lecture Notes in Computer Science*, pages 195–213. Springer, 2005.

[19] Jay Ligatti, David Walker, and Steve Zdancewic. A type-theoretic interpretation of pointcuts and advice. *Science of Computer Programming*, Winter 2005/2006.

[20] Bruce J. MacLennan. *Principles of programming languages: design, evaluation, and implementation (2nd ed.)*. Holt, Rinehart & Winston, Austin, TX, USA, 1986.

[21] Hridesh Rajan, Robert Dyer, Youssef Hanna, and Harish Narayanappa. Preserving separation of concerns through compilation. In Lodewijk Bergmans, Johan Brichau, and Erik Ernst, editors, *In Software Engineering Properties of Languages and Aspect Technologies (SPLAT 06), A workshop affiliated with AOSD 2006*, March 2006.

[22] Hridesh Rajan, Robert Dyer, Harish Narayanappa, and Youssef Hanna. Nu: Towards an aspect-oriented invocation mechanism. Technical Report 414, Iowa State University, Department of Computer Science, Mar 2006.

[23] Hridesh Rajan and Kevin J. Sullivan. Eos: instance-level aspects for integrated system design. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 297–306, New York, NY, USA, September 2003. ACM Press.

[24] Hridesh Rajan and Kevin J. Sullivan. Aspect language features for concern coverage profiling. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 181–191, New York, NY, USA, 2005. ACM Press.

[25] Kevin J. Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information hiding interfaces for aspect-oriented design. In *The Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)*, pages 166–175, Sept 2005.

A Direction for Research on Virtual Machine Support for Concern Composition

Harold Ossher
IBM T. J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10590, USA
+1-914-784-7975
ossher@us.ibm.com

ABSTRACT

This position paper suggests research directions in the area of virtual machines supporting aspect-oriented capabilities in the context of object-oriented languages.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors - *Run-time environments.*

General Terms

Languages.

Keywords

Aspect-oriented software development, separation of concerns, dynamic software composition and decomposition.

1. INTRODUCTION

This brief position paper suggests research directions I consider important in the area of virtual machines (and perhaps intermediate languages) supporting aspect-oriented capabilities in the context of object-oriented languages. Rather than a detailed model or solution, I'll suggest a broad class of runtime models that I believe would be fruitful to explore and to use as a basis for innovative virtual machine concepts.

Section 2 discusses some key requirements, and section 3 discusses the proposed class of runtime models.

2. REQUIREMENTS

The fundamental problem to be addressed is the provision of suitable primitives for adding or removing concerns. One example is primitives for binding and unbinding or enabling and disabling advice. This is valuable, but not the whole story. In general, primitives supporting concern composition and decomposition are the goal.

I will consider here just two key requirements I believe such primitives should satisfy: *naturalness* and *safety*.

2.1 Naturalness

Object-orient practitioners are used to thinking about a *sea of objects* at runtime, sending messages to one another. This is very natural to them, and it models the real world effectively. It is much less natural to aspect-oriented practitioners, however, because it forces them to think about translating all aspect-oriented effects into object-oriented effects. Concerns are not first class, and operations on them, like composition and decomposition, are meta-operations, introducing an additional level of complexity. This shows through particularly unpleasantly during debugging. Developers are confronted with the woven runtime objects, often involving mangled names are glue code, rather than with runtime elements that match their source code, such as separate, clean objects and aspects.

I argue that one of the key elements of research into virtual machine support for concerns is exploration of good, natural runtime models. In section 3 I discuss a general class of runtime models I believe are especially promising.

2.2 Safety

The naturalness issue, as well as other considerations like those that apply to virtual machines and JITs in general, imply that VM support for composition can be preferable to pre-execution weaving even in the absence of dynamic AOSD, in which composition and decomposition can be initiated on demand at run time. The safety considerations described here apply primarily to the dynamic case, though even VM-based implementations of essentially static composition must beware of the pitfalls.

Addition or removal of a concern requires, in general, coordinated changes to many objects or classes. A key element of safety is therefore *atomicity*: when a concern is added or removed, all its elements must be added or removed as an (effectively) atomic operation. If it is interleaved with execution, various semantic problems can arise. For example, two pieces of advice in an aspect might be written to cooperate, where one computes a value and stores it in an aspect-local variable, and the other uses it. The pointcuts might be such that the producer advice always executes before the consumer. If the aspect is added in such a way that the consumer advice is added, then some execution is allowed to happen, then the producer is added, it is possible that the consumer advice will be triggered before the producer.

A related issue is ensuring that composition or decomposition do not disrupt the running program. For example, suppose that at some instant a method that has both before and after advice associated with it is executing. The before advice has thus already executed, and might expect the after advice to do some cleanup, perhaps as critical as committing a transaction. If the concern containing both pieces of advice is removed at that particular instant, what happens to execution of the after advice? If the advice is simply removed and not executed, semantic problems result, even if the removal is atomic.

These sorts of problems can be dealt with in a variety of ways. For example:

- Assuming that the primitives will be generated only by a compiler in a manner that is guaranteed to be safe.
- Some special support for currently-executing code, along the lines that JITs use to allow optimization of methods without disrupting current executions.
- Suitable interlocks within the VM.

Another element of safety has to do with proper handling of additional state introduced by a concern. This is essentially the same as the data migration problem in databases. When new state is introduced, it must be initialized, perhaps in a fixed way, perhaps based on the existing state. If a concern is removed and later reinstated, is it expected that the state values previously computed be retained?

This position paper does not provide solutions to these problems. My point here is that these issues are crucial, and need to be at least considered in all work on VM support for aspect-oriented technology.

3. SEA OF FRAGMENTS

Crosscutting concerns, by their very nature, affect many scattered elements in the dominant decomposition of the system. In the case of object-oriented systems, many objects and/or classes are affected when a new concern is added (or removed). The behavior of some of their methods is enhanced or overridden. Some additional state or methods might be added. This is generally implemented, and often defined, by *weaving*: transforming the underlying language constructs. Even if weaving is provided as a VM operation, this diminishes the first-class nature of concerns: though they may be first-class groupings, their elements get absorbed into the language's runtime representation and do not remain as first-class elements participating in execution.

A concern consists of a number of fragments, such as pieces of advice or members to be introduced. There is also a specification, within the concern or separate, of how the fragments are to be associated with objects, such as the pointcuts at which advice is to be applied, and the objects or classes into which new members are to be introduced. It would seem natural that, once the concern is added to a running system, its fragments remain as elements of the runtime representation. This is equally true of approaches like Hyper/J [7], which explicitly has a fragment-composition model, and approaches like AspectJ [5], which does not: each AspectJ aspect is a fragment consisting, in turn, of advice and intertype declaration fragments, and it is more natural to have the aspect and its sub-fragments as part of the runtime representation than to have all or part of them "woven away."

This suggests a runtime model consisting of fragmented (or faceted) objects. In this sort of model, an object has a unique identity, but it is not usually a single chunk of memory with a single table of associated methods. Instead, it is a constellation of a number of fragments, each of which is a single chunk of memory with a single table of associated methods (modulo inheritance from or delegation to other objects), but not necessarily with its own identity. These fragments combine to determine the behavior of the object, delegating to one another according to the manner in which they are combined. Bill Harrison and I analyzed the various ways in which such fragments can interact in an earlier paper [1].

There are many ways to realize this sort of model. For example, one can think of it in terms of groups of co-operating objects rather than as objects as constellations of fragments, as we did in that analysis [1]. It is along the lines of dynamic role models [6], Object Teams [2], and subjective objects [8]. My objective here is not to offer a model in detail, but to claim that exploration of such models is an especially fruitful direction for research in virtual machine support for aspect-oriented languages.

How natural are these models in fact? It is hard to beat the simplicity and real-world evocativeness of object models. I believe fragmented-object models come close, however, because they mirror the real-world situation of objects acquiring or losing characteristics (roles) during their lifetimes. When a single person marries, s/he acquires some new behaviors and characteristics of married people, while still remaining the same person. If s/he divorces, s/he loses these. Everyone understands this in the real world. The goal is to come up with ways of surfacing the concepts of fragmented-object models to developers that mirror the natural situation as much as possible. This might not seem like an important issue for a virtual machine, and perhaps it is not in the guts, but it is important that the runtime model that developers need to understand and debug be simple and natural.

3.1 Operations

Fragments are thus associated with, or part of, objects. They are also associated with concerns. Indeed, a concern is a set of fragments, possibly along with instructions about how to compose those fragments into the system (or the instructions might be separate, a model I prefer). Another way to think of this is that the runtime model is a multi-dimensional sea or space of fragments, and concerns are hyperslices [10, 7]. The actual composition involves establishing the appropriate connections between fragments and objects. Removing a concern involves breaking such connections.

This leads to initial thoughts about VM operations at two levels: the individual object and fragment level, and the concern level. I postulate an approach in which the fragment-level operations in isolation are not safe, but they may only be used as part of larger concern-level operations, which are safe.

3.1.1 Fragment-level operations

Operations are needed to:

- Create a new object. This is really just creation of a unique identity. For the object to acquire state or behavior requires the association of fragments. Of course, most object creation is likely to be accompanied by immediate association of at least one fragment.
- Associate a fragment with an object, establishing one of a number of possible delegation relationships between them [1]. If the fragment contains state, there must be a

means to initialize it as part of this operation (perhaps just to *null*).

- Remove a fragment from an object, breaking the delegation relationships.

A fragment could be a single method (piece of advice) that is associated with an existing method of the object in such a way that the new fragment executes before (or after or around) the original method. Alternatively, the fragment could contain some state to be added to the object, and perhaps some methods to manipulate it. Addition or removal of fragments thus covers the important case of binding and unbinding advice, in addition to introduction of new state and behavior.

3.1.2 Concern-level Operations

At a minimum, operations are needed to:

- Add a concern into the running system
- Remove a concern from the running system

These operations are realized in terms of a number of fragment-level operations, and the VM must enable them to be performed atomically. It must also, as noted above, have a means of ensuring that execution is not corrupted: there must be an approach to handling the situation where execution is currently in progress within an existing fragment with which some new fragment becomes associated or dissociated.

To provide full-scale, first-class status for concerns, and to enable composition, additional operations would be valuable, such as:

- Compose concerns "on the side" so that a collection of concerns can be added or removed as a unit.
- Form a concern from a collection of fragments.
- Locate join points based on queries.

Exploring the set of appropriate operations is one of the research topics in this area.

3.2 Language model

It seems convenient, and perhaps essential, to describe the associations between fragments and objects in terms of delegation. There have been long-standing debates between delegation and inheritance, and although they have been shown to have a degree of formal equivalence [9], they do have different convenience characteristics. Classes and inheritance are often preferred because of the static typing they provide, and yet Ungar and Smith, in their landmark paper on Self, made powerful arguments for the power and simplicity of delegation [11]. Work on Self also showed that the performance penalty often attributed to flexible delegation approaches can be greatly reduced [4, 12, 3]. The introduction of fragments into the mix seems to me a force in the direction of delegation, and I think a sea-of-fragments model would be best defined and implemented in terms of delegation. This does not mean abandoning all the value of higher-level organizational and typing structures, such as classes and interfaces; I believe there is potential for realizing much or all of their benefit in layers above the underlying delegation layer.

4. CONCLUSION

This brief position paper raised the important requirements of naturalness and safety for virtual machine support for aspect-oriented capabilities, construed broadly. It also suggested a sea-of-fragments runtime model based on delegation as a promising approach to explore.

5. ACKNOWLEDGMENTS

Many thanks to Bill Harrison, Peri Tarr, David Ungar and Adrian Colyer for many enjoyable and insightful discussions over the years, from which emerged many of the insights expressed in this position paper. Thanks to the anonymous reviewers for some most helpful comments.

6. REFERENCES

[1] William Harrison and Harold Ossher, "Member-Group Relationships Among Objects." AOSD '02 Workshop on Foundations Of Aspect-Oriented Languages (FOAL).

[2] Stephan Herrmann, "Object Teams: Improving Modularity for Crosscutting Collaborations." Proc. of Net.ObjectDays, Erfurt, 2002.

[3] Urs Hölzle and Ole Agesen, "Dynamic vs. Static Optimization Techniques for Object-Oriented Languages." Theory and Practice of Object Systems 1(3), 1995.

[4] Urs Hölzle, Craig Chambers, and David Ungar, "Optimizing Dynamically-Typed Object-Oriented Programming Languages with Polymorphic Inline Caches." In *ECOOP `91 Conference Proceedings*, Geneva, Switzerland, July, 1991. Published as Springer Verlag LNCS 512, 1991.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, Jeffrey Palm and William G. Griswold. "An Overview of AspectJ." In *Proc. 15th European Conference on Object-Oriented Programming*, pp. 327-353, 2001.

[6] Bent Bruun Kristensen and Kasper Osterbye, "Roles: conceptual abstraction theory and practical language issues." Theory and Practice of Object Systems 2(3), 1996.

[7] Harold Ossher and Peri Tarr. "Using Multi-Dimensional Separation of Concerns to (Re)Shape Evolving Software." *CACM* 44(10): 43–50, October 2001.

[8] Randall B. Smith, David Ungar: A Simple and Unifying Approach to Subjective Objects. Theory and Practice of Object Systems 2(3), 1996.

[9] Lynn Andrea Stein, Henry Lieberman, David Ungar, "A Shared View of Sharing: The Treaty of Orlando." In Object-Oriented Concepts, Databases and Applications, ACM Press/Addison-Wesley, 1989.

[10] Peri Tarr, Harold Ossher, William Harrison, and S. M. Sutton, Jr., "N degrees of separation: Multi-dimensional separation of concerns." In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99),* 107–119, IEEE, May 1999.

[11] David Ungar and Randall B. Smith, "Self: The Power of Simplicity." In *OOPSLA '87 Conference Proceedings*, pp. 227-241, Orlando, FL, October, 1987.

[12] David Ungar, Randall B. Smith, Craig Chambers, and Urs Hölzle, "Object, Message, and Performance: How They Coexist in Self." Computer, 25(10), October, 1992, pp. 53-64.