Java Virtual Machine support for Aspect-Oriented Programming

Alexandre Vasseur, Joakim Dahlstedt, BEA Systems
avasseur@bea.com, jda@bea.com
in affiliation with Jonas Bonér, Terracotta Inc.
jonas@terracottatech.com

BEA Systems, Java Runtime Product Group
Folkungagatan 122, S-102 65 Stockholm, Sweden

ABSTRACT

The majority of the frameworks for Aspect-Oriented Programming (AOP) use bytecode weaving, in addition to that, bytecode instrumentation is becoming more and more popular in enterprise software in general, as a way of adding services to applications, more or less transparently.

Unfortunately, there are many problems with bytecode instrumentation, problems that these frameworks and products will inherit. The key questions are: up to which point can bytecode instrumentation based weaving techniques scale and achieve manageability, transparency and efficiency? Is there a risk that products that are relying on these techniques will reach an end-point that limits further innovation towards more efficiency, ease of use and dynamicity?

We believe Java Virtual Machine (JVM) level support for AOP will address these issues and provide a solid ground for further innovations in the field of Aspect-Oriented Software Development (AOSD) in Java.

This report will first discuss the different implementation techniques for weaving, followed by a discussion of the problems with bytecode instrumentation based weaving that we have today, as well as potential future problems. This includes problems like, inefficient instrumentation, double bookkeeping, increasing complexity, multiple agents, reflective join points, etc.

We will then propose a novel technology for supporting AOP directly in the JVM that we have implemented in the JRockit JVM. A technology that we believe will address the problems outlined above.

Keywords

Aspect-Oriented Programming, AOP, AOSD, weaving, Java Virtual Machine, JVM

1. INTRODUCTION

Aspect-Oriented Programming [1] (AOP) is gaining momentum in the software community and the enterprise space at large. Introduced by Parc back in the 90's, it has been getting more and more mature through several initiatives and innovations in the research community, the open source community and the enterprise in the last two years. There has been a lot of traction in the Java community that recently lead to the merger of AspectWerkz [3] and AspectJ [4] now housed at the Eclipse under the name AspectJ 5 [5]. AspectJ is sponsored by BEA Systems and IBM and can be considered as the de-facto standard for AOP in Java.

As the popularity of AOP is growing and the research community is moving things forward, vocabulary, concepts, and implementations have gained in consistency, allowing for better tool support and developer experience - such as with AspectJ Eclipse plug-in AspectJ Development Tools (AJDT) [6].

AOP has gone through several implementations techniques, ranging from source code weaving to bytecode instrumentation based weaving. Bytecode instrumentation is the technique that has been most widely adopted in Java, in particular after the advent of Java 5 JVMTI [7]. Bytecode instrumentation is now used by several enterprise products in the area of application management and monitoring and more recently Plain Old Java Object (POJO) based middleware [13] and transparent clustering [14].

Unfortunately, there are many problems with bytecode instrumentation, problems that these frameworks and products will inherit. The key questions are: up to which point can bytecode instrumentation based weaving techniques scale and achieve manageability, transparency and efficiency? Is there a risk that products that are relying on these techniques will reach an end-point that limits further innovation towards more efficiency, ease of use and dynamicity?

Up to which point can bytecode instrumentation based weaving techniques scale and achieve manageability, transparency and efficiency? Is there a risk that AOP implementations that are relying on these techniques will reach an end-point that limits further innovation towards more efficiency, ease of use and dynamicity?

We believe Java Virtual Machine (JVM) level support for AOP will address these issues and provide a solid ground for further innovations in the field of Aspect-Oriented Software Development (AOSD) in Java.

This report will first discuss the different implementation techniques for weaving, followed by a discussion of the problems with bytecode instrumentation based weaving that we have today, as well as potential future problems. This includes problems like, inefficient instrumentation, double bookkeeping, increasing complexity, multiple agents, reflective join points, etc.

We will then propose a novel technology for supporting AOP directly in the JVM that we have implemented in the JRockit JVM [8]. A technology that we believe will address to the problems outlined above.

2. CURRENT STATE OF WEAVING

Weaving is the process of taking cross-cutting code and the regular "base" application and "weave" them into one single unit, one single application.

Weaving can happen at different periods in time:

- Compile time weaving: post processing the code, e.g. ahead of deployment time (thus ahead of runtime) (as in AspectJ 1.x).
- Load time weaving: weaving is done as the classes are loaded i.e. at deployment time (as in AspectWerkz 2.0).
- Runtime weaving: weaving can occur at any time during the lifetime of the application (as in JRockit and the SteamLoom Project [12]).

This process can also be done in many different ways:

- Source code weaving: input is the developed source code and output is modified source code that invokes the aspects (as in AspectJ 1.0).
- Bytecode weaving: input is the compiled application classes' bytecode and output is modified bytecode of the woven application (as in AspectWerkz 2.0 and AspectJ 1.1 and beyond).

Source code weaving is limited in the sense that all source code must be available and presented to the weaver so that aspects can be applied. This makes it impossible for example to implement generic monitoring services - with or without use of explicit AOP constructs. Compile time weaving suffers from the same problem. All bytecode that will be deployed needs to be prepared before deployment in a post compile time.

Bytecode weaving vs. JVM weaving is the subject of this report and will be discussed in the following sections.

A side note is that a limited form of weaving has been available in the JVM for some time: dynamic proxies [15]. This API has been part of the JDK since 1.3 and it allows to create a dynamic proxy of an interface (or a set of interfaces) which gives the possibility of intercepting each invocation to the interface(s) declared methods and redirect it to an invocation handler implementing arbitrary logic. This is not really weaving by definition, but it resembles it, in the way that it gives a simple way of doing method interception. This technique is used by various frameworks to do simple form of AOP, for example, the Spring Framework [16].

3. PROBLEMS WITH BYTECODE INSTRUMENTATION BASED WEAVING

It is worth emphasizing that the problems described below are tied to bytecode instrumentation and as a consequence affects different AOP implementations (such as AspectJ). These problems have impact on all bytecode instrumentation based products in general, such as application monitoring solutions, profiling tools or other AOP-applied solutions as their use is getting more and more popular.

3.1 Instrumentation is inefficient

The actual instrumentation part of the weaving is usually very CPU intensive, and sometimes also consumes significant amounts of memory. This can affect startup time. For example, to intercept all calls to the `toString():String` method or all accesses to a certain field, one needs to parse almost every single bytecode instruction in all classes, one by one. This also means that a lot of intermediate representation structures will be created by the bytecode instrumentation framework to expose the bytecode instructions in a usable way. This could potentially mean that the weaver needs to parse all bytecode instructions in all classes in the whole application (including third-party libraries etc.), e.g. in the worst case more than 10,000 classes.

If more than one weaver is use, the overhead will be multiplied.

For some pointcut expressions, the underlying weaver can be optimized by first looking at the class' bytecode constant pool section to determine if there might be a match. Unfortunately this approach can not be used for most of the pointcut for which the complete class hierarchy and member database must be known in order to perform the matching – as discussed below.

3.2 Double bookkeeping: Building a class database for the weaver is expensive

In order to know whether a class/method/field should be weaved or not, the weaver needs to do matching on metadata for this class or member. Most AOP frameworks and AOP-applied products have some sort of high level expression language (pointcut expressions) to define where (at which join points) a code block (advice) should be weaved in. These expression languages can for example let you pick out all methods that have a return type which implements an interface of type T. This information is not available in the bytecode instructions representing the call to a specific method M. The only way of knowing if this specific method M should be weaved or not is to look it up in some sort of class database, query its return type and check if its return type implements the given interface T.

You might be thinking: why not just use the `java.lang.reflect` API? The problem with using reflection here is that there is no way of querying a Java type reflectively without triggering the class loading of this particular class, which will trigger the weaving of this class before we know enough about it in order to do the weaving (in load time weaving infrastructures). Simply put: we end up with the classic chicken and egg problem.

The weaver therefore needs a class database (usually built up in memory from the raw bytecode read from the disk) to do the required queries on if needed for the actual join points to be found. This problem can sometimes be avoided by limiting the expression language expressiveness, but that usually limits the usability of the product.

This in memory class database is also redundant once the weaving is done. The JVM already has all this information in its own database, well optimized, (that for example serves the `java.lang.reflect` API). So we end up doing double bookkeeping of the whole class structure (object model) which consumes significant and unnecessary memory, as well as adds a startup cost in creating this class database, and maintaining it when a change occurs.

If more than one weaver is use, the overhead will be multiplied – as in most cases each weaver maintains its own class database.

3.3 Changing bytecode at runtime adds more complexity

Java 5 brought the HotSwap API [9] as part of the JVMTI specification. Before Java 5, this API was only available when running in debug mode, and only for native C/C++ JVM extensions. It allows changing the bytecode, i.e. to redefine a class, at runtime. It is used by some AOP frameworks, and AOP-applied products to emulate runtime weaving capabilities.

Despite being very powerful, this API limits usability, scalability and is inefficient. Since bytecode is being changed at runtime, instrumentation costs (CPU overhead and memory overhead) are also happening at runtime. Also, if there is a need to do a change in many places, this means redefining many classes as well. The JVM will then have to redo all the optimization and inlinings that it may have done.

It is also very limited. The API does not specify where the current running bytecode can be retrieved. A weaver thus needs to make the assumption that this bytecode is on disk, or it needs to keep track of it. This is a major issue when multiple weavers are used as explained in the next section.

Further, none of the current implementations of the HotSwap API supports schema change which the specification states as being optional. This means that it is not possible to change the schema of a class at runtime, e.g. add methods/fields/interfaces etc that might be needed for the underlying instrumentation model. This makes it impossible to implement certain types of runtime weaving and thus requires the user to "prepare" the classes in advance. Such a technique is for example used in AspectWerkz[3] to provide hot-deployment of around advice.

3.4 Multiple agents is a problem

When multiple products are using bytecode instrumentation, unexpected problems may happen. Problems related to precedence, notification of changes, undoing of changes etc. This perhaps is not so much a problem today, but this will be a significant problem in the future. A weaver can be seen as an agent (as referred to in the JVMTI specification) that performs instrumentation at load time or runtime. When there are multiple agents, it is a high risk that the agents will get in each others way, changing the bytecode in a way that was not expected by the next agent, making the assumption that it is the sole configured agent.

Here is an example of a problem that can happen when two agents are unaware of each other. If for example an application uses two agents, one AOP weaver and one application performance product (that are both doing bytecode instrumentation at load time), there is a risk that the woven code may not be part of the performance measurement as illustrated below:

```
// Say this is the original user code
void businessMethod() {
    userCode.do();
}

// -- Case 1
// Say the AOP weaver was applied BEFORE the
// performance management weaver
// The weaved code will behave like:
void businessMethod() {
    try {
        performanceEnter();
        // hypothetical advice
        aopBeforeExecuting();
        userCode.do();
    } finally {
        performanceExit();
    }
}
// i.e. the AOP code will affect the measure

// -- Case 2
// Say the AOP weaver was applied AFTER the
// performance management weaver
// The weaved code will behave like:
void businessMethod() {
    //hypothetical advice
    aopBeforeExecuting();
    try {
        performanceEnter();
        userCode.do();
    } finally {
        performanceExit();
    }
}
// i.e. the AOP code will NOT affect the measure
```

This illustrates a problem with precedence between the agent: there is no fine grained configuration to control the ordering at a join point (or pointcut) level. The ordering is not well-defined.

Some other situations might lead to more unpredictable results. For example when a field access is intercepted, it usually means that the field get bytecode instructions are moved to a newly added method and replaced by a call to this new method. The next weaver will thus see a field access from another place in the code (from this newly added method) that then might not be matched by its own matching mechanism and configuration.

To summarize, the main problems are:

Which bytecode does the agent see? The problem is that normally the bytecode to be weaved is obtained from the class loading pipeline but the dependent bytecode to build up the class database from is read from disk. When multiple agents are involved bytecode on disk is not anymore the one being executed, since some agent might have already changed the bytecode. This means that the second agent has an incorrect view of the bytecode. This also happens when the HotSwap API is used.

3.5 Intercepting reflective calls is tedious

Current weaving approaches can only instrument execution flows that can be (at least partially) statically determined. Consider the following code sample that invokes the method `void doA()` on the given instance `foo`:

```
public void invokeA(Object foo) throws Throwable {
    Method mA = foo.getClass().getDeclaredMethod(
        "doA", new Class[0]
    );
    mA.invoke(foo, new Object[0]);
}
```

This kind of reflective access is often used in modern libraries, to create instances, to invoke methods, or to access fields.

From a bytecode perspective, the call to the method `void doA()` is not seen. The weaver will only see calls to java.lang.reflect API. There is no simple and performant way of weaving calls that are made reflectively. This is an important limitation in how weaving can be done and how AOP is implemented today. Best practices recommend the developer to use execution side pointcuts instead. Obviously, from a JVM perspective, there will be a method dispatch to the `doA()` method, even if it does not appear in the source code or bytecode. JVM weaving has proven to be to be the only weaving mechanism that addresses this issue in an efficient way.

3.6 Other problems

Bytecode instrumentation, especially when done on-the-fly (load time or runtime), is seen with skepticism by some people. There is an emotional angle to changing code on-the-fly that should not be underestimated, especially when it is paired up with a mind-bending revolutionary new technology such as AOP or transparent injection of services. Clashes that may happen when multiple agents are involved will increase this skepticism.

Another potential problem is the 64K boundary for class files stated in the Java specification. Method bodies are limited to a 64K total bytecode instruction size. This might be a problem when weaving already large class files, like for example the resulting class file when compiling a JavaServer Pages (JSP) [17] file to a Servlet [18]. When instrumenting this class, it might break the 64K limit and then cause a runtime error.

4. PROPOSED SOLUTION

4.1 JVM support for weaving

JVM weaving is the natural answer to most of the issues discussed above. The following examples show that the JVM is already doing most of the work involved to do weaving: When a class gets loaded, the JVM does read the bytecode to build up the data needed to serve the `java.lang.reflect.*` API purpose. Another example is method dispatching. Modern JVMs compile the bytecode of methods or code blocks to more advanced and efficient constructs and execution flows (doing code inlining where applicable). Due to the HotSwap API requirements, the JRockit JVM (and probably other JVMs too) also bookkeeps which method calls which other method, so that a method body can still be hotswapped in all expected places - inlined or not - if its defining class is redefined at runtime.

As a consequence, instead of changing the bytecode to weave in an advice invocation - say before a specific method call - the JVM could actually have knowledge about it and simply do a dispatch to the advice at any matching join point prior dispatching to the actual method.

As bytecode would be untouched, immediate advantages can be expected such as

- no startup cost due to bytecode instrumentation
- full runtime support to add and remove advices at any place, any time, at linear cost
- implicit support to advise reflective invocations
- no extra memory consumption to replicate the class model to some framework specific structures

This is very different from the C level events that have been defined in the JVMDI specification [10], such as JVMDI_EVENT_METHOD_ENTRY or JVMDI_EVENT_FIELD_ACCESS. In the JVMDI case, first one would have to deal with C level API, which makes it complex for most developers and fragile or complex to distribute, and second the specification does not provide a fine grained join point matching mechanism but actually requires ones to subscribe to all such events, thus still happening with an undeniable overhead - hence the D(ebug) in "JVMDI".

The following code sample introduces the JRockit weaving API that will be detailed in the next section.. The program below dispatches to the static method `advice()` just before the `sayHello()` method gets called :

```
public class Hello {
    // -- The sample method to intercept
    public void sayHello() {
        System.out.println("Hello World!");
    }

    // -- Using the JRockit JVM support for AOP
    static void weave() throws Throwable {
        // match on method name
        StringFilter methodName =
            new StringFilter(
                "sayHello",
                StringFilter.Type.EXACT
            );

        // match on callee type
        ClassFilter klass = new ClassFilter(
            Hello.class,
            false,
            null
        );
        // advice is a regular method dispatch
    Method advice =
        Aspect.class.getDeclaredMethod(
            "advice",
            new Class[0]
        );

    // Get a JRockit weaver and subscribe
    // the advice to the join point picked
    // out by the filter
    Weaver w = WeaverFactory.createWeaver();

    w.addSubscription(new MethodSubscription(
        new MethodFilter(
            0,
            null,
            klass,
            methodName,
            null,
            null
        ),
        MethodSubscription.InsertionType.BEFORE,
        advice
        ));
    }

    // -- Sample code
    static void test() {
        new Hello().sayHello();
    }

    public static void main(String a[])
        throws Throwable {
        weave();
        test();
    }

    // -- Sample aspect
    public static class Aspect {
        public static void advice() {
            System.out.println("About to say: ");
        }
    }
}
```

4.2 Subscription and action based model

The JRockit JVM AOP support exposes a Java API that is deeply integrated in the JVM method dispatching and object model components. In order to not tie the JVM to any current or future AOP specific technology direction we have decided to implement an action dispatch and subscription model.

The API allows defining well defined subscriptions at specified pointcuts, for which it is possible to register one action to which the JVM will dispatch. An action is composed of

- a regular java method - that we will reference as the action method - that will be invoked for each join point that matches the subscription
- an optional action instance on which to invoke the action method
- an optional set of parameter-level annotations that dictate the JVM which arguments the action method is expecting from the invocation stack.

The action can be flagged as a before action, an after returning action, an after throwing action or an instead-of action (similar to AOP around concept).

In order to invoke the API, one has to get a handle to a `jrockit.ext.weaving.Weaver` instance. The weaver instance will control which operations are allowed according to its caller context. For example you might not want a deployed application in an application server to create a weaver to subscribe action methods to some container level or JDK specific join points, while a container level weaver may actually subscribe to application specific join points. This weaver visibility concept mirrors the visibility rules from the underlying class loaders delegation model.

A very simple comparison of how these constructs are mapped to the regular AOP constructs might shed some light on this model:

- the subscription can be seen as a kinded pointcut, or actually a kinded pointcut (field *get(),set(),* method *call()*, etc) composed with a *within()/withincode()* pointcut.
- the action instance can be seen as the aspect instance
- the action method can be seen as the advice

Readers familiar with AOP may already understand that to implement a complete AOP framework on top this JVM level API, some more development will be required. An intermediate layer which will manage the aspect instantiation models (per clause), implement the *cflow()* pointcuts, and implement full pointcut composition and orthogonality is required.

4.3 The action method

An action method (similar to the AOP advice concept) is as a regular Java method of a regular class (that will act as the aspect). It can either be static method or member method. Its return type has to follow some implicit conventions. Its return type should be void for a before action, and should be of the type that will be placed on the stack as the result of the action invocation for an instead-of action (similar to AOP around advice semantics).

The action method can have parameters, whose annotations further control context exposure as illustrated in the following code sample:

```
public class SimpleAction {
    public static void simpleStaticAction() {
        print("hello static action!");
    }
    public void simpleAction() {
        print("hello action!");
    }
    public void simpleAction(
            @CalleeMethod WMethod calleeM,
            @CallerMethod WMethod callerM) {
         print(callerM.getMethod().getName());
         print(" calling ");
         print(calleeM.getMethod().getName());
     }
}
```

This code sample introduces the `jrockit.ext.weaving.WMethod`. This method acts as a wrapper for `java.lang.reflect.Method`, `java.lang.reflect.Constructor` and the class' static initializer which is not represented in java.lang.reflect.*. This is similar to the AspectJ `JoinPoint.StaticPart.getSignature()` abstraction.

In order to support instead-of and the ability to decide whether to proceed the interception chain or not (as implemented in AOP through the concept of `JoinPoint.proceed()`) we also introduced the `jrockit.ext.weaving.InvocationContext` construct as illustrated below.

```
public class InsteadOfAction {
    public Object instead(
            InvocationContext jp,
            @CalleeMethod WMethod calleeM) {
        return jp.proceed();
    }
}
```

4.4 The action instance and the action kind

As illustrated in the previous code samples, an action method can be either static or not. If the action method is not static, ones need to pass in an action instance on which the JVM will invoke the action method.

This follows the syntax style with which a Java developer would invoke a method reflectively using `java.lang.reflect.Method.invoke(null/*static method*/, .../*args*/)`. Although, as opposed to this example, with JVM AOP support, the underlying action invocation will not involve any reflection at all.

Giving the user control over the action instance opens up many interesting use cases. For example, one could implement a simple delegation pattern to swap a whole action instance with a different implementation at runtime, without having to involve the JVM internals.

For addition, this will be useful to implement AOP aspect instantiation models (per clause) such as *issingleton(), pertarget(), perthis(), percflow()* etc, while not locking the JVM API to some predefined semantics.

Before registering the subscription to the weaver instance, it is given a kind that acts as the advice kind: *before, instead-of, after-returning* or *after-throwing*.

The code to create a subscription is as follow:

```
// Get a Weaver instance that will acts a
// container for the subscription(s) we create
Weaver w = WeaverFactory.getWeaver();

// Regular java.lang.reflect is used to refer
// to the action method "simpleStaticAction()"
Method staticActionMethod =
    SimpleAction.class.getDeclaredMethod(
        "simpleStaticAction",
        new Class[0]//no arguments
    );

MethodSubscription ms = new MethodSubscription(
    .../* where to match*/,
    InsertionType.BEFORE,
    staticActionMethod
);
w.addSubscription(ms);

// Use of an action instance to refer to the
// non static action method "simpleAction()"
Method actionMethod =
    SimpleAction.class.getDeclaredMethod(
        "simpleAction",
        new Class[0]// no arguments
    );

// Instantiate the action instance
SimpleAction action = new SimpleAction();
MethodSubscription ms2 = new MethodSubscription(
    ...,// where to match, explained below
    InsertionType.BEFORE,
    actionMethod,
    actionInstance
);
w.addSubscription(ms2);
```

AOP semantics such as *within()* and *withincode()* type patterns are also implemented through variations around this API.

4.5 Subscription on events

As shown in the previous code sample, the subscription API is relying on the `java.lang.reflect.*` object model and some simple abstraction like `jrockit.ext.weaving.WMethod` to unify Method, Constructor and class' static initializer handling.

Subscriptions can be made on events triggered by: field access and modification, method and constructor calls, exception throwing and catching as well as static initialization of a class.

If we, for example, look at a method subscription construct, the first parameter in the call to `new jrockit.ext.weaving.MethodSubscription(...)`, must be a `jrockit.ext.weaving.Filter` instance, which has several concrete implementations, to match on methods, fields etc.

A `jrockit.ext.weaving.MethodFilter` instance will act as the definition on which the JVM weaver implementation will do the join point shadow matching for method and constructor call pointcuts. A `MethodFilter` allows filtering on modifiers, annotations, declaring type, name, return type (it also exposes extra structures to support within()/withincode() semantics):

The user can also pass in a `jrockit.ext.weaving.UserDefinedFilter` instance to implement a finer matching logic. The `UserDefinedFilter` callback mechanism is used to implement more advanced matching schemes (a concept that is similar to Spring AOP's `MethodMatcher` and `ClassFilter`).

All of these structures are optional, and if null is encountered, it means "match any".

The following will thus match all method calls whose name starts with "bar". Note that we pass in several null values in this very simple case:

```
StringFilter sf = new StringFilter(
    "bar", STARTSWITH
);

MethodFilter mf = new MethodFilter(
    0, null, null, sf, null, null
);

MethodSubscription ms = new MethodSubscription(
    mf,
    InsertionType.BEFORE,
    staticActionMethod
);
w.addSubscription(ms);
```

5. DISCUSSION

5.1 Benefits

There are several benefits in using JVM weaving instead of bytecode instrumentation. From an high level perspective, the weaving appears as a natural extension to the JVM capabilities. It is less intrusive in many ways with performance, scalability and usability benefits.

5.1.1 No more bytecode instrumentation increases scalability

The bytecode is not modified. The regular compilation pipeline from bytecode to executable code is followed in the JVM internals. There is no more need to parse the bytecode instructions and represent them in some intermediate structures.

The weaver becomes ubiquitous. Even though ones may want to register subscriptions at startup time, it is no longer a requirement. This greatly reduces the startup time of an application since there is no need at all to analyze bytecode instructions in order to find the interesting join points. This also gives the opportunity to develop truly dynamic systems, allowing deployment and undeployment of aspects at any point in time without any extra overhead or complexity.

5.1.2 No more redundant bookkeeping of types decreases memory usage and improves scalability

As bytecode instrumentation does not occur anymore, there is no longer a problem with double bookkeeping the object model. The subscription API relies on the `java.lang.reflect.*` model that already provides this information in a familiar way to the Java developer.

5.1.3 Multiple agents are kept consistent

As the bytecode of the woven classes does not get modified, there is no more risk of conflicts between two different agents changing the bytecode in two different ways that might be incompatible - hiding properties of the original program from another. The registration order of the subscription acts as the precedence rule. If a class is marked as being `Serializable`, it means that no hidden structures needed to support the runtime execution of the woven advice is added, which means that regular serialization will be fully supported. Usually bytecode instrumentation techniques need to ensure that serialization is preserved (for example dealing with the `serialVersionUID` field).

5.1.4 Support for intercepting reflective invocations

By using JVM level method dispatching, all reflective calls (method invocation or field get or set) can be matched as if they were regular calls and all registered actions will then get triggered. This occurs without any extra cost or implementation specific details and complexity.

5.2 Limitations

Some fine-grained semantics that AspectJ defines are not easily addressed at the JVM level. AspectJ for example supports the concept of *preinitialization*, *initialization* and *constructor execution* pointcuts. A constructor execution pointcut will pick out the constructor as it appears in the source code, while the initialization pointcut will pick out all constructor execution(s) that lead to having an initialized instance, thus including this(...) constructor delegations. Such a difference is not easily handled by the JVM. It may actually also be compiler dependant where more aggressive inlining strategies may occur. Consider for example:

```
public class Timeout {
    int delay;
    String cause;
    Timeout(int aDelay, String aCause) {
        this.delay = aDelay;
        this.aCause = aCause;
    }
    Timeout(int aDelay) {
        this(aDelay, "unknown");
        // A compiler could produce
```

```
            // inlined equivalent:
            // -- no call to this(..,..) but instead:
            // this.delay = aDelay;
            // this.aCause = "unknown";
        }
}
```

Since we currently do not have a grammar (AST etc.) for defining pointcut expressions, but a Java API, supporting a very expressive and fine-grained pointcut language like for example the one in AspectJ, will require an additional abstraction layer on top of the existing API. Part of it is already available as the pointcut parsing and matching logic based on `java.lang.reflect.*` structures has been introduced in recent AspectJ 5 releases and is now accessible in the `org.aspectj.weaver.tools` package.

6. CONCLUSIONS

Bytecode instrumentation techniques are now widely used in the Java platform in several different areas, ranging from Aspect-Oriented Software Development to more specific applied solutions such as application monitoring, persistence, or distributed computing. In an attempt to become more usable and transparent, the techniques of load time weaving and instrumentation at deployment time is becoming popular.

Unfortunately, such techniques do not provide the proper properties to match scalability and usability requirements, especially as it becomes more and more used, and mixed through the use of several different instrumenting agents from different products. JVM weaving and JVM support for AOP as implemented in JRockit happen to be a natural way approach to the problem and drive the innovation and the technology further. The proposed Java API that bridges JVM method dispatching internals to user defined action and subscription depending solely on the `java.lang.reflect` API fills the gap elegantly, as well as addressing major scalability and usability issues.

Nevertheless, widespread adoption requires a good assessment of this new API towards real use cases - such as AOP or runtime adaptability of large applications.

7. FUTURE WORK

Despite that JVM weaving brings huge value and addresses scalability and usability problems tied to bytecode instrumentation techniques, there are still some interesting drawbacks that will need to be solved so that some use cases can be addressed completely - possibly with complimentary approaches.

Some bytecode instrumentation based products are using very fine grained change that may not be possible to mirror in the (current) JVM AOP API. There are for example some use cases that deal with the synchronized blocks, so that different locking strategies - for example distributed - can be transparently injected into a regular application. Such a fine grained manipulation usually requires conditional execution of the synchronized block, or even complete removal of it so that it is replaced by some proprietary locking API call. Such a specific need can be addressed within the JVM but it is actually impossible to come with a solution that works for each use-case and that is efficient. It is also interesting to note that leading AOP frameworks are not (yet) exposing the synchronized block (or monitor entry and monitor exit lock acquisition and release) as join points.

Native support for various action instance life-cycles is something that would be good to have. For example support for pure per instance based deployments of action instances, similar to the work done in SteamLoom [12] is interesting.

As bytecode instrumentation is gaining popularity, introducing such a new API is not neutral. It would represent a fairly high cost to have a product developed so that it works for JVM that would support this API - such as JRockit - and for JVM that would not support this API. A specification, for example a Java Community Process (JCP) [19], in this area would be beneficial.

8. REFERENCES

[1] Kiczales, G., Lamping, J., Mendhekar, A. ,Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J., Aspect-Oriented Programming, ECOOP1997

[2] Gosling, J., Joy, B., Steele, G. The Java Language Specification (2nd edition) Addison-Wesley, 2000.

[3] Bonér,J., Vasseur,A., AspectWerkz, a dynamic, lightweight and high-performant AOP framework for Java, 2002-2005. (http://aspectwerkz.codehaus.org)

[4] Kiczales,G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G., An Overview of AspectJ, ECOOP 2001.

[5] Press release, January 19th, 2005, "AspectJ and AspectWerkz to Join Forces" (http://www.eclipse.org/aspectj/aj5announce.html)

[6] AspectJ Development Tools – AJDT (http://www.eclipse.org/ajdt/)

[7] JVMTI (JSR-163 - http://www.jcp.org/en/jsr/detail?id=163)

[8] JRockit JVM, BEA Systems (http://www.jrockit.com)

[9] HotSwap API (JSR-163 - http://www.jcp.org/en/jsr/detail?id=163)

[10] JVMDI, Java Virtual Machine Debug Interface (http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jvmdi-spec.html)

[11] JSR-175, A Metadata Facility for the Java Programming Language (http://www.jcp.org/en/jsr/detail?id=175)

[12] Bockisch, C., Haupt, M., Mezini, M., Ostermann, K.: Virtual Machine Support for Dynamic Join Points. In: International Conference on Aspect-Oriented Software Development. (2004)

[13] JBoss EJB 3 (http://www.jboss.org)

[14] Terracotta Virtualization Server (http://www.terracottatech.com)

[15] Java Dynamic Proxy Classes (http://java.sun.com/j2se/1.5.0/docs/guide/reflection/proxy.html)

[16] The Spring Framework (http://www.springframework.org)

[17] JavaServer Pages Technology (JSP) (http://java.sun.com/products/jsp)

[18] Java Servlet Technology (http://java.sun.com/products/servlet/)

[19] Java Community Process (JCP, http://jcp.org)