A Selective, Just-In-Time Aspect Weaver

Yoshiki Sato[1], Shigeru Chiba[1], and Michiaki Tatsubori[2]

[1] Dept. of Mathematical and Computing Sciences
Tokyo Institute of Technology
{yoshiki,chiba}@csg.is.titech.ac.jp
[2] IBM Tokyo Research Laboratory
mich@trl.ibm.com

Abstract. Dynamic AOP (Aspect-Oriented Programming) is receiving growing interests in both the academia and the industry. Since it allows weaving aspects with a program at runtime, it is useful for rapid prototyping and adaptive software. However, the previous implementations of dynamic AOP systems suffered from serious performance penalties. This paper presents our new efficient dynamic AOP system in Java for addressing the underlying problem. This system called Wool is a hybrid of two approaches. When a new aspect is woven in, the programmers can select to reload into the JVM a modified class file in which hooks for executing advice are statically embedded, or they can insert hooks as breakpoints in the JVM. Since the two approaches have different performance characteristics, the programmers can select the best one for each join point. Our experimental result shows, under a certain circumstance, Wool runs dynamic AOP application about 26% faster than a traditional static code translation approach.

1 Introduction

Recently, practical demands are being made of dynamic aspect-oriented programming (AOP [14]) systems [17, 18, 2, 20, 19]. Unlike static AOP, a dynamic AOP system allows dynamically weaving and unweaving an aspect into/from a program. Moreover, advice and pointcuts are changeable during runtime. These dynamic features extend the application domains of aspect-oriented programming. Dynamic AOP can make development cycles shorter [7] and it allows for aspects that can adapt the behavior of application software at runtime to follow the changes of the runtime environment and requirements [11, 25, 21].

The most typical technique for implementing dynamic AOP systems is based on static code translation although it is not efficient. This approach statically inserts pieces of code, which we call hooks, into all join points, and these hooks determine at runtime whether or not there is associated advice to be activated at each join point, in contrast to static AOP systems like AspectJ [13]. These

This work was supported in part by the CREST program of Japan Science and Technology Corp.

runtime checks imply serious performance overhead although they are necessary since dynamic AOP allows turning advice on and off during runtime.

This paper presents our Java-based dynamic AOP system called *Wool*, which exploits our new implementation technique for addressing the performance problem mentioned above. Wool inserts hooks into a program at runtime *just in time* when the programmer directs the program to start using an aspect. Wool allows the programmers to select from two implementation techniques the best one for each join point. The first one is to insert the hooks as breakpoints handled through the debugger interface of the Java virtual machine (JVM). The other one is to produce a program in which the hooks are embedded as method calls and reload that new program into the JVM. These two techniques do not require a custom JVM, but work with the standard JVM.

The rest of this paper is organized as follows. Section 2 describes a typical implementation technique of dynamic AOP systems and a performance problem of that technique. Section 3 presents our new implementation technique for dynamic AOP. It also shows an overview of the current implementation of Wool. Section 4 compares Wool to other AOP systems. Section 5 presents the results of our experiments. We conclude the paper in section 6.

2 Dynamic AOP

Aspect-oriented programming can be classified into two categories: static AOP and dynamic AOP. The static AOP systems such as AspectJ weave in the aspects at compile time or load time. The woven aspects cannot be removed or reconfigured during runtime. On the other hand, the dynamic AOP systems can weave aspects in at runtime. The programmers can dynamically *plug* and *unplug* an aspect in/from running software. This section shows the benefits of dynamic AOP and typical implementations of AOP systems.

2.1 Need for Dynamic AOP

Dynamic AOP is not just a mechanism that sounds fascinating but useless in practice. It is a necessary mechanism especially, if an aspect implements a non-functional concern cutting across several modules and the requirement of the functionality dynamically changes at runtime. Non-functional concerns are additional features such as transactions, distribution, security, and logging. They are not directly involved with the core logic of the application and thus they are not mandatory for the application software to provide the minimum service.

Profiling a performance of software (or logging) is a good example showing that a dynamic AOP system is useful. It is recognized as a non-functional concern that can be well modularized using AOP [7][9]. Since the code fragments for collecting profiling information tend to be spread over the whole program, they should be modularized into an aspect. However, the performance profiling implemented on static AOP systems is not useful from the programmatic viewpoint. Suppose that the software is a Web-based business application, which must run

24 hours a day. Our scenario is that we first run the software without profiling code and, once it shows performance anomaly, perhaps under heavy load, we insert profiling code. The profiling code should be inserted without shutting down the software since the anomaly may be due to the workload up to that point. If the software is restarted, all the internal data structures are reset and hence the information necessary for analyzing the anomaly would be lost. Furthermore, we would need to interactively plug and unplug various kinds of profiling code until solving the anomaly. Each profiling code would cut across different join points for collecting different profiling information. We thus need dynamic AOP. Although we could use large profiling code that collects all the information, it would imply serious performance impacts. We should use minimal profiling code at a time for reducing performance impacts. To satisfy these requirements, dynamic AOP is a good solution.

Adaptable response cache in a web application is also a good example to show the usefulness of dynamic AOP. The implementation of the response cache includes not only caching the results of method calls but also invalidating the cached results that scatter in the software. Since the response cache is a non-functional and crosscutting concern, it cannot be modularized with object-oriented programming; AOP is necessary [21]. However, to make the response cache adaptable, the software must be able to dynamically switch a number of aspects, in which various strategies are modularized, as the runtime environment changes. Yagoub *et al.* reported that there is no universal caching strategy that is optimal for all web applications and all the configurations [26]. For example, if the cache provided by an aspect shows a low hit ratio, the software should switch that aspect to another. If only part of the cache shows a high hit ratio, the software should remove the aspects that do not provide that part of the cache. The traditional object-oriented techniques like Design patterns never modularize such a crosscutting concern, and still less switch it at runtime. Also, static AOP does not even work in this example. If we use static AOP, all the caching aspects must be statically woven in advance. Note that they are woven at different join points and hence, whenever the program execution reaches one of the join points, they must dynamically examine whether every cache is turned on or off. This runtime check causes a serious performance overhead. On the other hand, if we use dynamic AOP, only the activated aspects can be woven to avoid the runtime check. Dynamic AOP enables efficient implementation of adaptable cache.

2.2 The implementation of AOP systems

Typical implementations of object-based AOP systems, including both static and dynamic AOP, insert hooks at a number of execution points such as method calls, field access, instance creation, and exception handling. These execution points are called *join points*. If the program execution reaches join points, the inserted hook intercepts it and executes a piece of code called *advice* if it is included in a set of join points identified by *pointcuts*. Different advice can be associated with each different pointcut. An aspect is a set of pairs of pointcuts and advice.

In most static AOP systems, a hook is usually implemented as inlined hooking code, in which pieces of aspects are directly embedded into a base program by static translations of source code or bytecode. However, several join points cannot be uniquely determined by the pointcuts, such as *cflow* or *this*. Such a set of join points depends on the current execution context and changes dynamically. Thus, hooking code must be embedded into potential join points with conditional statements, which examine if the advice should be executed in the execution context.

Generally, a dynamic AOP system must examine whether any advice should be executed at every join point when the execution of a program passes that point. In dynamic AOP, all the join points are dependent on the execution context, since the set of join points are specified at runtime. Furthermore, the set of join points changes dynamically. Thus, the check whether or not the system should execute advice must continue after the join point has been specified.

2.3 Static code translation

There exists a well-known approach that enables every join point to be checked at runtime, and which is supported by static code translation of application programs. For example, JAC [18] and Handiwrap [2] are dynamic AOP systems using a static code translation approach, in which a compiler (or a translator) inserts minimal hooks for all potential join points (Figure 1). They translate the code of a program to a version with inserted hooks. The translation is performed by the source-to-source or binary-to-binary, during compilation or class loading. Most static AOP systems also use static code translation and this is more or less appropriate to their purpose because most intercepted join points are identified statically.

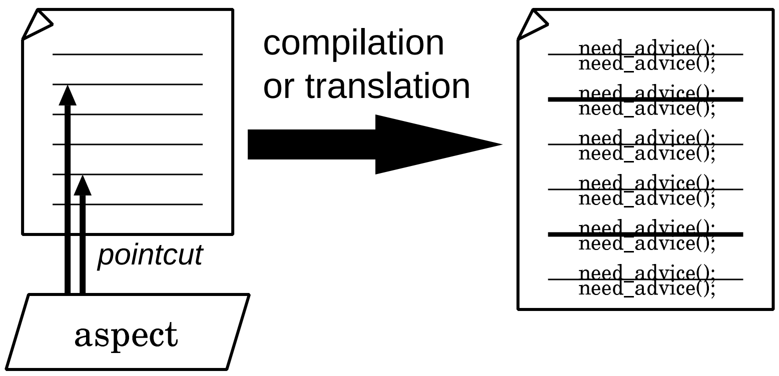

Fig. 1. Static code translation.

Static code translation does not cause much of a performance penalty in advice execution, while involving some overhead in normal operations with no woven aspect. The execution of advice is fast since an inserted hook is represented as just a method call. However, even if no aspects are woven, all checks whether or not the system should execute advice is performed. This results in unnecessary method calls or verbose indirection of object references, which involves overhead in normal operations that cannot be ignored.

Popovici *et al.* [19] have implemented a Just-In-Time (JIT) aspect compiler based on Jikes RVM [1]. Their JIT compiler inserts hooks at all potential join points only at the time of the first just-in-time compilation. Thus, their work can be regarded as a static code translation approach mentioned above. They avoided adding options to the JIT compiler that could recompile bytecode since that would increase the complexity of the JIT compiler support too much. They reported they could limit the overhead due to the hooks since their hooks are implemented using native code, not Java byte code. Unfortunately, the JIT compiler approach is irreconcilable with recent high-performance runtime technologies like Sun's HotSpot(TM) technology or the IBM JIT compiler [23], which involves the mixture of a JIT compiler and interpreter.

3 Wool

We developed Wool, which inserts hooks into the program on demand, in Java. Since the hooks are inserted after all of the intercepted join points are specified, Wool does not insert unnecessary hooks. This section presents the details of our new dynamic AOP system Wool and shows how it enables efficient dynamic AOP.

3.1 An overview of Wool

Wool is implemented as a Java library that provides dynamic AOP functionality, consisting of APIs to write aspects, a weaver to compose aspects with programs, and a subsystem for accepting a request for weaving from the outside of the running program.

Wool allows the aspect to be woven either locally, from within an application running on the same JVM, or remotely when sent to the subsystem of Wool. The following code shows how the aspect is woven in by Wool.

```
WlAspect azpect = WlAspect.forName("ProfileAspect");
Wool wool = Wool.connect("localhost", 5432);
wool.weave(azpect);
```

In a locally woven case, the aspect instance `azpect` is created in the running program. The weaver instance `wool` is connected to the subsystem of Wool. Weaving runs immediately after the method `weave()` is called. Alternatively in a remotely woven case, the aspect instance is actually created and recomposed outside of the JVM in which it will be woven. It is then serialized and sent over the network to the subsystem of Wool in the target JVM.

3.2 Just-in-time hook insertion

Wool adopts a hybrid approach so that the programmers can choose a suitable hook at a join point considering the entire cost, and which hooks are breakpoints or method calls. In Wool, just-in-time hook insertion is done in two timeframes at runtime, as shown in Figure 2.

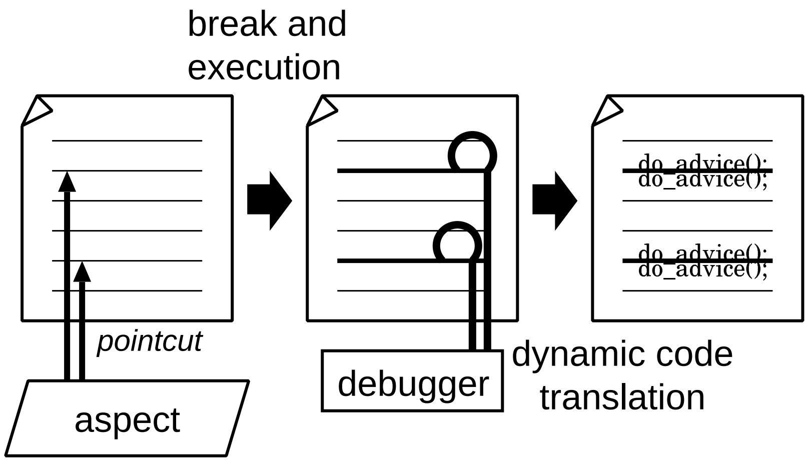

Fig. 2. Two timeframes for hook insertion.

The strategy for deciding at the hooked join point whether advice is executed or embedded into the program is simple. All of the hooks are represented as breakpoints first. At each hooked join point, there are alternative ways, one is executing pieces of advice by using the debugger and the other is embedding hooks into the program using dynamic code translation. If the hooked join point using a breakpoint is judged likely to be intercepted again and again in the future, and if the degradation it causes is estimated to be higher than that caused by dynamic code translation, such a hook should be embedded into the program instead of executing the advice by using the debugger. After the hook is embedded, the breakpoint at the join point is removed.

A comparison of the implementation techniques of dynamic AOP systems is shown in Table 1. Wool is a hybrid of the last two techniques. Unlike static code translation, both of the two techniques that Wool adopts do not insert any unnecessary hooks (Column 1 in Table 1).

As a breakpoint The first hook insertion method, which we call breakpoint-based execution, where all the hooks are inserted as breakpoints, which are set at runtime through standard debugger interface in Java called JPDA (Java Platform Debugger Architecture) [24]. The JPDA allows a programmer to register requests for execution events inside a JVM and controls execution for each event notification. These breakpoints are set for all join points specified by a pointcut. If the thread of control reaches one of the breakpoints, it switches to the debugger thread and the advice associated with that join point (breakpoint) is run. Using JPDA doesn't require the modification of the runtime system.

The execution overhead due to breakpoint is not a serious problem since the HotSpot (TM) VM that comes with the Java 2 SDK 1.4 runs a program together with a just-in-time compiler even if any breakpoints are set. In addition, hooks in the form of breakpoints can be inserted into programs so quickly (Column 3 in Table 1). Although programs must be run in a debug mode, it doesn't cause much performance penalty under normal operations without active advice (Column 4 in Table 1).

For frequently executed advice, the overheads for breakpoint-based execution are not negligible (Column 1 in Table 1). The large number of context switches to execute the advice causes the overhead, since advice has to be executed separately in the debugger process.

As a method call The second hook insertion method, which we call dynamic code translation: To reduce the overhead caused by context switches, a frequently invoked join point expressed as a breakpoint is replaced with a modified method in which the hooks are directly embedded. The method body is modified at the bytecode level so that a bytecode sequence for executing the advice is embedded at the join points contained in the method body. At the breakpoint, all join points specified by the pointcut are identified, so hooks can be statically embedded into the programs without garbage (unnecessary) hooks as in other static-code-translation-based dynamic AOP systems.

The runtime replacement of bytecode is done using the hotswap mechanism [8] of the JPDA. The hotswap mechanism allows a new class to be reloaded at runtime while under the control of a debugger. The actual reloading isn't performed immediately when the static code translation is completed, because the cost of such a translation is very large. If there is a method that should be replaced with a hook embedded method, dynamic code translation is forked, the breakpoint-based execution continues until the translation is finished. Therefore, the dynamic code translation stops the application thread for a short time and uses the translation time effectively. After replacing the method, the thread of control does not stop at the join points contained in the method body. The hooks are embedded into the program as simple method calls, and therefore the advice execution is much faster than using the debugger (Column 1 in Table 1).

Dynamic code translation is not efficient under certain circumstances. It causes only a single context switch to embed hooks into the program. However, the cost of the translation and the hotswap performed for every crosscut class is relatively high if advice is rarely executed (Column 2 and 3 in Table 1). In this case, dynamic code translation is just unnecessary as most of hooks are in the static code translation approach in Section 2.3.

Table 1. Comparison of the three approaches. Wool is a hybrid of the last two techniques, which are using breakpoint-based execution and dynamic code translation. Each column indicates the degree of the efficiency of using that approach.

	frequently executed advice	rarely executed advice	hook insertion	normal operation without aspect
Static code translation	○	×	○ (statically)	×
Breakpoint-based execution	×	○	△	○
Dynamic code translation	○	×	×	○

3.3 Aspect in Wool

Wool provides the programmers with APIs to define an aspect in Java. It does not provide a special aspect language for easily writing an aspect, which is different from the languages such as AspectJ or any others that are intended to enhance flexibility and abstraction. Using these APIs, an aspect can be instantiated in the Java program. Therefore, the aspect can be composed and changed by a program dynamically. This means that pointcuts and advice can be reconstructed while the target program runs.

The following fragment of a program is a sample profiling aspect described in Java with Wool APIs:

```
public class ProfileAspect extends WlAspect {
  Timer timer = new Timer();
  int count = 0;
  Pointcut timedentry = Pointcut.methodCall("public","FigureElement","paint","*");
  public void weave(Wool wool) throws WeaveException {
    wool.insert(new BeforeAdvice(timedentry) {
      public void advice(Joinpoint joinpoint) {
        timer.start();
        count++;
      }
     });
    wool.insert(new AfterAdvice(timedentry) {
      public void advice(Joinpoint joinpoint) {
        timer.stop();
      }
     });
  }
}
```

Here, the class `ProfileAspect` inherited from `WlAspect` is used for profiling the bottleneck of a program. In particular, the above example is specified by the `Pointcut` object for profiling the method call that belongs to the class `FigureElement` and named `paint` and that has a `public` modifier. This aspect inserts *before* advice and *after* advice to measure the elapsed time and the number of the method calls. `BeforeAdvice` and `AfterAdvice` represent *before* and *after* advice, respectively. Advice in Wool is inserted by using the method `insert()` in the class of `Wool`.

Aspect The first step in the use of Wool is to create a `WlAspect` object representing an aspect defined by programmers. This step is for creating the aspect and makes it accessible from a program. In an aspect of Wool, the programmers can define it in the following two ways:

– Define the subclass of `WlAspect`, or
– Add advice to the scratch object of `WlAspect`.

The subclass of `WlAspect` represents an encapsulation of crosscutting concerns. Programmers can define aspect variables in it, which are accessed from advice or aspect methods or introductions such as `timer` and `count` shown in the above example. In addition, it contains initial weaving advice described in the method `weave()`, inherited from `WlAspect`. It is called on the return from the weaver at the time an aspect is actually woven into a program. It is only by using the method `weave()` that the programmers can insert advice into a program in the subclass of `WlAspect`.

To construct and reconstruct an aspect object dynamically, Wool provides another way to create it from scratch. This feature is useful because an aspect, which is the intercepted join point identified by the pointcut or the operation defined by advice, can be formed according to the behavior of the running program. To do this, a `WlAspect` object must be created as follows:

```
WlAspect azpect = WlAspect.scratchAspect();
azpect.add(new BeforeAdvice(log) {
    public void advice(Joinpoint joinpoint) {
      /* some code */
    }
  });
```

The created object `azpect` represents an empty aspect that has no advice or introductions although the method `add()` adds advice to the aspect later. If new advice is added to a non-empty aspect like the class `ProfileAspect`, advice inserted in the method `weave()` is left as it is, and the new advice is just added as extra advice.

The added advice is not immediately reflected in the program. In Wool, advice is synchronized with the program only by the method `weave()` or `unweave()`. Thus, the behavior of a running program is changed only when those methods are called.

Pointcut Wool provides several methods for identifying the set of join points by using the `Pointcut` class. The `Pointcut` class has some static methods to identify a set of join points and some methods to be used for some logical operations. For example, the method `methodCall()` identifies a call to the method with four `String` arguments. Those arguments are used for indicating a modifier, a method name, a declared class, and a signature. Table 2 lists several methods in `Pointcut`.

Table 2. Methods in `Pointcut` for identifying a set of join points.

`static Pointcut methodCall(String, String, String, String)`	identify a call to the method.
`static Pointcut methodExecute(String, String, String, String)`	identify an execution of the method.
`static Pointcut fieldGet(String, String, String)`	identify a read of the field.
`static Pointcut fieldSet(String, String, String)`	identify a write of the field.
`static Pointcut instanceCreate(String, String, String)`	identify a creation of the instance.
`static Pointcut exceptionHandle(String)`	identify a handling of the exception.
`static Pointcut within(String)`	identify any join point defined in the class.
`Pointcut and(Pointcut)`	perform an AND operation.
`Pointcut or(Pointcut)`	perform an OR operation.

Advice Wool provides methods for inserting a piece of code called advice into a program by the `WlAdvice` class and its subclasses, such as `BeforeAdvice` or `AfterAdvice`. Advice consists of a pointcut and an advice body. The constructor of the `WlAdvice` class takes as a parameter a `Pointcut` object to identify the join point. In addition, an advice body is described in the method `advice()` declared in the class inherited from the class `WlAdvice`. The object of `WlAdvice` is inserted into a program in the method `weave()` on the `WlAspect` class through the object of `Wool`.

If advice is defined as an anonymous class like a closure:

```
public void weave(Wool wool) throws WeaveException {
  wool.insert(new BeforeAdvice(log) {
    public void advice(Joinpoint joinpoint) {
      /* can access the external variables */
    }
  });
}
```

The code in the method `advice()` can access external variables. Consequently, the scope of the aspect can be made naturally because aspect variables can be accessed from an anonymous class inherited from `WlAspect`. Moreover, the advice is easily changed and modified at runtime.

A parameter of the method `advice()`, the object of `Joinpoint`, contains reflective information about the current join point for the advice to use. It is similar to *thisJoinPoint* of AspectJ. Mainly, this object is used to obtain certain dynamic information such as the currently executing object or the target object or the arguments. The current version of Wool doesn't support obtaining more reflective information such as data structures of the class for the sake of efficiency. However, such an optimization technique as partial evaluation [15] offers the possibility of efficiently providing rich reflective information for programs, since it can statically pack that information only into the advice that requires them.

Introduction Although the limitations of the JPDA prevent Wool from implementing an introduction directly, it is easy to implement it indirectly. When a class is replaced with a new one, the JPDA restricts the new one to changing the schema like fields and the hierarchy like subclasses or the interfaces and class modifiers and method modifiers, and to deleting methods. Thus, the introduction itself is restricted with the JPDA. However, the introduced method or field is actually referred to only from the advice code. Therefore, by adding a hidden map or a list for the introduction to all of the classes at load-time, then making the advice code use the hidden variable, Wool can allow for the addition of class elements.

3.4 Control of the weaver

Wool provides an optional function for programmers to control the behavior of a weaver. This function operates at the time when an aspect actually weaves the program, in other words, when the effect of an aspect appears in the running program. In dynamic AOP systems, the timing of the weaving is important because there is a non-determinacy when an aspect is woven from a remote JVM and there is a necessity to care for a paired advice in relation to the activation frames.

This function is implemented by delegating methods related to the weaving operation from Wool to the programmer. A programmer can control Wool by overriding the methods of `WlAspect`, specifically `hook()` and `initWeave()`. The object of `Wool` is passed to the programmer through those two methods as a parameter. Thus, by implementing the weaving operation by hand with several provided methods, the programmer can control Wool and take care of paired advice using dynamic information. Again, the programmer can select the method of hook insertion as described below in detail. Table 3 lists the available methods through the object of `Wool`.

3.5 Implementation of just-in-time hook insertion

We present the implementation issues of just-in-time hook insertion by describing the details of weaving step-by-step. The order of the weaving process in Wool is:

(1) Scan classes.
(2) Insert hooks as breakpoints.
(3) The programmer selects the most suitable method.

Table 3. Available methods in `Wool` for the control of Wool.

`void advice(Joinpoint)`
execute advice associated with the join point.

`void embedHook(Joinpoint, Pointcut or String,` *optional* `boolean)`
embed hooks into the program by using dynamic code translation. Second optional parameter triggers undocking the translation thread.

`int countActivationFrame(String)`
count the number of activation frames in the context the intercepted program is running in.

`void filterClass(String, boolean)`
restrict the loaded classes to be effected by an aspect.

(4)-1 Execute using the debugger, or
(4)-2 Embed the hook and call the advice.

Following are the details of each step.

Scan classes After Wool is attached to the target program, the application threads except for the threads like the garbage collection and JIT compiler threads are suspended for a while. Wool scans all of the loaded classes and finds out the join points specified by any pointcut. The method `initWeave()` is called just before this scan. For example, if some classes are filtered by the method `filtering()` in `initWeave()` as follows:

```
public class ProfileAspect {
  public void initWeave(Wool wool) throws WoolException {
    wool.filterClass("^java.*|^sun.*", false);
  }
```

those classes are excluded from the scanning. then

Insert hooks as breakpoints Wool sets breakpoints to specify each join point in a set of filtered classes. In order to set the breakpoint, Wool use the subclasses of `Hook` (`CallHook`, `GetHook`, etc.) included in the `wool.hook` package that is implemented using the class `BreakpointRequest` in JPDA. At the same time, pieces of any advice represented as a closure is associated with the join point through the objects of `Hook`. Finally, all of the threads that Wool has suspended are resumed.

The programmer selects the most suitable method When any thread of the target program reaches the first join point, it is intercepted by Wool. Wool calls the method `hook()`. A programmer can avoid the executing advice that join point for the paired advice by overriding the method `hook()` in the subclass of `WlAspect`. Wool gives programmers dynamic information about the join point through the object of the functions using `Joinpoint`(`CallJoinpoint`, `GetJoinpoint`, etc.) included in the `wool.joinpoint` package, which are all implemented using the class `BreakpointEvent`. At the same time, the programmer can select whether to activate dynamic code translation by the method `embedHook()` or to execute the advice by the method `advice()`:

```
Pointcut p
  = Pointcut.methodCall("public","FigureElement","paint","*");
public void hook(Wool wool, Joinpoint joinpoint)
  throws WoolException {
  if (wool.countActivationFrame("main") > 0)
    wool.advice(joinpoint);          // breakpoint-based execution
  else
    wool.embedHook(joinpoint, p); // dynamic code translation
}
```

This fragment of a program means that if there is no activation frame at the join point on the thread named `main`, the advice associated with the joinpoint `joinpoint` is activated. Otherwise, dynamic code translation is performed. The method `embedHook()` takes the object of `Pointcut` or the name of the class as a parameter.

Execute using the debugger There are two cases when the debugger executes advice. One is that the method `hook()` is not overridden, which is the default case. The other is that the method `advice()` is called in an overriding `hook()`. Just by calling the method `advice()`, the appropriate advice associated with that join point is executed.

Embed the hook and call the advice When the method `embedHook()` is called, Wool creates a hook for the class to be installed using Javassist [4], which is a load-time bytecode modification tool, and calls the method `redefineClass()`, which is declared in the class `VirtualMachine` in JPDA, to replace it with the new one. During the translation and replacement, the intercepted program allowed to resume execution. The advice is executed by the debugger substituting the advice as required until the replacement is completed.

Once dynamic code translation has been executed, the control of Wool will not return to the aspect program for the sake of efficiency. Not to adopt dynamic code translation or to insert hooks per thread it is better to continue breakpoint-based execution because hooks can be embedded anytime under the control of Wool.

3.6 Taking care of activation frames

Using just-in-time hook insertion, there is an exceptional case that we have to treat in a special way when substituting a method in which hooks are embedded.

This is when the execution of some advice involves a join point contained in the method currently being executed. For example, suppose that a draw method in a Rectangle class is currently being executed and the activation frame associated with that method is on the execution stack. After the class file of Rectangle is reloaded with the hotswap mechanism, however, the execution of the draw method with that activation frame on the stack is still being performed according to the definition of the draw method given by the old class file. Thus, the hooks contained in the new class file are not effective for that execution. The hooks are effective only for the execution of the draw method started after the reloading. However, the draw method might recursively call itself after the class file. To avoid this problem, dynamic code translation is automatically delayed, instead breakpoint-based execution is performed on the activation frame until the activation frame is popped from the stack.

We also have to be careful with the execution of a pair consisting of before and after advice woven at the same join point. If that pair is woven accidentally while the method containing that join point is executed, only the after advice will be executed at the end of that execution. The before advice will not be executed since the method execution had already been started. This behavior might cause a problem if the after advice depends on the results of the before advice. For example, the before advice might record the current time and the after advice can use that value to compute the elapsed time. In this case, after advice must not be executed if the corresponding before advice was not executed. To solve this problem, our technique allows the programmers to select the behavior in that case using the dynamic information at the join point.

4 Related Work

In this section, we discuss some AOP implementations related to our work, and compare them to Wool. Most current AOP implementations are based on code translation performed by a preprocessor at compile-time or by an extended classloader at load-time of the classes. Two extreme dynamic AOP systems have already proposed exceptions to static code translation, where hooks consist of all-breakpoints or all-methodcalls. Both of these systems have drawbacks in their program execution performance. Wool can avoid these performance penalties by taking a suitable approach for each join point according to the programmer's specification.

An earlier version of AspectJ [13] pre-processes the source code of the aspects and produces a base Java program used to generate a pure Java program that includes woven aspects within it. Even though it only supports static AOP, AspectJ is a typical compiler-based AOP system. Since it is a static AOP system, whether to weave the advice at a join point is determined at compilation time. Also, the advice activity never changes during the runtime in AspectJ. This is sometimes a problem for faster development cycles [7] and for adaptable aspects [11, 25, 21].

Several researchers have addressed the problem of compile-time weaving by shifting the timing of aspect weaving to later stages. Approaches using bytecode-modification tools such as BCA [12] and Javassist [4] use a customized Java class loader to allow weaving at load-time. Extensions of a just-in-time (JIT) compiler like OpenJIT [16] allow weaving at the time of dynamic compilation by the JIT compiler. These are useful for faster development cycles. With these approaches, however, the chance of composition of an aspect with a program is restricted to only one time, at load-time or at dynamic compilation. In order to allow the dynamic activity of advice code, we need some tricks like runtime class evolution [10] to decompose the aspects from a program. We employed the hotswap mechanism of the JPDA for that in Wool.

PROSE [20] uses the JVM debugger interface called JPDA to insert a hook as a breakpoint, which is same as Wool when it inserts only hooks as breakpoints. They report that the execution of advice is too slow in their system to be acceptable. However, we think this approach is useful in limited cases. For example, when a system administrator must recover from system failure as soon as possible, a lightweight diagnosis aspect could be helpful. Meanwhile, when Wool inserts all of the hooks as method calls, this is the same as our previous work [6]. Our experiment has shown that dynamic code translation and class hotswapping impose heavy costs in execution time. However, dynamic compilation may amortize such costs in the long term.

5 Experimental Results

This section first shows the result of our preliminary experiments validating the fundamental of Wool approach basing in a debug mode and combining two hooking means in Wool. After that, it reports the result of our application benchmark which compares Wool to other implementation approaches to dynamic AOP systems. We performed all the experiments on the Sun Java 2 SDK v1.4.0 HotSpot^{TM} Client VM / Solaris8 / Sun Blade1000 (CPU:UltraSPARC-III dual 750MHz, RAM:1GB).

5.1 Preliminary Experiment

Debug mode Wool forces application programs run in the debug mode but it is not a major problem with Java 2 SDK 1.4. Although [20] reported that this overhead is too large to use the JPDA for implementing a dynamic AOP system, this overhead has been significantly reduced by using Java 2 SDK 1.4. We measured the overhead incurred by a debug mode to show that Wool adopts a realistic method. Table 4 summarizes the relative execution time of the SPECjvm98 [22] benchmarks in the debug mode of Sun Java 2 SDK 1.4. The observed performance loss is less than 5%.

Two kinds of hooks To demonstrate the differences of the two kinds of hooks, the breakpoint and the method call, we compared the performance of a join

Table 4. The overhead for SPECjvm98 in the debug mode of Sun Java2 SDK 1.4.

Benchmark	overhead
_200_check	103.52 %
_201_compress	99.18 %
_202_jess	104.64 %
_209_db	101.54 %
_213_javac	100.82 %
_222_mpegaudio	101.33 %

point hooked by a breakpoint with the same one using a method call, both using Wool. In these measurements, the join point was an empty method call, and the advice was empty. These measurements involved 10,000 iterations. The results of these micro-measurements are shown in Table 5.

Breakpoint hooking takes approximately 700 times longer than method-call hooking on average. The elapsed time for breakpoint hooking varies widely depending on the implementation of the process scheduler used in the experimental environment because a breakpoint must be intercepted by a debugger process. Consequently, once a hook is inserted as a method call, it brings about a large performance improvement. The average time of the hook as a method call shown in Table 5 does not include the time elapsed during dynamic code translation in order to measure the pure elapsed time for the hook as a method call.

Table 5. Hooks as breakpoints and method calls in Wool.

Measurement	Average	Minimum	Maximum	Hook insertion
breakpoint	9.956[ms]	9[ms]	103[ms]	
method call	14.3[us]			435[ms]

5.2 Wool measurements

To demonstrate the effectiveness of the proposed just-in-time hook insertion, we compared the overhead of Wool with other techniques. We picked the `jess` benchmark program from the SPECjvm98 benchmarks and measured the execution time of the program with one of the input data called monkey banana. The `jess` benchmark is the Java Expert Shell System based on NASA's CLIPS expert shell system, which has over 10,000 lines of code and 140 classes.

We provided a before advice code which does nothing and let it woven into all the `public` method bodies in the `jess` program. The methods woven an advice code exists 163 and totally called 87,457 times. For comparison, we measured the execution time of the program with the advice woven varying the underlying systems to the one with static code translation stated in Section 2.3, the one only with dynamic code translation, the one only with breakpoint-based hooks, and Wool. For making use of Wool's hybrid approach, we implemented a simple profiler using Wool APIs as follows:

```
public void hook(Wool wool, Joinpoint joinpoint)
  throws WoolException {
  wool.advice(joinpoint);
  Class clazz =
    ((ExecutionJoinpoint)joinpoint).method().declaringType();
  if (map.increment(clazz) > 100)
    wool.embedHook(joinpoint, clazz.getName());
}
```

This means that if a class is being frequently intercepted, the hooks are embedded into this class dynamically. Using this simple profiler and adjusting the threshold, the method for hook insertion was automatically and suitably selected without requiring in-depth knowledge of the application.

Table 6. Elapsed Time [ms] of `jess`. The results in AspectJ is 1013 ms just for reference.

	Static code translation	Dynamic code translation	Breakpoint-based execution	Wool
pointcut	0	2,428	2,428	2,428
hook insertion	0	3,553	0	1,196
execution	10,938	4,077	398,286	4,514
elapsed time	10,938	10,058	400,714	8,138

Table 7. The numbers of translated classes, inserted hooks, and pointcut test. The numbers in parenthesis represents the comparison to AspectJ.

	Static code translation	Dynamic code translation	Breakpoint-based execution	Wool
translated classes	149 (196%)	76 (100%)	0	15 (20%)
inserted hooks	2,815 (1727%)	163 (100%)	163	163
execution times	1,077,338 (1231%)	87,457 (100%)	87,457	87,457

Table 6 lists the results of the benchmark execution. The total time consists of the pointcut time (elapsed time for scanning classes), the hook insertion time (elapsed time for runtime code translation and hotswapping) and the execution time (the rest of the elapsed time). Table 7 lists the numbers of translated classes, inserted hooks, and executed pointcut tests, for each hook implementation approach. These results show that Wool ran dynamic AOP application about 26% faster than a dynamic AOP system using static code translation approach. This is because Wool avoided inserting unnecessary hooks. The static code translation inserted hooks into the program 17 times as many as Wool, and thus resulted in 12 times more pointcut tests. Moreover, Wool was about 19% faster than dynamic code translation and about 98% faster than breakpoint-based execution. This is because Wool allowed switching the breakpoint and method call implementations at every join point. The results for dynamic code translation show that compiling extra 61 (76 - 15) classes did not improve the performance against Wool. The compilation cost 2357 (3553 - 1196) msec. whereas the execution time was reduced by only 437 (4514 - 4077) msec. The breakpoint-based execution caused very large performance degradations because of over 87,000 context switches.

According to Table 6, Wool was 4 times slower than the ideal result using AspectJ with respect to the pure execution time excluding time for pointcut and hook insertion. This is mainly because Wool reifies runtime contexts at every join points whereas AspectJ does not unless reifying is explicitly required. In fact, another experiment by us showed that the execution performance of AspectJ was 25-30% slower if advice includes a special variable named $thisJoinPoint$ and thus AspectJ reifies part of runtime contexts. Another reason is that hooks are not specialized for the type of each join point. Therefore, the hook code is indirectly invoked at a join point and the context there is indirectly accessed from advice. The overhead by Wool could be reduced if we employ the techniques proposed in [5] and [3].

6 Conclusion

This paper presented a new dynamic aspect weaver called Wool, which makes it possible to implement efficient dynamic AOP systems. Wool is implemented in Java without modifying the existing runtime system. It integrates a technique using breakpoints provided by the debugger interface of the JVM and a technique using the hotswap mechanism, which allows us to reload a class file that has already been loaded. This selective functionality is delegated to programmers with dynamic information about the target program. Furthermore, it provides a framework for taking care of activation frames by controlling the timing of the aspect weaving.

Our experiment showed Wool runs dynamic AOP application about 26% faster than a dynamic AOP system using static code translation approach under a certain circumstance. This is because Wool avoids inserting unnecessary hooks. Moreover, the experiment showed. Wool is about 19% faster than dynamic code translation, and about 98% faster than breakpoint-based execution. This is because Wool allows programmers to select the most suitable hooking means at each joinpoint from breakpoint or method call implementation.

Our first version of Wool requires the programmers to make decisions about the hooks. This manual selection has a high probability of producing good results. However, sometimes the programmer does not know the best combination of hooks as breakpoints and as method calls. In the future, we will implement a sophisticated profiler like that of the HotSpot VM to automatically select the most appropriate hooks.

Acknowledgement

We would like to express our deep gratitude to our shepherd, the anonymous reviewers, and the program co-chairs. Their valuable suggestions and comments helped us revise this paper. We also thank Shannon Jacobs for his great efforts to fix numerous English problems in this paper.

References

1. Alpern, B., Attanasio, C.R., Barton, J.J., Burke, M.G., Cheng, P., Choi, J.D., Cocchi, A., Fink, S.J., Grove, D., Hind, M., Hummel, S.F., Lieber, D., Litvinov, V., Mergen, M.F., Ngo, T., Russell, J.R., Sarkar, V., Serrano, M.J., Shepherd, J.C., Smith, S.E., Sreedhar, V.C., Srinivasan, H., Whaley, J.: The Jalapeno virtual machine. IBM System Journal **39** (2000) 211–238
2. Baker, J., Hsieh, W.: Runtime Aspect Weaving Through Metaprogramming. In: AOSD 2002. (2002) 86–95
3. Braux, M., Noyé, J.: Towards Partially Evaluating Reflection in Java. Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '00) (2000)
4. Chiba, S.: Load-time structural reflection in Java. In: ECOOP 2000. LNCS 1850, Springer-Verlag (2000) 313–336
5. Chiba, S., Nishizawa, M.: An Easy-to-use but Efficient Java Bytecode Translator. In: Second International Conference on Generative Programming and Component Engineering (GPCE'03), Erfurt Germany (2003)
6. Chiba, S., Sato, Y., Tatsubori, M.: Using HotSwap for Implementing Dynamic AOP Systems. 1st Workshop on Advancing the State-of-the-Art in Run-time Inspection, july, 2003, Darmstadt, Germany held in conjuction with ECOOP 2003 (2003)
7. Davies, J., Huismans, N., Slaney, R., Whiting, S., Webster, M., Berry, R.: Aspect oriented profiler. In: 2nd International Conference on Aspect-Oriented Software Development. (2003)
8. Dmitriev, M.: Towards flexible and safe technology for runtime evolution of java language applications. In: In Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001 International Conference, Tampa Bay, Florida, USA (2001) 14–18
9. Easy Software Foundation: ajProfiler - easy java profiler. http://http://ajprofiler.sourceforge.net/ (2002)
10. Evans, H., Dickman, P.: Zones, contracts and absorbing changes: An approach to software evolution. In: Proceedings of OOPSLA'99, Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications. Number 10 in SIGPLAN Notices vol.34, Denver, Colorado, USA, ACM (1999) 415–434
11. Joergensen, B.N., Truyen, E., Matthijs, F., Joosen, W.: Customization of Object Request Brokers by Application Specific Policies. In: Middleware 2000 conference. (2000)
12. Keller, R., Hëlzle., U.: Binary component adaptation. In: ECOOP'98 - Object-Oriented Programming. LNCS 1445, Springer-Verlag (1998) 307–329
13. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: ECOOP 2001. LNCS 2072, Springer-Verlag (2001) 327–353

14. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Proceedings European Conference on Object-Oriented Programming. Volume 1241. Springer-Verlag, Berlin, Heidelberg, and New York (1997) 220–242
15. Masuhara, H., Kiczales, G., Dutchyn, C.: A compilation and optimization model for aspect-oriented programs. In: Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings. Volume 2622 of Lecture Notes in Computer Science., Springer (2003) 46–60
16. Ogawa, H., Shimura, K., Matsuoka, S., Maruyama, F., Sohda, Y., Kimura, Y.: OpenJIT frontend system: an implementation of the reflective JIT compiler frontend. In: ECOOP 2000. LNCS 1850, Springer-Verlag (2000)
17. Orleans, D., Lieberherr, K.: DJ: Dynamic adaptive programming in Java. In: In Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns. LNCS 2192, Springer-Verlag (2000) 73–80
18. Pawlak, R., Seinturier, L., Duchien, L., Florin, G.: JAC: A flexible framework for AOP in Java. In: Reflection 2001. (2001) 1–24
19. Popovici, A., Alonso, G., Gross, T.: Just in Time Aspects: Efficient Dynamic Weaving for Java. In: 2nd International Conference on Aspect-Oriented Software Development. (2003)
20. Popovici, A., Gross, T., Alonso, G.: Dynamic Weaving for Aspect-Orinented Programming. In: AOSD 2002. (2002) 141–147
21. Segura-Devillechaise, M., Jean-Marc Menaud, G.M., Lawall, J.L.: Web Cache Prefetching as an Aspect: Towards a Dynamic-Weaving Based Solution. In: 2nd International Conference on Aspect-Oriented Software Development. (2003)
22. Spec - The Standard Performance Evaluation Corporation: SPECjvm98. http://www.spec.org/osg/jvm98/ (1998)
23. Suganuma, T., Ogasawara, T., Takeuchi, M., Yasue, T., Kawahito, M., Ishizaki, K., Komatsu, H., , Nakatani, T.: Overview of the IBM Java just-in-time compiler. IBM Systems Journals **39** (2000) 175–193
24. Sun Microsystems: Java^{TM} platform debugger architecture. http://java.sun.com/j2se/1.4/docs/guide/jpda/index.html (2001)
25. Truyen, E., Jrgensen, B.N., Joosen, W.: Customization of component-based object request brokers through dynamic configuration. In: Technology of Object-Oriented Languages and Systems. (2000)
26. Yagoub, K., Florescu, D., Issarny, V., Valduriez, P.: Caching Strategies for Data-Intensive Web Sites. In: In Proceedings of the 24th International Conference on Very Large Databases (VLDB), Cairo Egypt (2000)