Adapting Virtual Machine Techniques for Seamless Aspect Support

Christoph Bockisch* Matthew Arnold+ Tom Dinkelaker* Mira Mezini*

*Darmstadt University of Technology / +IBM T.J. Watson Research Center
bockisch@informatik.tu-darmstadt.de / marnold@us.ibm.com /
{dinkelaker, mezini}@informatik.tu-darmstadt.de

Abstract

Current approaches to compiling aspect-oriented programs are inefficient. This inefficiency has negative effects on the productivity of the development process and is especially prohibitive for dynamic aspect deployment. In this work, we present how well-known virtual machine techniques can be used with only slight modifications to support fast aspect deployment while retaining runtime performance. Our implementation accelerates dynamic aspect deployment by several orders of magnitude relative to mainstream aspect-oriented environments. We also provide a detailed comparison of alternative implementations of execution environments with support for dynamic aspect deployment.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—run-time environments

General Terms Languages, Measurement, Performance

Keywords aspect-oriented programming, virtual machine support, aspect weaving, dynamic deployment, envelope-based weaving

1. Introduction

Current approaches to compiling aspect-oriented programs are inefficient. This inefficiency has negative effects on the productivity of the development process and is prohibitive for dynamic aspect deployment. Elsewhere we have presented a new compilation technique for aspect-oriented programs [13] which improves the compilation performance, however, at the cost of increased runtime overhead. In this paper, we show how well-known virtual machine optimization techniques can be used with only slight modifications to support fast aspect compilation while retaining runtime performance. Furthermore, these optimizations can be combined with established implementations of dynamic features like class loading to provide excellent performance for dynamic aspect deployment.

With aspect-oriented programming, a new kind of modules, i.e., *aspects*, is introduced. An aspect encapsulates a concern whose implementation would otherwise be scattered over several modules and tangled with the code of other concerns. There are several flavors of aspect-oriented programming [32]. The focus of this paper is on the pointcut-advice flavor [32], which is supported by the majority of aspect-oriented languages. A *pointcut* describes a set of points in the execution of a program; the projections of these points onto places in the program's code are called *join point shadows* [24, 33]. *Advice* are associated to pointcuts and specify functionality that is bound to join points selected by the pointcut. A more detailed discussion is provided in section 2.1.

In addition to the workload of ordinary source code compilers, current compilers for aspect-oriented languages perform additional operations, which involve transformations, called weaving, throughout the entire program code.

1. Join point shadows are searched for, and this search is expensive because it must involve the complete application code. Shadows are, generally, scattered throughout the code – a natural consequence of the crosscutting nature of the concern expressed in an aspect. For example, the shadows for a pointcut that selects all calls to a `public` method are spread throughout the program.
2. Advice invocation logic is woven in at several different locations. E.g., when each call to a specific method must be advised, all call sites are affected. Hence, an advice invocation weaving action must be repeated several times.
3. More importantly, weaving an advice call at each individual join point shadow, generally, is complex: Inserting instructions for advice invocation potentially entails updating other instructions and control structures like the exception handler map.

Aspects can be activated either before starting the application or dynamically, what we call *dynamic aspect deployment*. When a new aspect is deployed at runtime, its corresponding pointcut-advice pairs are introduced into a running system. Join point shadows of the pointcuts must be searched for and the respective advice calls must be woven[1]. That is, the aspect weaving overhead affects the runtime behavior of the system.

To improve performance of AOP compilers in previous work [13] we introduced a new technique for weaving pointcut-advice, called *envelope-based weaving*. With envelope-based weaving indirections are introduced between a method call or field access site and the respective class members being accessed. The indirection takes the form of a method, called *envelope*, one per each method or field of any class in the application. Instead of calling a method or accessing a field directly the respective envelope is called. The envelope represents the original member and, in turn, carries out the actual method call or field access.

[1] Aspect deployment may involve other operations, e.g., solving conflicts between aspects; these operations are outside the scope of this paper.

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.

Envelopes simplify the weaving process in three ways:

1. Join point shadows are localized exclusively inside envelopes; hence, their search is better localized.
2. Weaving has to be executed at less shadows - only within the envelopes and not e.g., at all call sites of a method.
3. Envelopes have a simple control flow, which makes the weaving action itself a cheap operation.

In a case study [13], we measured the effect of these simplifications on the efficiency of the weaving process by comparing the compilation time of a program once with a simple aspect and once without aspects: Compiling XalanJ [52] with a simple aspect using AspectJ's `ajc` compiler 1.2.1 [10] took up to 3 times longer than compiling the same application without the aspect using `javac`. The same experiment with envelope-based weaving only produced an overhead of 0.3 times.

Yet, without special support from the execution environment, envelopes also introduce some runtime penalty. When using envelopes for all methods and fields of application classes and without virtual machine support, we measured an overhead of 19.2% on average for the SPEC JVM98 benchmark applications (see section 7.1). Furthermore, the focus in [13] was on static compilation; important issues related to dynamic aspect deployment were left out of focus.

To avoid the runtime performance degradation, in this paper we propose to make the virtual machine (VM) *envelope-aware* so that it understands the concept of envelopes and can optimize accordingly. Current major VMs employ just-in-time compilation which is to dynamically compile bytecode at runtime into native machine code. In this step sophisticated optimizations are applied to the generated code. We show that existing dynamic optimization techniques – available in most modern VMs to improve performance in the presence of dynamic dispatch – can seamlessly be adopted to (made aware of) envelopes. Making the VM envelope-aware has a number of advantages:

1. *Eliminating performance overhead of envelopes*. The VM can ensure that envelopes are inlined for hot methods to avoid the level of indirection and, thus, provide performance identical to a system not using envelopes.
2. *Efficient dynamic aspect deployment*. Only envelopes are modified by the weaving, and we can use existing efficient invalidation mechanisms of the VM to replace envelopes after modifications.
3. *Advising reflective operations*. Since Reflection [27] is a service of a virtual machine, it is only possible to support advising reflective method calls or field accesses by modifying this service in the virtual machine[2].

One might argue that adding AOP-specific optimizations to the Java virtual machine (JVM) also has disadvantages, such as increasing the complexity of the JVM, and increasing the dependencies between the JVM and AOP constructs. However, aspect-oriented programming is a new programming paradigm, just like object-oriented programming was a new paradigm as compared to structural programming. As such, it is natural to design and implement VMs dedicated to aspect-oriented languages just as dedicated VMs for object-oriented languages exist.

The performance is likely to be a key factor in whether aspects are adopted mainstream and treated more as a language feature. Applying targeted optimizations in the virtual machine is an effective way to achieve optimal performance; the goal of this research is to advocate and evaluate this approach by investigating its performance potential.

The primary contributions of this paper are as follows:

- We describe how a VM can be modified to become *envelope-aware* to obtain the compilation performance benefits of envelopes while minimizing the runtime performance drawbacks.
- We describe how existing dynamic optimization technology can be used to achieve efficient dynamic deployment.
- We describe our design and implementation of the aforementioned techniques in Jikes Research Virtual Machine [28].
- We present experimental results describing the effectiveness of our approach and provide a comprehensive comparison between different implementations of deployment.

The remainder of this paper is structured as follows. The following section gives background information about aspect-oriented programming, envelopes, aspect weaving and dynamic optimizations in virtual machines. Section 3 provides an overview of our approach for making a virtual machine envelope-aware. Sections 4 and 5 present alternative optimizations for reducing the overhead of envelopes and for efficiently invalidating compiled code at aspect deployment. Related aspect-oriented execution environments are presented in section 6 and evaluated in section 7.1. Section 7.2 evaluates alternative optimization implementations. Section 8 concludes the paper and discusses areas of future work.

2. Background

In this section, we will provide some background on aspect-oriented programming, envelopes, Java bytecode, conventional and envelope-based weaving, and dynamic optimizations in virtual machines. Readers familiar with these concepts may want to skip the particular sub-sections. The following sub-section introduces aspect-oriented programming and basic terminology by means of a small example. In section 2.2 we summarize our previous work [13] in describing the semantics and the terminology of envelopes. We give a comparative discussion of conventional and envelope-based aspect weaving in section 2.3. Finally, section 2.4 discusses established optimization techniques.

2.1 AOP Background

For an introduction to the pointcut-advice flavor of aspect-oriented programming consider an application using the `Services` class (listing 1) to access data from a database. The application distinguishes two different trust contexts from which database accesses may happen. Internal requests to data are trusted and originate from the method `Facade.internal()`. In contrast, external requests come through `Facade.external()` and are not trusted. To avoid security leaks, external requests should be validated before they may execute.

Listing 1 shows an excerpt of the `Services` class with two service methods that execute an SQL query on the database. The query is passed as a string parameter. Using parameters that originate from possibly untrusted sites in SQL statements bears the risk of an SQL command injection attack where unexpected parameter strings are used to alter the effect of the SQL query, e.g., in order to gain unauthorized access to data. Such attacks can be prevented by checking unsafe SQL statements before sending them to the database. The sequence diagram in figure 1 shows a possible execution that uses the `Services` class from listing 1.

```
1 class Services {
2   Database db;
```

[2] A weaver could also instrument reflective operations to check at runtime whether there is an advice for the operation to be executed reflectively and to execute the advice if this is the case. However, an aspect-aware reflection mechanism provided at VM-level is more appropriate [51].

```
3
4   void service1(String sql) {
5     // ...
6     db.execQuery(sql);
7   }
8
9   void service2(String sql) {
10    // ...
11    db.execQuery(sql);
12  }
13 }
```

Listing 1. Example `Service` class.

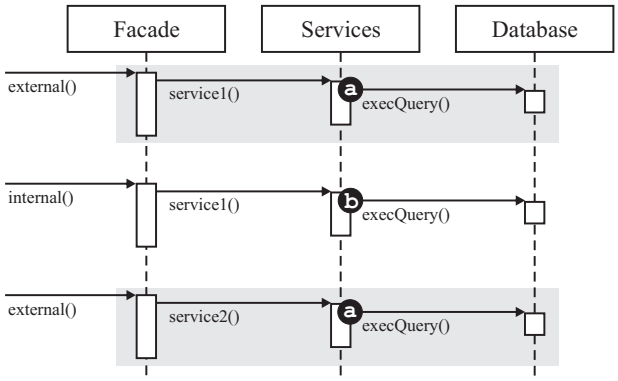

Figure 1. Sequence diagram for an example execution using class `Service`.

In listing 2, the checks for the example (listing 1) are modularized into the aspect named `QueryInjectionChecker` provided in the syntax of the AspectJ language [10]. First, the aspect defines *when* to check SQL statements by specifying the *pointcut* `untrustedDBCalls` in lines 3 to 6. Pointcuts are expressions that refer to a set of *join points* which are points in the execution of a program. In the example, join points selected by `untrustedDBCalls` are the method calls to the `execQuery()` method marked with index `a` in the sequence diagram (figure 1). Second, the aspect defines *what* should happen at these join points, in terms of *advice* which is a piece of code associated with the pointcut (lines 8 to 11). The advice also specifies whether the actions it defines should be executed before, after or around selected join points.

```
1  aspect QueryInjectionChecker {
2
3    pointcut untrustedDBCalls(String sql):
4      call(ResultSet Database.execQuery(String))
5        && args(sql)
6        && cflow(void Facade.external(Request));
7
8    before(String sql): untrustedDBCalls(sql) {
9      if (isAttack(sql))
10       throw new SQLInjectionException();
11   }
12
13   // ...
14 }
```

Listing 2. Aspect checking for SQL command injection.

The pointcut given in the example consists of several sub-expressions. The `call` sub-expression (line 4) selects all call sites to `execQuery()`. But pointcuts can also select join points based on their *dynamic context*. In our example, this is used to select only requests which are in in the control flow of the `external()` method, by means of the `cflow` sub-expression in line 6. For illustration, the gray boxes in the sequence from figure 1 mark the control flow of the `external()` method in that sequence.

Pointcuts can also be used to reify the join point's *context*. This means that values from the context are passed as arguments to the associated advice. The `args` sub-expression in line 5 (listing 2) reifies the argument of the method call. It is bound to the name `sql` by which the advice can access it to check the SQL query for command injection.

The predominant implementation approach for the pointcut-advice flavor of AOP builds upon the following idea: While join points generally depend on runtime execution properties, it is possible to locate a set of code locations, called *join point shadows* [24, 33], whose execution potentially yields a join point. For example, the join points for calls to the method `execQuery()`, which are selected by the pointcut in our example aspect, have join point shadows in lines 6 and 11 in the `Service` class in listing 1. As can be seen in figure 1, not all executions of joint point shadows (index `a` and `b`) really are join points but only those which also match the dynamic context specified in the pointcut (index `a`).

In current approaches, the aspect *weaver* is the part of the compiler that is responsible for inserting aspects into the program. It finds all join point shadows in an application for a given set of pointcuts; a call to the advice functionality, which is mapped to a Java method, is inserted at these shadows [34]. If necessary, the weaver also generates a check for the dynamic properties, which possibly skips the advice invocation. In our example, the weaver generates such checks because of the `cflow` sub-expression (listing 2, line 6). The checks test whether execution is in the control flow of the `external()` method. The approach just sketched is pursued by AspectJ's [30] `ajc` [10] and `abc` [5] compilers, the CaesarJ compiler [7, 17], Steamloom [23, 14, 43], AspectWerkz [11, 15] and JAsCo [46] with the list being non-exhaustive.

2.2 Envelopes Background

To support envelope-based weaving [13], the application has to be transformed into the envelope-style. All call or access sites to members in the application are replaced with an invocation of a corresponding *envelope*[3]. The envelope is a method that merely calls the original member it replaces. By introducing such an indirection, all direct accesses to any class member are localized in a single place.

Envelopes, which are modeled as Java methods called *envelope methods*, can be inserted into programs by using bytecode modifications. There are two steps in the transformation process: (a) envelopes are added for all members of a class, and (b) the bytecode of the program is changed to use the envelopes instead of the members they envelop. In our current work we support field accesses and method calls[4], though envelopes for other join point models can be thought of.

In our model, there are three kinds of envelope methods (*envelopes* for short): *Proxy envelopes* for method calls (also *proxy* for short), *getter* and *setter envelopes*, for field accesses (also, *getter*, respectively *setter* for short). We call the latter two uniformly *accessor envelopes* (or *accessor*). A method for which a proxy is generated is called an *implementation method* of the proxy, a field for which an accessor is generated is called an *enveloped field*. The

[3] In section 4.2 we will show that lazy introduction of envelopes yields better performance, conceptionally, though, this indirection exists from the start of the application.

[4] Though AspectJ distinguishes between method *call* and *execution* join points, we support both by means of method call envelopes.

body of an envelope method basically consists of a method call, respectively a direct field access.

```
1  class C {
2    public C() {init();}
3
4    private int field;
5
6    public void init() {
7      int f = field;
8      ...
9    }
10 }
```

Listing 3. Original class.

Listing 3 shows a simple Java class that is transformed to envelope-style as shown in listing 4. For each field declared in the class `C` (listing 3, line 4) a getter and a setter method is created (listing 4, lines 6-7). The names of the generated methods encode the fully qualified field name and the kind of access. To be able to access accessor envelopes in the same way as the fields, they have the same visibility, and for static fields accessors are also declared static[5].

```
1  class C {
2    public C'() {this();}
3    private C() {init'();}
4
5    private int field;
6    private int get_C_field() {return field;}
7    private void set_C_field(int v) {field = v;}
8
9    public void init'() {init()};
10   private void init() {
11     int f = get_C_field();
12     ...
13   }
14 }
```

Listing 4. Class in the envelope-style.

Proxy envelopes are an indirection inserted between the actual call site and the actually executed callee. Since abstract methods, including methods declared in interfaces, do not have an implementation, they can never be the actual target of a call and do not need an envelope. For all other kinds of methods we create envelopes which call their implementation method, as shown in lines 2 and 9 of listing 4.

A proxy envelope gets the same modifiers, formal parameters, return type, and `throws` clause as its implementation method. The name is derived from the implementation method's name – in listing 4 it is presented as the implementation method's name with an additional `'` charecter at the end.

Finally, the implementation methods must be transformed in order to ensure that envelopes are called instead of directly calling a method or accessing a field. Listing 3 shows the call to a method (line 2) and the access to a field (line 7) before the introduction of envelopes. Lines 3 and 11 in listing 4, illustrate how these calls are transformed to insert the envelopes into the call chain.

The presented bytecode modifications can be performed with simple operations, in one pass, and for each class individually, e.g., by using a post-processor as in [13]. In the implementation presented in this paper, they are performed at class loading-time.

[5] Fields can also be defined as constants in interfaces, but it is not valid in Java to add accessor methods to interfaces. However, since compile-time constants will be inlined in the application's bytecode and access to inlined constants can not be detected at runtime, it does not make sense to support advising access to constants, anyway.

2.3 Conventional vs. Envelope-based Weaving

Weaving into envelopes is far simpler than weaving into normal methods. This is because envelopes have a control flow structure that does not need costly updates when weaving into. As well, there is no exception handling in envelopes. Hence, weaving into envelopes does not require costly updates of the code structure.

In contrast, other weaving techniques directly weave advice calls into the method at join point shadows of member accesses. A problem with this approach is that weaving must take care not to destroy the control flow structure of method's bytecode. In the following, we present an example with several bytecode excerpts in order to show the differences between conventional and envelope-based weaving in detail.

In listing 5, we show a class `Timer` with a field member `startTime`. The method `setStartTime()` (lines 5–11) sets the field `startTime` (line 3) to the value of parameter `t`, in case `isValid(t)` is `true`. Method `isValid()` simply checks if `t` is below the limit 100 (line 17). If the time value is negative, an `IllegalArgumentException` is thrown (lines 15–16).

Further, listing 5 shows an aspect `MyAspect` (lines 21 to 24) that advises access to the `startTime` field. The aspect is woven into the class, once with the conventional and once with the envelope-based weaving approach, and the results are compared in the following.

```
1  public class Timer {
2
3    int startTime;
4
5    void setStartTime(int t) {
6      try {
7        if(isValid(t)) {startTime = t;}
8      } catch (IllegalArgumentException e) {
9        MyApp.showErrorDialog();
10     }
11   }
12
13   static boolean isValid(int t)
14       throws IllegalArgumentException {
15     if (t < 0)
16       throw new IllegalArgumentException();
17     return (t < 100);
18   }
19 }
20
21 aspect MyAspect {
22   before() : set(int Timer.startTime)
23     { /* advice */ }
24 }
```

Listing 5. The `Timer` class.

Figure 2 shows the bytecode and the exception table of the method `setStartTime()`. A single instruction in the bytecodes can be addressed by its *bytecode index* (bc-index). The *exception table* specifies how to handle exceptions occurring in specific code regions. Each row defines a division of bytecode for which exceptions will be caught. The start and the end bytecodes of that division are determined by the indices in columns `from` and `to`. If an exception of the type specified in column `type` occurs while executing the bytecode division, exception handling code is executed by continuing with the instruction at the index in column `target`. Normal execution passes the instructions at bc-index 0–12 and 19. As can be derived form the exception table, an `IllegalArgumentException` is handled by the instructions from bc-index 15 to 16.

2.3.1 Conventional Weaving

The box in figure 2 that borders the instruction at bc-index 9 marks the join point shadow that is matched by the pointcut of `MyAspect`

```
int startTime;

void setStartTime(int);
  0:   iload_1
  1:   invokestatic    #isValid(int)
  4:   ifeq      +8
  7:   aload_0
  8:   iload_1
  9:   putfield        #startTime
  12:  goto      +7
  15:  astore_2
  16:  invokestatic    #showErrorDialog()
  19:  return

Exception table:
from  to  target type
  0   12  15     IllegalArgumentException
```

Figure 2. Original bytecode of `setStartTime()`.

specified in line 22 in listing 5. In the case of conventional weaving, the box marks the location at which call to advice is directly woven in.

The results of weaving `MyAspect` with the conventional weaving approach are shown in figure 3. All places in the bytecode excerpt that have been subject to bytecode manipulation are marked with gray boxes. Before the matched join point shadow, a call to the advice method is inserted, which has the size of three bytes. As a consequence, all following bytecodes (up from former bc-index 9) are moved downward three bytes and their bc-indices are changed. Note that the branch instruction at bc-index 4, still points to the `goto` instruction which now is at bc-index 15. Because the `goto` was moved, the relative offset of the jump had to be updated (gray box, bc-index 4). The same is true for the bc-indices in the exception table.

```
void setStartTime(int);
  0:   iload_1
  1:   invokestatic    #isValid(int)
  4:   ifeq    +11
  7:   aload_0
  8:   iload_1
  9:   invokestatic    #advice()
  12:  putfield        #startTime
  15:  goto    +7
  18:  astore_2
  19:  invokestatic    #showErrorDialog()
  22:  return

Exception table:
from  to  target type
  0   15  18     IllegalArgumentException
```

Figure 3. Bytecode of `setStartTime()` after conventional weaving.

2.3.2 Envelope-Based Weaving

Figure 4 shows an excerpt of the modified bytecode after inserting the envelopes. For illustration, at index `A` the structure of a plain envelope (here for setter `set_startTime()`) is shown, i.e., an envelope without advice that only consists of a *member access block* and a *return block*. In general, the member access block contains instructions that either call a method or perform a field access. The return block terminates the envelope's execution and possibly returns the result. In our example, the member access block (bc-indices 0–2) only writes the field and the return block (bc-index 5) simply returns without result.

```
int startTime;

void setStartTime(int);
  0:   iload_1
  1:   invokestatic    #isValid(int)
  4:   ifeq    +8
  7:   aload_0
  8:   iload_1
  9:   invokevirtual   #set_startTime()
  12:  goto    +7
  15:  astore_2
  16:  invokestatic    #showErrorDialog()
  19:  return

Exception table:
from  to  target type
  0   12  15     IllegalArgumentException

void set_startTime(int);
  0:   aload_0                          A   member
  1:   iload_1                              access
  2:   putfield        #startTime
  5:   return                              return

int get_startTime();
  ...

static boolean isValid(int);
  1:   iload_0
  2:   invokevirtual #isValid'(int)
  5:   ireturn
```

Figure 4. Bytecode after inserting envelopes.

With envelope-based weaving, the call to advice happens in so-called *advice blocks*. In figure 5 bc-index 0, such an advice block (index B) has been inserted into the envelope. Because the inserted advice block has the length 3, the indices of the subsequent bytecodes of the member access and the return block have been increased by 3. Note that no existing instruction or exception table needs to be updated.

```
void set_startTime(int);
  0:   invoke          #advice0()  B   advice
  3:   aload_0                         member
  4:   iload_1                         access
  5:   putfield        #startTime
  8:   return                          return
```

Figure 5. Envelope with an advice.

Due to the block structure of envelopes, weaving inside of them is very simple. Conventional approaches have to cope with substantially more complexity, and as a result often manage this complexity using bytecode toolkits [13]. The toolkit helps to update

structures of manipulated methods such as jump targets and exception handler tables; however, these updates have additional costs, and have a significant effect on weaving performance. Furthermore, bytecode toolkits generally introduce overhead on their own due to high memory demands.

2.3.3 Weaving Conditional Logic

To realize dynamic pointcuts which only match, when a join point shadow is executed in a specified dynamic context, a *conditional advice block* is used, which in addition to the advice call contains a dynamic test that checks a condition.

In figure 6, a conditional advice block (index C) has been inserted into the sequence (bc-indices 0–4). If the test (bc-indices 0–1) fails the advice execution is skipped (bc-index 1) by jumping to the next block in the sequence (bc-index 7).

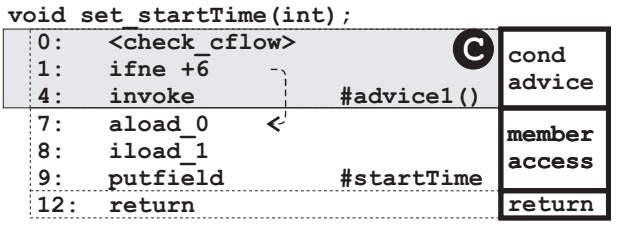

Figure 6. Envelope with a conditional advice.

2.3.4 Multiple Advice Weaving

In the weaving process, several advice blocks may be inserted into the sequence of blocks in an envelope. When executing an envelope method, the blocks are executed in sequential order. After weaving an advice block, relative jump offsets in the blocks need never to be updated, because they only point to the instruction following the last instruction of the current block. Note that, in figure 7, even though another advice block (index D) has been woven into the envelope from figure 6, the relative offset of the jump at bc-index 1 needs not to be changed.

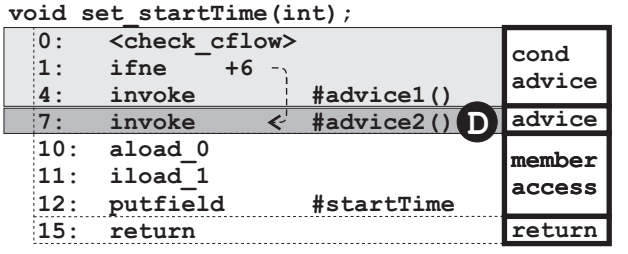

Figure 7. Envelope multiple advice.

2.3.5 Limitations of Envelope-Based Weaving Without VM Support

The improved weaving performance due to envelopes does not come for free. Envelopes change the program's structure, especially its call graph, affecting the non-functional behavior of the program. The additional indirections from envelopes degrade performance even if no aspect weaving takes place.

In our initial prototype [13], which worked completely at the bytecode level providing dynamic deployment by using the Class Redefinition feature of Java 5 [18], we observed a degradation of the application's performance of 19.2% in an experiment discussed in section 7.1. Also the general purpose Class Redefinition does not provide optimal performance for dynamic deployment. In [13] we measured the time needed for deploying an aspect to an application consisting of 11 classes. This deployment took 101 ms with our approach which is, although faster than competing implementations of dynamic deployment, not satisfactory.

Another drawback of the initial prototype is that, e.g., no envelopes can be generated for native methods and that reflective field accesses circumvent the envelopes.

2.4 Dynamic Optimization In Modern VMs

Modern virtual machines (VM) [35, 44, 6, 26, 21] contain advanced optimizing compilers, often called a *just-in-time* (JIT) compiler, as well as an *adaptive optimization system*. For efficient startup time, most VMs begin executing the program by using an interpreter or an efficient non-optimizing compiler (i.e., Jikes RVM's *baseline* compiler). As the program executes, the application is profiled to identify the *hot methods*, i.e., those methods that contribute the most to the program's execution time. Profiling is usually performed using invocation and loop counters, or by using a timer-based sampling mechanism.

Methods that are identified to be hot are then compiled with the optimizing compiler. Most JIT compilers in high-performing VMs contain multiple optimization levels. Lower optimizations levels consume less compile time but produce lower quality code; optimizations at high levels consume more compile time and perform aggressive optimizations to produce high-quality code. If a method continues to be identified as hot, it may be recompiled multiple times at higher optimization levels, until the highest level is reached.

One of the most important optimizations applied by a JIT compiler is *method inlining*. Inlining is made more complex for object-oriented languages which support dynamically dispatched (virtual) method calls, where the runtime type of the receiver object determines the method that is actually invoked. One technique for performing inlining at potentially polymorphic call sites is *guarded inlining* [45, 9], where a runtime test is inserted to check whether it is safe to execute the inlined code. If not, a full virtual dispatch is executed.

Inlining is complicated even further for dynamic languages like Java as classes may be dynamically loaded throughout the execution of the program. As a result, call sites that are currently monomorphic may become polymorphic in the future if class loading occurs. To achieve high performance in the presence of class loading, most JVMs perform *speculative method inlining* [19, 45, 9], as described below.

2.4.1 Speculative Method Inlining

The goal of speculative method inlining is to perform inlining based on the *current* class hierarchy, with the knowledge that the inlining decisions *can* be reversed, or *invalidated* when necessary when the class hierarchy changes. There are three primary steps when performing speculative inlining:

1. *Dependency tracking.* When inlining methods, the VM performs bookkeeping to identify which methods need to be invalidated when classes are loaded. For example, if class loading causes a call site `C` to become polymorphic, any compiled method that had inlining performed at call site `C` needs to be invalidated.
2. *Recompilation of invalidated methods.* When class loading occurs, the VM recompiles all affected methods (identified by the dependency tracking from step 1 above) and performs new optimizations based on the new class hierarchy. This ensures that future invocations of these methods will execute correctly.

3. *Invalidation of methods that are currently executing on the stack.* Although recompilation (from step 2 above) ensures that *future* invocations will execute correct code, the VM must also invalidate methods that are *currently* executing on the stack. For example, consider a method `main()` that is invoked only once and loops indefinitely. If inlining was performed at call site `C` in `main()`, and that inlining needs to be invalidated, the optimized code for `main()` needs to be updated *while it is executing* on the stack, to ensure that the remainder of execution executes the correct method at site `C`.

Steps 1 and 2 are accomplished fairly easily in a VM. Dependency tracking to map inlined call sites to compiled methods is easily maintained using a lookup table. Recompilation also is easy to perform because VMs already have advanced recompilation support for the adaptive recompilation infrastructure. The main challenge to speculative optimization is step 3: invalidating methods that are active on the stack.

There are multiple VM technologies for invalidating speculative optimizations in methods that are active on the stack. We briefly present two that are relevant in the context of this paper: *on-stack replacement* and *code patching*.

On-stack replacement. On-stack replacement (OSR) [25, 20] allows changing the compiled version of a method while it is active on the call stack. It can be used for several purposes, including deoptimizing code for debugging [25], undoing speculative optimizations in the presence of class loading [25, 20], and promoting long running methods to higher levels of optimization [20]. OSR can be performed lazily, one stack frame at a time, when the method returns from the caller frame. The disadvantage of on-stack replacement is that it is fairly complex, and is not implemented by all virtual machines.

Code patching. For VMs that do not implement OSR, guarded inlining combined with *code patching* can be used for speculative method inlining [45]. With traditional guarded inlining (as discussed in section above) a conditional instruction is inserted in front of the inline location to check whether the inlined code can be executed. Code patching can be used to gain the functionality of guarded inlining, but without the runtime overhead of executing the conditional test.

In a code patching approach, the conditional (or guard) is assumed to be true by default, and is compiled into a no-op instruction. Thus, the inlined code executes unconditionally and there is no overhead of executing a conditional test. If the assumptions change and the inlined method body is invalidated, the no-op instructions that guard the inlined code are dynamically replaced, or *patched* with a jump to a full dispatch to the method.

Code patching is effective at removing the runtime overhead of the guard itself; however, the existence of the slow path (the full dispatch) can still interfere with optimizations, resulting in poorer quality code than with OSR. Even though the guard starts out as a no-op, it represents a potential if-then-else structure in the method's control flow (where the 'else' clause is the full method dispatch). Instructions that follow the guarded code must be optimized assuming that either path may have been taken, thus data flow information from the inlined code cannot be propagated outside of the guarded region.

To enable more effective optimization in the presence of guarded code, techniques such as *code splitting* [45, 9] can be used to prevent the data flow information from the slow path (full dispatch) from merging back into the main control flow. While this approach increases method size, and thus compile time, we show in section 7.2 that it can be as effective as on-stack replacement in terms of the quality of the code produced.

3. An Envelope Aware VM

We address the limitations of the envelope-based weaving summarized in 2.3.5 by making the virtual machine (VM) aware of the envelopes. We used the Jikes RVM version 2.4.1 [28] for our extensions. The extended RVM is available for download at [2].

In this section, an overview of the VM modifications is given. Sub-section 3.1 presents how envelopes are generated in our modified virtual machine. Sub-section 3.2 outlines the optimizations we apply to avoid the runtime overhead due to indirections introduced by envelopes. Sections 4 and 5 elaborate on how we implemented these optimizations in the Jikes RVM. Sub-section 3.3 focuses on language features that need virtual machine level support for envelope-based weaving.

3.1 Constructing Envelopes

Applications are transformed into the envelope style at class loading-time. This is easily possible because the transformation can be executed in a single pass and separately for each class (cf. section 2.2 and [13]). When a class is loaded, proxies for all its methods and accessors for all its fields are generated and added to the class. Further, mappings between envelopes and their enveloped members are created at this point (later used by the virtual machine).

Other than in the conceptional discussion in 2.2 we rename the original methods and create proxy envelopes that have the original method name. This way, no method call sites need to be changed. This is different from the initial prototype [13] where renaming was not feasible for constructor methods; if renamed, the VM's bytecode verifier would not recognize them anymore as constructors and, thus, would regard them as illegal. Instead, the signature of methods and constructors was changed by adding a dummy argument which posed an additional overhead. By adjusting the constructor handling in the virtual machine renaming methods and constructors becomes possible in the VM integrated implementation of the envelope-based weaving.

Renaming the implementation methods also has implications for reflective method calls and the look-up for native methods which we will discuss in section 3.3.

3.2 Optimizations

Envelopes create two primary performance challenges to address:

1. *Reduce the overhead of envelopes to ensure high performance when no aspects are deployed.* Section 4 discusses two approaches for reducing the overhead of envelopes. The basic idea is that the JIT compiler can ensure that accessor envelopes and implementation methods are always inlined into their callers, thus eliminating the overhead introduced by the envelope's extra level of indirection.
2. *Ensure that dynamic deployment is performed correctly and efficiently.* A primary goal of this work is to support efficient dynamic aspect deployment, where the implementation of envelopes may change at runtime when new advice are woven into them. To enable correct dynamic deployment, any speculative optimizations that were applied must be invalidated when deployment occurs. The suite of techniques for speculative inlining, described in section 2.4.1 can be applied, enabling envelope methods to be invalidated and recompiled to include the dynamically woven advice. Section 5 discusses two approaches for invalidation to ensure correct dynamic weaving.

3.3 Support for Special Language Features and Join Point Context Reification

Handling native methods. The implementation of native methods (methods with the `native` modifier) is not available as Java

bytecode but as system dependent machine code in a dynamic library. To execute such a method the virtual machine searches the loaded dynamic libraries for a function providing its implementation. The look-up is based on a naming convention [29]: The function's name is derived from the Java method's signature. Since the introduction of envelopes renames original methods, this look-up mechanism had to be adapted accordingly.

Handling reflection. Reflection is provided in Java by means of an API [27] that allows to introspect classes as well as to select methods and fields by their names and call respectively access them. Reflection is affected by envelope-based weaving in two ways:

- First, methods are added to classes that should not be accessible via reflection; the programmer expects a class' structure to be as it is defined in the source code. To address this issue, Jikes' reflection were modified so that accessors and implementation methods are hidden from the programmer introspecting the application.
- Second, when the programmer deploys an aspect which advises calls to certain methods or accesses to certain fields, he/she expects that also reflective calls or accesses are advised. This issue is implicitly resolved for reflective method calls: As the original method names and signatures have been taken over by the proxies the proxy is automatically used instead of the implementation method. For field accesses some changes are necessary. A `Field` object from the Reflection API provides methods for setting and getting the value of the encapsulated field. The implementation of these methods was modified to call the respective accessor envelopes instead of performing a memory access to write or read the field.

Handling join point context reification. When an advice is executed at a join point, it can *access values from the context of the join point*. There are several such values supported by current aspect-oriented languages:

- Advice executing before a method call can declare to access the arguments of the method and the receiver object. Advice executed after a call can also access the value returned by the called method.
- Advice on field writes and reads can access the object whose field is written/read. For field write join points, also the field's new value is accessible.
- For all kinds of join points, the active object (`this`) can be accessed.

To make these values available, additional instructions that retrieve the required values and pass them to the advice call are inserted by the aspect weaver at join point shadows. This process is also called reification of values from the join point's context, or *context reification* for short. For an AspectJ compiler, all context values are available as local variables or in the operand stack at the join point shadow.

With envelope-based weaving, the situation is slightly different: The join point shadow is within the envelope rather than within the call site context which a programmer intends to advise. The question is how this affects context reification. Fortunately, all values but the active object are also available as local variables or in the operand stack of the envelope. It is also possible to access the active object (`this`), which is a local variable of the caller; we outline an implementation for it in section 8.

4. Reducing the Overhead of Envelopes

We explored two approaches to avoid performance degradation relative to an untransformed program due to introducing envelopes, each with different trade-offs:

1. *Eager envelopes.* Envelopes are created and eagerly inserted at class loading-time (as described in section 3.1) and the optimizer is adviced to inline them.
2. *Lazy envelopes.* Envelopes are created at class loading-time, but are not inserted into the invocation chain until they are advised. When advice is woven into an envelope, all calls to advised methods are replaced with calls to their corresponding envelopes.

In the following, each of these techniques is described in section 4.1, respectively 4.2. Subsequently, their trade-offs are discussed.

4.1 Eager Envelopes

With the eager approach in place, envelopes are invoked by default, thus dynamic deployment is simple and requires very few changes to the VM. Reducing the overhead of envelopes required a number of changes to the optimizing compiler's inliner to ensure that (a) envelopes are always inlined, and (b) other inlining decisions are made the same as if envelopes had not been present. Furthermore, a number of changes to other VM subsystems were done to account for non-obvious performance degradations caused by envelopes. These changes are discussed below.

4.1.1 Effects on the Baseline Compiler

The baseline compiler provided by the Jikes RVM does not offer the possibility to inline method calls. Consequently, we do not remove the indirection for baseline compiled methods. However, methods that are hot will be promoted to a higher level of optimization where envelopes are optimized accordingly. With the exception of the envelope call, the baseline compiler generates the same machine code for implementation methods as it does for the original method in the absence of envelopes.

4.1.2 Effects on the Optimizing Compiler

When compiling bytecode, Jikes' optimizing compiler successively transforms the bytecode to different intermediate representations which eventually results in machine code. At each such transformation step, different optimizations are applied. Initially, the bytecode is transformed into *high level intermediate representation* (HIR for short) based on a virtual instruction set.

The objective of our modifications to the optimizing compiler was to ensure that the HIR of methods is not affected by envelopes, since, this would guarantee that all subsequent optimizations are carried out without any modification. To achieve this goal, knowledge on envelopes was added into different modules of the optimizing compiler, as elaborated in the following.

Jikes' optimizing JIT compiler uses an *inline oracle* to make decisions about inlining methods. The questions asked to the oracle are of the form *"should method* callee *be inlined into method* caller*?"*, and the answer is either *no*, *yes*, or *yes with a guard*.

By construction, we ensure that implementation methods as well as accessor methods are always monomorphic. Hence, one would expect the inline oracle to always inline them into their caller. However, some of its heuristics prevent the oracle from always making these decisions. Thus, we modified the inline oracle to always decide to inline implementation methods and accessor methods. To facilitate dynamic deployment, we additionally enhanced the oracle to answer *yes with a guard* when an envelope (proxy or accessor method) is inlined.

Apart from this, we also had to make sure that the remainder of inlining decisions is the same as it would be without envelopes, to receive the same HIR. For this purpose, we modified three inlining heuristics:

- First, the oracle considers the size of the called method when deciding whether or not to inline at a call site. When the oracle is asked if a call to a proxy method should be inlined, it would make its decision based on the proxy method's bytecode size. Proxies interfere with size estimates because the proxy itself is small, but it is guaranteed to have a (possibly large) implementation method inlined into it. We modified the inliner so that it does not inline a proxy unless it would have inlined the corresponding implementation method.
- Next, the oracle ensures that the inline sequence does not exceed a fixed length to avoid excessive inlining and, thus, explosion of code size. An unmodified oracle would count the length of the proxy methods in the inline sequence length. As a result, it would eventually reach the maximum inlining length faster as it would without envelopes. Knowing that proxy methods will be optimized away and do not increase the size of the compiled code, we adjusted the inline sequence length heuristics accordingly.
- Finally, for values resulting from a field access, the optimizing compiler can apply special analyses on receiver types which, in turn, are used by the inline oracle to eliminate guards. As field accesses are replaced by calls to accessor methods, we had to ensure that the same analyses are performed for values returned by an accessor.

4.1.3 Effects on the Adaptive Optimization System

Before optimization and inlining occur, envelopes execute as explicit method calls. As the program executes, the adaptive optimization system (AOS) will profile these calls as regular methods and try to optimize them. One might expect that an AOS in a VM would not be thrown off by the existence of many such calls, and would produce similar code as it does in their absence. Unfortunately, this was not the case. Jikes RVM would eventually converge on the same performance, but the existence of a large number of envelope calls pollutes the profile data, causing compilations to occur at different points, and delays optimization of some of the application methods. As a result, startup time is degraded and the application runs longer before reaching steady-state performance.

We addressed this performance issue with a simple change. We modified Jikes RVM's timer-based sampler so that if an implementation method is sampled at runtime, the stack is walked down one additional frame, thus crediting the sample to the proxy envelope instead. As a result, the proxy will be identified as a hot method and compiled; our inlining modifications discussed in the previous subsection, will ensure that the (hot) implementation method is inlined into the proxy and optimized, as well.

4.2 Lazy Envelopes

After observing the complexity of the solution required to achieve high performance with eager envelopes, we also explored an alternative approach of lazy envelopes. As mentioned at the beginning of the section, lazy envelopes are created at class loading-time, but are not invoked as long as they are not advised; instead, the implementation method (or field access) is directly executed. Thus, if no aspects have been deployed, the executing program is essentially identical to the original program without envelopes.

When advice is deployed at runtime, it is woven into the envelope method that corresponds to the relevant method or field access. Since envelopes are no longer being called by default, all code that executes the advised method (or field access) must be dynamically updated to invoke the newly advised envelope instead.

To implement lazy envelopes, both the baseline and optimizing compiler needed to be modified to:

1. check whether field accesses and method calls have advice, and if so, to invoke the corresponding envelope instead. In the optimizing compiler, this is performed early in the compilation process prior to inlining, during the construction of the intermediate representation.
2. record dependency information for all calls and field accesses, to identify code to be updated when advice is deployed.
3. have an invalidation mechanism available for dynamically redirecting calls and field accesses to calls to corresponding envelopes.

4.3 Eager vs. Lazy Envelopes

With eager envelopes, the inliner handled the inlining of envelopes, which automatically provided the necessary dependency tracking and invalidation mechanism. Since this functionality is already provided by the virtual machine, this approach is easier to implement than lazy envelopes and, thus, may be integrated into more virtual machines.

Inlining and dependency tracking had to be performed "manually" for lazy envelopes, and no modifications to the inlining heuristics and the adaptive optimizing system are necessary. Because with lazy envelopes, the code generated by the baseline compiler no longer contains unnecessary envelope calls, the overhead at program startup is minimized. In long running applications, however, where all hot methods are optimized, eager envelopes ultimately achieve the same result as lazy envelopes.

5. Invalidation of Speculative Optimizations

So far, we have described how Jikes RVM generates envelopes, and how the JIT compiler's method inlining is modified to generate efficient code. This section presents changes to Jikes RVM to enable dynamic weaving. When weaving occurs, the affected envelopes must be recompiled and all compiled code containing inlined copies of these envelopes must be invalidated.

To track method inlining dependences, we used Jikes RVM's existing method dependency database that is used for speculative inlining. When an envelope is inlined into a method, a dependency is added to this database. When dynamic deployment occurs, it calls Jikes RVM's existing invalidation routine passing the envelopes that have been invalidated. This call triggers the Jikes RVM's mechanism for invalidating all necessary compiled method bodies. The next invocation of these methods invokes the lazy compilation stub, triggering a baseline compilation of the the method.

After dynamic deployment has occurred, several compiled methods may have been invalidated, and need to be recompiled. The adaptive optimization system of the VM treats these methods as it would at program startup, i.e., initially compiling them at a low level of optimization, and eventually promoting them to higher levels over time. This approach smooths out the performance impact of dynamic deployment and avoids pauses that may occur if all affected methods were immediately optimized aggressively.

To invalidate methods that are currently active on the stack, we utilize two common techniques as discussed in section 2.4.1: on-stack replacement, and guarded inlining.

5.1 On-Stack Replacement

Jikes RVM contains an implementation of on-stack replacement, described by Fink-Qian [20]. The implementation in Jikes RVM is used for speculative inlining, and for promoting long-running

methods from baseline compiled code. When performing speculative inlining, a patch point is placed at all inlined call sites that may be invalidated. When invalidation occurs, the patch point is replaced with a jump to an *OSR point*, which triggers on-stack replacement. The key advantage of OSR is that if execution jumps to an OSR point, it never merges back into the body of the compiled code. Therefore, the dataflow information along this path does not merge back in and hinder optimizations based on forward dataflow.

Using Jikes RVM's mechanism of OSR-based speculative inlining would work reasonably well for envelopes; however, given the expected frequency of envelop calls in the code we optimize one step further. In our implementation, rather than placing a patch point and OSR point at each inlined call site, we place them after all yieldpoints and calls in the optimized method. When the method is invalidated, these points are all patched and OSR will be triggered once the next yieldpoint is reached, or when execution returns from a call.

To ensure correctness in this model, when invalidation occurs, the VM must wait for all threads to progress to the next yieldpoint to ensure that OSR was triggered if necessary. Threads that are further up the call chain will perform OSR when the stack frame is popped and the patch point after the call site is executed. A similar technique of waiting for threads to reach the next yieldpoint was used in [9] to allow removing additional redundant guards, thus creating larger regions of guard free code. In Jikes RVM, ensuring that all threads reach a yieldpoint is particularly easy because all non-executing threads are guaranteed to be already stopped at yieldpoints. At invalidation time, the VM simply needs to stop and restart the currently executing threads.

With eager envelopes, envelopes are not inlined by the baseline compiler, so invalidation is needed for the optimizing compiler only. With lazy envelopes, invalidation is needed in baseline code as well, so OSR points are placed in code compiled by both the baseline and optimizing compiler.

5.2 Guarded Inlining

As an alternative to OSR, we experimented with using code patching and guarded inlining to enable envelope invalidation. Inserting guards on all inlined envelopes is relatively easy using Jikes RVM's infrastructure for speculative inlining. For eager envelopes, we modified Jikes RVM's inlining routine (described in section 4.1) to stipulate that envelope methods require a code patching guard when inlined.

When using this approach to guard envelopes, two additional changes were necessary to Jikes RVM. First, Jikes RVM does not normally use guards in combination with static methods because static methods can not be affected by dynamic class loading. As a result, we needed to modify the locations in the JIT compiler that assumed a guarded inline must have a receiver object. Second, Jikes RVM may perform guarded inlining of polymorphic methods; for these polymorphic calls, a dynamic type test is used as a guard rather than code patching. When inlining polymorphic envelopes into implementation methods, we modified Jikes RVM to insert a patch point prior to the dynamic type test, allowing us to invalidate the implementation method when dynamic deployment occurs.

However, as mentioned in the last section, code patching combined with guarded inlining is not sufficient to eliminate all envelope overhead. Even though the guard itself has no runtime overhead, the existence of the failed guard path interferes with a number of optimizations. To address this problem, we extended Jikes RVM's implementation of code splitting and redundant guard elimination.

5.2.1 Splitting

Jikes RVM performs a simple local code splitting pass, to help exploit optimization opportunities created by guarded inlining. Using envelopes creates a larger number of guarded inlines, thus stressing this splitting infrastructure. We discovered a number of shortcomings in the Jikes RVM's existing simple heuristics. While it catches the most dominant case of back-to-back guarded inlines, it misses several common cases, such as nested inlining.

We wrote a new splitting algorithm in Jikes RVM to perform slightly more aggressive splitting. It is similar to the algorithm for feedback-directed splitting described in [8], but for simplicity does not use profile information. The algorithm maintains a worklist of merge points. A merge point for our algorithm is defined as a merge in the control flow graph where one or more incoming edges are known to be infrequent, and one or more are known *not* to be infrequent (or non-infrequent). It is desirable to eliminate these merge points so that the dataflow of the infrequent path does not pollute the regular path.

The worklist is initialized with all of the control flow merges created by guarded inlining (where one path is known to be infrequent) and splitting is performed as follows. A merge point is removed from the worklist and further processed; if the basic block at that merge point is below a size threshold, the block is duplicated. The infrequent paths are directed to the duplicated block, while the remainder of the paths remain unchanged. If duplicating the block created a new control flow merge (between infrequent and non-infrequent paths), the new merge is added to the worklist. The algorithm continues until the worklist is empty, or a space bound is reached.

The size thresholds for duplication can be varied to adjust the aggressiveness of the splitting. Similar to method inlining, more aggressive splitting has the potential to produce more efficient code, but will consume more compile time and compiled code space, thus the splitting thresholds may vary depending on the optimization level, with higher levels performing the most aggressive splitting.

While this splitting algorithm was designed to improve the performance of envelopes, we discovered that it also improved the performance of the base Jikes RVM (independent of using envelopes). The steady-state performance of the `mtrt` benchmark was improved by over 20%. We are in the process of contributing our splitting back to the Jikes RVM open source code base.

5.2.2 Redundant Guard Removal

A desirable side effect of performing splitting is that many guards may now be redundant and able to be removed completely. Jikes RVM performs redundant branch elimination, based on global value numbering. However, similar to the optimized placement of OSR points (section 5.1), we enable additional optimization by enforcing that all threads will be allowed to progress to the next yieldpoint at the time of invalidation. By enforcing this property, guards can be eliminated more aggressively; if invalidation did not occur by the time of the previous yieldpoint was executed, it is guaranteed not to occur until the next yieldpoint executes.

To exploit this property, we wrote a redundant guard removal optimization phase that focuses specifically on removing code patching inline guards. The algorithm removes redundant guards by exploiting the following property: if (1) the taken branch of guard A dominates guard B, and (2) no yieldpoints or method calls can be executed between A and B, then guard B is redundant (i.e., guaranteed never to fail) and can be removed. A method call can execute a yieldpoint, thus it is implied that no yieldpoints *or* calls can occur between A and B.

This optimization is implemented in Jikes RVM as a linear pass by traversing the dominator tree, and propagating guard "liveness". In this case, liveness means that the existing code is safely pro-

tected by an existing guard, so no further guards are needed. The "true" branch of a guard creates liveness, and liveness is killed by yieldpoints, calls, or a control flow merge that contains a non-live incoming edge.

In addition to improving envelop performance, this optimization also provided modest performance improvements for the base version of Jikes RVM (without envelopes), improving mtrt by about 3%. This optimization is also being contributed back to the open source code base.

6. Related Work

This section presents several other currently available systems that support dynamic aspect deployment, and describes the similarities and differences from our approach. Section 7 compares the performance of these systems and the implementation presented in this paper.

The systems discussed include AspectWerkz 2.0 [11], PROSE 1.3.0 [39], JAsCo 0.8.7 [3], Steamloom 0.6 [43], and our previous prototype [13] offering limited support for dynamic envelope-based weaving and using standard Java 5 *Class Redefinition* [18]. For a detailed discussion of the former four and other implementations also refer to [16].

6.1 AspectWerkz

AspectWerkz [11, 15] provides runtime weaving capabilities for sets of join point shadows which are specified before runtime. The weaving process [50] is divided into two distinct phases: Preparation and activation. In the preparation phase classes are transformed such that in the activation phase advice calls can be inserted at prepared join point shadows.

Preparation can be performed either by a post-compiler or a special class loader. When preparing, AspectWerkz replaces each join point shadow with a call to a wrapper, which is similar to introducing envelopes. In the activation phase, advice calls are inserted into wrappers, as in our approach. Wrappers that changed in this phase are replaced by means of the Java 5 standard feature Class Redefinition [18].

In contrast to our approach, the wrappers of AspectWerkz do not reduce the number of join point shadows where weaving happens. Instead of redirecting the call sites to a single wrapper generated for each callee, as done with envelopes, AspectWerkz merely generates one wrapper per member and per class in which it is called. Furthermore, the generation of wrappers in AspectWerkz *does* affect the generated machine code, i.e., the generated machine code with and without wrappers is not the same.

6.2 PROSE

PROSE [39] has a two layered architecture consisting of the *dynamic AOP engine* which is basically an API and the *execution monitor*. The AOP engine is used to dynamically deploy an aspect; when requested to do so, it determines the set of join point shadows and passes it to the execution monitor. The latter monitors the execution and calls the AOP engine back any time a join point shadow from the set is executed; in response to the callback, the AOP engine calls the corresponding advice.

There are two different implementations of the execution monitor. The first implementation [37, 38], which we will refer to as PROSE 1 in the following, uses the standard JVM tools interface (JVMTI) of Java 5 virtual machines; the JVM is instructed to generate events at join point shadows which are in turn handled by PROSE's AOP engine. The second version, PROSE 2 [36], implements the execution monitor by means of a modified Java Virtual Machine.

6.3 JAsCo

JAsCo [46] also uses a registry for aspects which stores all active pointcut-advice bindings. At potential join point shadows so called traps are called which notify the registry and let it execute advice if any is applicable. JAsCo implements two optimizations of this approach [47]. The first one is the Jutta compiler which generates custom implementations of single trap methods which directly call applicable advice, thereby circumventing the registry. Second, calls to traps are not inserted eagerly, but only when aspects are deployed. Class Redefinition is used to replace a method without traps with a version including traps at runtime.

The lazy insertion of traps is similar to the lazy introduction of envelopes, as discussed in this paper. Also, generating custom traps that contain woven code for specific join points bears similarities. However, unlike in our approach, the number of weaving locations is not reduced and the use of a bytecode toolkit is still necessary – JAsCo employs the Javassist toolkit [4].

6.4 The Steamloom VM

Steamloom [43, 23] is an extension to the Jikes RVM [28] with dedicated support for aspects and dynamic aspect deployment. In contrast to other approaches, e.g., JAsCo or AspectWerkz, that only use bytecode toolkits as an external means for weaving, Steamloom has integrated the bytecode toolkit BAT [12] into the VM. Similar to our approach, join point shadow search and advice weaving makes use of the VM's internal representation of loaded classes. Invalidating old versions of methods in which advice calls are woven makes use of standard features of the virtual machine; however, the VM's optimizations are not modified in order to more naturally support dynamic weaving.

Similar to our approach, Steamloom restricts the scope of join point shadow search for member accesses. To do so, for each member an index is stored that points to all join point shadows at which the member is accessed. When searching for join point shadows, Steamloom only searches the corresponding index instead of the complete bytecode. However, Steamloom does not localize the weaving operations for member accesses as still each retrieved shadow must be manipulated individually, and, hence, weaving performance is still degraded.

6.5 Envelope-Based Weaving With Class Redefinition

In [13], we have presented a dynamic envelope-based weaver that transforms Java bytecode classes through a post-processor and uses Class Redefinition [18] to exchange envelopes after weaving at runtime. The transformed bytecode conforms to the Java Virtual Machine Specification [31] and, hence, may run on any standard JVM.

Because no VM-level support exists for envelopes in the prototype, envelopes must be generated in a different way than presented in this paper. This implies that neither the execution performance nor the performance of invalidating methods after dynamic deployment was specially optimized.

Also, the way envelopes are generated in the prototype prohibited generation of envelopes for native method. Finally, Reflection was not specially supported and in some cases may not behave as intended after the transformation.

7. Evaluation

This section provides two kinds of evaluation of the envelope-aware Jikes RVM. In section 7.1 the performance of the envelope-aware virtual machine using lazy envelopes and OSR is compared to other AOP implementations with support for dynamic deployment. In section 7.2, we discuss the performance of the alternative imple-

mentations we discussed in the paper, i.e., eager envelopes and code patching.

7.1 Comparison of AOP Implementations

We extended the SPEC JVM98 benchmark suite [42] to deploy an aspect by providing a modified benchmark harness [1]. The aspect deployed by the harness is presented in AspectJ syntax in listing 6. The aspect advises all calls to methods or constructors within any class in any sub-package of `spec.benchmarks`. The advice simply increments a counter.

```
1 aspect Aspect {
2   static long counter;
3   before() :
4       call(* spec.benchmarks..*.*(..)) ||
5       call(* spec.benchmarks..*.new(..)) {
6     counter++;
7   }
8 }
```

Listing 6. Aspect used by the modified SPEC JVM98 benchmark suite to measure the deployment.

The modified benchmark harness executes the SPEC JVM98 benchmark applications and aspect deployment according to the following schema:

1. Perform *n* iterations of the application (*initial-phase*).
2. Deploy the aspect (*deploy*).
3. Perform *n* iterations of the application (*deployed-phase*).
4. Undeploy the aspect (*undeploy*).
5. Perform *n* iterations of the application (*undeployed-phase*).
6. Start over at step 2 for *m* times.

To abstract over the details of aspect deployment in different AOP environments, the extended benchmark harness defines an interface `Deployer` declaring the methods `deploy()` and `undeploy()`, which is implemented for each AOP environment. The number *n* of iterations in each phase and the number *m* of complete cycles are arbitrary. Their influence on the evaluation results will be discussed below.

The extended benchmark tells us different properties of an execution environment with support for dynamic aspect deployment:

- It measures the time needed for the very first run in the initial phase (step 1). During the latter phase class loading and, if applicable to the AOP implementation, preparation of loaded classes takes place. The median of the times for the *n* benchmark iterations in the initial-phase tells us the general performance of the virtual machine when no aspects are deployed. We use the median of the runs to measure the steady-state performance, i.e., to ignore the startup performance.
- The benchmark also tells us how long it takes to deploy and undeploy an aspect (the time needed for steps 2 and 4). We also measure the performance of the first benchmark run after deployment and after undeployment. This reflects the need to re-compile methods after they have been modified by deploying or undeploying an aspect. For the performance values of deployment, undeployment, and startup after deployment and undeployment we use the median of the measured times of all *m* cycles. This is to rule out the initialization efforts for the respective systems.
- The median of the performance figures in the deployed-phase indicates the influence of advice calls on performance. The median of the undeployed-phase figures indicates whether an aspect that was once deployed leaves any footprint after being undeployed.

For the evaluation, the benchmarks are run at the largest size (`-s100`), each in a separate VM instance. The number of cycles (*m*) was three and the number of iterations per phase (*n*) was fifteen which allowed all systems to reach steady state. We ran the benchmarks with 512 MB of heap to rule out the different memory needs of the execution environments.

Tables 1 through 4 show the results of the aforementioned benchmarks for the approaches discussed in section 6 and the envelope-aware virtual machine presented in this paper. Since the implementations extend different Java virtual machines, it is not possible to directly compare all benchmark results. For each approach that is based on a different Java VM, we calculate the relative overhead of executing a benchmark on that approach's extended VM as compared to executing the same benchmark on the approach's unextended base VM. These relative overheads can be used to compare all approaches.

The versions of the benchmarked AOP implementations and their base virtual machines are as follows. AspectWerkz 2.0, JAsCo 0.8.7 and the prototype of envelope-based weaving using Class Redefinition (EBW-Unaware) extend the HotSpot 1.5.0 VM, Steamloom 0.6 is based on Jikes RVM 2.3.1, and the approach described in this paper (EBW-Aware) is based on the Jikes RVM 2.4.1. JAsCo is executed using the suggested `HotSwap 2` variant and having the *inlinecompiler* and the *trapall* flags enabled. The former setting enables a new weaver which provides better performance, the latter setting allows to advise all methods, including `private` methods, as the other AOP environments also do.

The tables do not include results for PROSE implementations. For PROSE1, the Java 5 based implementation, the severely degraded performance in the deployed phase prohibited careful benchmarking. It was not possible to run the benchmarks at their full size. Experiments with smaller sizes revealed an overhead of up to 152,278%. The PROSE2 implementation supports the baseline compiler of Jikes RVM only and is for this reason not competitive. Figures are also missing for the `javac` benchmark on AspectWerkz; the latter crashed when it was started with enabling aspect deployment for the join points in our benchmark aspect. For JAsCo, we lack figures for undeployment and the undeployed-phase because it crashed at undeployment for all benchmark applications.

The goal of the work presented in this paper was to provide a virtual machine which supports fast aspect deployment while providing good steady-state performance. Table 1 shows how many milliseconds are spent for deploying and undeploying the aspect of our benchmark in the above mentioned execution environments for aspect-oriented programs. For every benchmark application our envelope-aware RVM provided the best performance with an average of 3 ms for both, deployment and undeployment. At most, it took 14 ms to deploy the benchmark aspect and 16 ms to undeploy it.

While Steamloom performs well for some benchmark applications, the deployment time goes as high as 4,977 ms and averages out at 941 ms. At undeployment, the measured times were considerably better with an average of 154 ms, though still two orders of magnitude slower than our implementation. JAsCo needs 1,685 ms for deploying the aspect on average. With AspectWerkz the time for undeploying the aspect even goes up to 20,230 ms for the `jess` application and is 195 ms, at least.

Table 2 shows the steady-state performance of the different execution environments. The first row in each segment shows the time

needed to execute the benchmark applications in steady-state on a virtual machine without support for dynamic aspect deployment. For each AOP implementation the overheads in the initial, deployed and undeployed phase are shown. The rows below show the overhead of the AOP implementations extending this virtual machine. The overhead is given as the factor by which the execution time is increased, i.e., an overhead of 1.5 means that execution is 50% slower.

As it was the goal of this work, our implementation implies no overhead to the steady-state performance of an application. JAsCo also provides a good steady-state performance in the initial phase with only 1.7% overhead. For the other approches the overhead in the initial phase reaches from 8.6% (Steamloom) to 121% (AspectWerkz). All approaches, except the one presented here, perform slightly worse in the undeployed phase than in the intitial phase. In AspectWerkz the overhead drops from 121% in the initial phase to 58.5%, which is still significant.

The overhead presented for the deployed phase also contains the overhead for executing the additional advice functionality. Different approaches have different schemes of invoking advice; e.g., some call a static method, while others look-up a receiver object and call a virtual method on it. For this reason, we do not discuss the variations of the overhead in the deployed phase.

AOP implementations may also imply a degradation of the startup performance. As shown in table 3, the envelope-aware Jikes RVM on average performs startup 5.6% slower than the unmodified Jikes RVM. This is mainly due to the `mtrt` benchmark where the overhead is 13.4%. Compared to the other benchmarked approaches, though, where the startup overhead reaches from 8.9% to 215.9% on average, our implementation still provides a reasonable startup performance.

When an aspect is dynamically deployed, affected methods are invalidated and re-compiled the next time they are invoked. As a consequence, during the first run after deployment just-in-time compilation is performed, similar to the very first run. Table 4, shows the performance of the first run after deployment and undeployment as overhead compared to the steady-state of the deployed- or undeployed-phase. Our implementation performs fairly well with an average overhead of 48.9% after deployment and 46% after undeployment. The prototype of envelope-based weaving without virtual machine integration performs best. Steamloom, performing roughly the same as our implementation after deployment, performs better after undeployment. AspectWerkz and JAsCo perform worse than our implementation.

7.2 Comparison of Alternative Optimization Techniques

So far we have shown that with virtual machine support the lazy envelopes approach performs very well compared to other AOP approaches with support for dynamic deployment of aspects. In the following, we compare compare different approaches to VM integration we have experimented with and discussed in this paper. First, we compare eager and lazy envelopes in terms of performance. Further, we also compare the invalidation mechanisms using on-stack-replacement and code patching for envelopes.

The second column of table 5, labeled "No Invalidation Mech, Steady", evaluates the performance of Jikes RVM when eager envelopes are used, but no invalidation mechanism is used; this methodology identifies the overhead introduced by the presence of envelopes separate from the invalidation mechanisms. Our system using envelopes is compared to the unmodified Jikes RVM, and the overhead is reported (higher numbers mean more overhead introduced by envelopes). Performance was measured at *steady-state*, meaning that the program is run long enough for the compilation activity to stabilize.

The steady-state overhead of using envelopes was small, with a max of 3.7% and averaging 0%. In some cases envelopes actually reduced execution time (in particular `db`); this result is not expected, because the compiled code is generally identical in both systems after inlining occurs. However, as with any modifications to a virtual machine, anomalous results are possible [22]. In our system, the baseline compiled code does not have envelopes inlined, and the order of sampling and optimization may be changed slightly; these differences could affect code layout memory locality, which produces unpredictable results. Overall, these results demonstrate that method inlining is effective at reducing the performance penalty of envelopes.

The third column of table 5, labeled "OSR Steady" shows the percentage of runtime overhead (relative to the unmodified Jikes RVM) that is present when eager envelopes are used *and* OSR points are inserted (as described in section 5.1) to ensure that invalidation is performed correctly if dynamic aspect deployment were to occur. There is very little additional overhead introduced by using OSR, demonstrating that it is an effective mechanism for enabling invalidation.

The fourth column of table 5, labeled "Guards Steady" reports the steady-state performance of a system using eager envelopes together with guarded inlining, code patching and splitting to enable invalidation for dynamic weaving, as described in 5.2. The overhead is computed relative to the unmodified Jikes RVM. With the exception of `mpegaudio`, guards performed roughly on par with that of OSR, demonstrating the effectiveness of the splitting and redundant branch elimination algorithms being used. For `mpegaudio`, envelopes with guards result in a 6.9% degradation relative to the original Jikes RVM. After investigating, we discovered that a guard is hindering optimization in a tight loop in mpegaudio; the loop contains a single basic block that is executed frequently, and the existence of a guard in that loop breaks up some of the optimizations performed by Jikes RVM's optimizing compiler. None of the optimizations are fundamentally blocked by the guard, but the loop optimizations in Jikes RVM are not particularly aggressive, and the global optimizations (inter-block) are not as aggressive as the intra-block optimizations.

Despite eager envelopes' minimal impact on steady-state performance, startup performance was more substantially affected. The fourth column, labeled "Startup" shows the performance of eager envelopes with OSR for the first run of the benchmarks with input size `-s100`. The startup overhead ranges from 2.7% to 18.3% with an average of 10.5% degradation relative to the unmodified Jikes RVM.

The startup overhead is caused by the reduced performance of the baseline compiled code. The final column in table 5 shows the overhead caused by eager envelopes in steady state when the optimizing compiler is disabled, thus code is compiled with the baseline compiler only. Because envelopes execute with no optimization in the baseline code when using eager envelopes, the overhead is quite high, ranging from 8.1% – 122.4%. When an application begins executing, all code is first compiled by the baseline compiler, thus short running programs, or "startup" scenarios will be degraded when the quality of the baseline code is reduced. This performance penalty is avoided with lazy envelopes, enabling the lower startup time overhead presented in section 7.1.

8. Conclusions and Future Work

Envelope-based weaving is a technique that facilitates fast weaving of aspects, which is particularly valuable in systems that allow dynamic aspect deployment. In this paper, we have shown that substantial improvement in dynamic deployment performance is possible by making the virtual machine envelope-aware. Well established dynamic optimization techniques can be used to avoid the

	compress	jess	db	javac	mpegaudio	mtrt	jack	avg
EBW-Aware (deploy)	1 ms	4 ms	0 ms	14 ms	2 ms	2 ms	1 ms	3 ms
EBW-Aware (undeploy)	0 ms	2 ms	0 ms	16 ms	1 ms	1 ms	1 ms	3 ms
EBW-Unaware (deploy)	63 ms	538 ms	19 ms	672 ms	109 ms	58 ms	149 ms	229 ms
EBW-Unaware (undeploy)	126 ms	635 ms	103 ms	1152 ms	236 ms	152 ms	380 ms	397 ms
AspectWerkz (deploy)	349 ms	16833 ms	215 ms	–	869 ms	3784 ms	1470 ms	3360 ms
AspectWerkz (undeploy)	442 ms	20230 ms	195 ms	–	864 ms	4247 ms	1476 ms	3922 ms
JAsCo (deploy)	331 ms	3042 ms	251 ms	4772 ms	1228 ms	729 ms	1446 ms	1685 ms
Steamloom (deploy)	102 ms	1211 ms	4 ms	4977 ms	55 ms	195 ms	45 ms	941 ms
Steamloom (undeploy)	3 ms	188 ms	2 ms	805 ms	17 ms	53 ms	16 ms	154 ms

Table 1. Time for deploying and undeploying the benchmark aspect on SPEC JVM98 benchmarks.

	compress	jess	db	javac	mpegaudio	mtrt	jack	avg
Jikes 2.4.1	4919 ms	1633 ms	8925 ms	4025 ms	5268 ms	3247 ms	3511 ms	
EBW-Aware (initial)	0.994	1.053	1.010	1.040	1.005	0.894	0.980	0.997
EBW-Aware (deployed)	1.532	1.241	1.006	1.226	0.933	1.190	0.924	1.150
EBW-Aware (undeployed)	1.029	1.040	1.004	1.045	0.914	1.003	0.915	0.993
HotSpot	5691 ms	1672 ms	9961 ms	3667 ms	5644 ms	1372 ms	2846 ms	
EBW-Unaware (initial)	1.227	1.438	1.014	1.292	1.081	1.033	1.256	1.192
EBW-Unaware (deployed)	1.235	1.472	1.010	1.320	1.118	1.129	1.282	1.223
EBW-Unaware (undeployed)	1.233	1.470	1.019	1.307	1.090	1.098	1.274	1.213
AspectWerkz (initial)	1.384	1.805	1.125	–	1.034	6.854	1.059	2.210
AspectWerkz (deployed)	1.544	1.869	1.124	–	1.094	7.855	1.145	2.272
AspectWerkz (undeployed)	1.052	1.714	1.125	–	1.044	3.453	1.124	1.585
JAsCo (initial)	1.001	1.003	1.016	1.046	0.998	1.019	1.036	1.017
JAsCo (deployed)	1.278	1.231	1.011	1.293	1.103	4.360	1.124	1.628
Jikes 2.3.1	5261 ms	1850 ms	7938 ms	4798 ms	5014 ms	3347 ms	3327 ms	
Steamloom (initial)	1.007	1.076	1.003	1.167	1.203	1.131	1.012	1.086
Steamloom (deployed)	1.235	1.256	1.145	1.221	1.140	1.954	1.032	1.283
Steamloom (undeployed)	1.144	1.277	1.180	1.240	1.434	1.972	1.089	1.334

Table 2. Steady-state performance for the SPEC JVM98 benchmarks.

	compress	jess	db	javac	mpegaudio	mtrt	jack	avg
Jikes 2.4.1	7488 ms	3365 ms	9549 ms	7342 ms	7259 ms	7074 ms	4847 ms	
EBW-Aware	1.058	1.067	0.991	1.033	1.079	1.134	1.029	1.056
HotSpot	5689 ms	2006 ms	10036 ms	4740 ms	5917 ms	1536 ms	3252 ms	
EBW-Unaware	1.237	1.536	1.031	1.476	1.117	1.268	1.350	1.288
AspectWerkz	1.524	5.329	1.152	–	1.333	7.953	1.661	3.159
JAsCo	1.031	1.605	1.012	1.343	1.074	1.195	1.212	1.210
Jikes 2.3.1	6547 ms	3378 ms	8554 ms	7217 ms	6756 ms	6197 ms	4103 ms	
Steamloom	1.052	1.089	0.998	1.119	1.121	1.140	1.101	1.089

Table 3. Startup performance for the SPEC JVM98 benchmarks.

	compress	jess	db	javac	mpegaudio	mtrt	jack	avg
EBW-Aware (startup-deploy)	1.191	1.492	1.034	1.689	1.242	2.734	1.221	1.515
EBW-Aware (startup-undeployment)	1.746	1.552	1.001	1.548	1.173	2.608	1.322	1.564
EBW-Unaware (startup-deploy)	1.011	1.259	1.000	1.359	1.043	1.160	1.125	1.137
EBW-Unaware (startup-undeployment)	1.018	1.304	1.012	1.447	1.081	1.143	1.191	1.171
AspectWerkz (startup-deploy)	1.051	6.514	1.015	–	1.160	1.403	1.526	2.111
AspectWerkz (startup-undeployment)	1.085	8.180	1.013	–	1.163	1.875	1.535	2.475
JAsCo (startup-deploy)	1.049	2.676	1.029	2.627	1.237	1.294	1.624	1.648
Steamloom (startup-deploy)	1.197	1.985	1.006	2.273	1.389	1.533	1.052	1.491
Steamloom (startup-undeployment)	1.222	1.465	0.975	1.437	1.099	1.518	1.006	1.246

Table 4. Startup performance after deployment undeployment of the benchmark aspect.

	No Invalidation Mech, Steady (%)	OSR Steady (%)	Guards Steady (%)	Startup (%)	Baseline Steady (%)
compress	0.0	0.0	-0.1	15.2	122.4
jess	0.7	2.1	3.5	3.3	15.5
db	-4.8	-4.8	-4.8	2.7	8.4
javac	3.7	4.2	5.3	18.3	17.8
mpegaudio	-0.6	-0.2	6.9	13.3	32.7
mtrt	-0.6	-0.1	-1.4	11.3	57.7
jack	1.8	3.0	1.7	8.4	8.1
average	0.0	0.6	1.6	10.5	37.5

Table 5. Percent overhead of using eager envelopes in Jikes RVM. (1.0 is 1%)

performance penalty of envelopes and to efficiently invalidate optimized code after dynamic aspect deployment.

The integration of envelopes into the VM represents an important conceptual contribution to the state-of-the-art of processing aspect-oriented languages. So far, only program elements that are identified by the static or dynamic weaver as join point shadows have the potential to yield join points. With VM integrated envelopes, any expression in the program that yields execution events included in the join point model can potentially turn into a join point at runtime, similar to the notion that all Java or Smalltalk methods are per-default virtual. With an execution environment where dynamic aspect deployment works almost instantly, as in the presented approach, using this new kind of flexibility will become more feasible and natural.

Making the virtual machine envelope-aware is also beneficial for aspect-oriented approaches employing static compilation. When a mapping between envelopes and their enveloped members is created by the compiler and passed to the virtual machine, possibly included in the bytecode, the optimizations presented in this paper can also be applied to envelopes generated before the runtime. In this way, the benefits of improved weaving performance can be combined with unaffected runtime performance.

There are several areas for future work. First, access to the active object from the join point's context in the advice will be supported. In fact, we have a partial implementation, where we adopt an optimization from Jikes RVM's JIT compiler to provide meta data for accessing `this` in each compiled method.

The second area of future work is to provide more sophisticated concepts of aspect-oriented programming. For example, in the current implementation of envelope-based weaving, only method calls and field accesses are supported as join points. AspectJ supports additional kinds of join points, e.g., exception handlers. We will evaluate how the presented mechanisms can be extended to support other join point models.

Further, some AOP approaches also define concepts such as thread-local or object-local dynamic aspect deployment [7, 23, 14, 49, 48, 41, 39, 40]. We will investigate how the VM techniques presented in this paper and others can be be exploited to support such advanced concepts.

Acknowledgments

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

References

[1] Homepage of the aspect-oriented extension of the SPEC JVM98 Benchmarks suite. `http://www.st.informatik.tu-darmstadt.de/DeployBench`.

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

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

[4] Javassist homepage. `http://www.csg.is.titech.ac.jp/~chiba/javassist/`.

[5] abc (AspectBench Compiler) homepage. `http://aspectbench.org/`.

[6] A.-R. Adl-Tabatabai, J. Bharadwaj, D.-Y. Chen, A. Ghuloum, V. Menon, B. Murphy, M. Serrano, and T. Shpeisman. The StarJIT compiler: A dynamic compiler for managed runtime environments. *Intel Technology Journal*, 7(1):19–31, Feb. 2003.

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

[8] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of Java. *ACM SIGPLAN Notices*, 37(11):111–129, Nov. 2002. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[9] M. Arnold and B. G. Ryder. Thin guards: A simple and effective technique for reducing the penalty of dynamic class loading. In *16th European Conference on Object-Oriented Programming (ECOOP)*, volume 2374 of *LNCS*, June 2002.

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

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

[12] BAT homepage. `http://www.st.informatik.tu-darmstadt.de/BAT`.

[13] C. Bockisch, M. Haupt, M. Mezini, and R. Mitschke. Evenelope-based Weaving for Faster Aspect Compilers. In *In Net.ObjectDays*, 2005.

[14] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual Machine Support for Dynamic Join Points. In *AOSD 2004 Proceedings*. ACM Press, 2004.

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

[16] J. Brichau, M. Haupt, N. Leidenfrost, A. Rashid, L. Bergmans, T. Staijen, A. Charfi, C. Bockisch, I. Aracic, V. Gasiunas, K. Ostermann, L. Seinturier, R. Pawlak, M. Südholt, J. Noyé, D. Suvée, M. D'Hondt, P. Ebraert, W. Vanderperren, M. Pinto, L. Fuentes, E. Truyen, A. Moors, M. Bynens, W. Joosen, S. Katz, A. Coyler, H. Hawkins, A. Clement, and O. Spinczyk. Report describing survey of aspect languages and models. Technical Report AOSD-Europe Deliverable D12, AOSD-Europe-VUB-01, Vrije Universiteit Brussel, 17 May 2005 2005.

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

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

[19] D. Detlefs and O. Agesen. Inlining of virtual methods. In *13th*

European Conference on Object-Oriented Programming (ECOOP), volume 1628 of *LNCS*, pages 258–278, June 1999.

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

[21] N. Grcevski, A. Kilstra, K. Stoodley, M. Stoodley, and V. Sundaresan. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In *3rd Virtual Machine Research and Technology Symposium (VM)*, May 2004.

[22] D. Gu, C. Verbrugge, and E. Gagnon. Code layout as a source of noise in JVM performance. In *Component And Middleware Performance workshop, OOPSLA 2004*, 2004.

[23] 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. ACM Press, June 2005.

[24] E. Hilsdale and J. 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.

[25] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI)*, pages 32–43, San Francisco, California, 17–19 June 1992. *SIGPLAN Notices* 27(7), July 1992.

[26] K. Ishizaki, M. Takeuchi, K. Kawachiya, T. Suganuma, O. Gohda, T. Inagaki, A. Koseki, K. Ogata, M. Kawahito, T. Yasue, T. Ogasawara, T. Onodera, H. Komatsu, and T. Nakatani. Effectiveness of cross-platform optimizations for a Java just-in-time compiler. *ACM SIGPLAN Notices*, 38(11):187–204, Nov. 2003.

[27] The Java Reflection API. `http://java.sun.com/j2se/1.4.2/docs/api/index.html`.

[28] The Jikes Research Virtual Machine. `http://jikesrvm.sourceforge.net/`.

[29] Jni (java native interface) homepage. `http://java.sun.com/j2se/1.4.2/docs/guide/jni/`.

[30] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. Lindskov Knudsen, editor, *Proc. ECOOP 2001*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.

[31] T. Lindholm and F. Yellin, editors. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Addison-Wesley, 1999.

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

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

[34] H. Masuhara, G. Kiczales, and C. Dutchyn. A Compilation and Optimization Model for Aspect-Oriented Programs. In G. Hedin, editor, *Proc. CC 2003*, volume 2622 of *LNCS*, pages 46–60. Springer, 2003.

[35] M. Paleczny, C. Vick, and C. Click. The Java Hotspot server compiler. In *Java Virtual Machine Research and Technology Symposium (JVM)*, pages 1–12, Apr. 2001.

[36] A. Popovici, G. Alonso, and T. Gross. Just-In-Time Aspects: Efficient Dynamic Weaving for Java. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 100–109. ACM Press, 2003.

[37] A. Popovici, T. Gross, and G. Alonso. Dynamic Homogenous AOP with PROSE. Technical report, Department of Computer Science, ETH Zürich, Zürich, Switzerland, March 2001.

[38] A. Popovici, T. Gross, and G. Alonso. Dynamic Weaving for Aspect-Oriented Programming. In *AOSD '02: Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, pages 141–147. ACM Press, 2002.

[39] The PROSE Homepage. `http://prose.ethz.ch/Wiki.jsp`.

[40] H. Rajan and K. 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, 2003. ACM Press.

[41] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In *AOSD*, pages 16–25, 2004.

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

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

[44] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. *ACM SIGPLAN Notices*, 36(11):180–195, Nov. 2001. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[45] T. Suganuma, T. Yasue, and T. Nakatani. An empirical study of method in-lining for a Java just-in-time compiler. In *Java Virtual Machine Research and Technology Symposium (JVM)*, pages 91–104, Aug. 2002.

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

[47] W. Vanderperren and D. Suvee. Optimizing jasco dynamic aop through hotswap and jutta. In *Proceedings of the 2004 Dynamic Aspects Workshop*, 2004.

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

[49] W. Vanderperren, D. Suvée, B. Verheecke, M. A. Cibrán, and V. Jonckers. Adaptive programming in jasco. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 75–86, New York, NY, USA, 2005. ACM Press.

[50] A. Vasseur. Dynamic AOP and Runtime Weaving for Java - How does AspectWerkz Address It? `http://aspectwerkz.codehaus.org/downloads/papers/aosd2004-daw-aspectwerkz.pdf`, 2004.

[51] A. Vasseur, J. Bonér, and J. Dahlstedt. JRockit JVM Support for AOP. `http://dev2dev.bea.com/pub/a/2005/08/jvm_aop_1.html`.

[52] Xalan-Java version 2.6.0. `http://xml.apache.org/xalan-j/`.