効率的なJava Dynamic AOPシステムを実現するJust-in-Time Weaver

佐藤芳樹[†] 千葉滋[†]

アスペクト指向プログラミング（AOP）とは，分散処理，同期処理，ロギングなどのモジュール間にまたがる関心事（crosscutting concerns）を，アスペクトという概念を用いて別にモジュール化し，記述するプログラミング手法である．特に近年，実行時にアスペクトをプログラムに織り込む（weave）Dynamic AOP システムが注目されている．Dynamic AOP システムでは，開発性の向上や，アスペクトを使用して実行時情報に基づき挙動を変えるプログラムを記述できるといった利点がある．本稿では，Dynamic AOP システムの効率的な実現のために，ジャストインタイム・フック埋め込み技術を提案し，それを実装した weaver である Wool について論じる．Java における既存の Dynamic AOP システムの実現手法には，独自 VM の実装，デバッガの利用，すべての join-point へ静的にフックを挿入する手法など様々な手法があった．しかし，そのどれもがプログラムの実行に，非常に大きな性能劣化を引き起こしていた．本稿で提案する手法は，実行性能を改善するために Sun JDK1.4 の JPDA が提供する実行時クラス再定義を利用している．この手法では，プログラムの実行がアスペクトに影響されるクラス中の join-point へ最初に達したときのみデバッガにより実行が捕捉される．そして，同時にその join-point へフックを静的に埋め込むため，以降の実行での性能を劣化させない．また，アスペクトが影響しないクラスが join-point のために改変されることはないため，その実行オーバヘッドもない．

Just-in-Time Weaver for Efficient Java Dynamic AOP Systems

Yoshiki Sato[†] and Shigeru Chiba[†]

An aspect-oriented programming (AOP) is a programming technique for modularizing crosscutting concerns, cutting across modules such as distribution, synchronization, and logging. The modularized crosscutting concern is called an aspect. Recently, there has been growing interests in dynamic AOP systems which can weave an aspect in a program at runtime. Dynamic AOP systems have several advantages that promote rapid prototyping and enable us to describe adaptive programs based on runtime information. In this paper, we propose just-in-time hook insertion technique for efficient dynamic AOP sysytems. Then we present an aspect weaver called Wool based on this technique. Although there are various existing techniques for dynamic AOP systems in Java, such as implementing an original VM, using a debugger, and inserting a hook in all join-point statically, have been proposed, huge performance degradation is a problem of these techniques. To reduce performance degradation, the technique proposed in this paper uses a mechanism for redefining a class at runtime, which the JPDA of Sun JDK1.4 supports. With this technique, the join-point is intercepted by a debugger with serious performance penalties only for the first time, but a hook is statically embedded at that join-point at the same time so that the join-point can be intercepted later with negligible penalties, and there is no degradation in use of classes without influence from aspects.

1. はじめに

現実的なアプリケーションの構築に必須となっている分散，同期，セキュリティ，セッション管理，トランザクション，ロギングなどのような非機能的な処理をオブジェクト指向言語ではうまくモジュール化できないということが問題視されている．アスペクト指向プログラミング（AOP）[10]では，このようなモジュール間にまたがる横断的関心事（crosscutting concerns）をクラスとは別にアスペクトとしてモジュール化する．また，アスペクトは weaver と呼ばれる言語処理系により合成（weave）される．

特に近年，実行時にアスペクトとプログラムを合

† 東京工業大学大学院情報理工学研究科
Graduate School of Information Science and Engineering, Tokyo Institute of Technology

成する Dynamic AOP（DAOP）が注目されている．DAOP ではアスペクトを合成するために実行中のプログラムを停止，コンパイル，ビルド，再スタートしないため，開発が効率化される．また，実行時の環境に応じて動的にアスペクトとプログラムが合成する適応的なアプリケーションを構築できるといった利点がある．

本稿では，効率的な DAOP システムを実現するために，ジャストインタイム・フック埋め込み技術を提案し，それを Java 用に実装した weaver である Wool について述べる．ジャストインタイム・フック埋め込みとは，アスペクトとプログラムの合成を可能な限り遅延させる手法である．可能な限り遅延させることで，プログラム中の指定された場所（join-point）で実行されるコード（アドバイス）を制御が及ばない場所に埋め込むのを避けることができる．したがってアスペクトとプログラムの合成処理が最適化され，結果的に DAOP システム全体の実行性能を上げることになる．DAOP システムでは合成にかかる時間が実行時間に含まれるため，高速な合成が望まれる．

Java でジャストインタイム・フック埋め込み技術を実現するにあたり，Wool は Sun Java2 SDK1.4（JDK1.4）の Java Platform Debugger Architecture（JPDA）のクラス再定義機能を利用した．クラス再定義機能は JDK1.4 で新たに導入された機能であり，同技術に基づく類似の研究はない．既存の手法には，Java 仮想機械（JVM）を改造する手法やコンパイル時にすべての join-point へフックを挿入する手法があるが，可搬性や実行性能に問題があった．Wool は JPDA を利用し，JVM を改造することなく効率的に DAOP システムを効率化する．また，本稿ではクラスを再定義する瞬間に JVM 内部でクラス定義の不整合が起こり，アスペクトが合成されない場合がある点について指摘し，その柔軟な解決手法を示す．

Wool ではアスペクトを Java で記述するための API（Application Programming Interface）を提供し，その中で合成のタイミングを制御するようなコードの記述ができる．DAOP ではアスペクトとプログラムが合成するタイミングがプログラムの実行に大きく影響するため，このような記述を言語機能として用意した．

我々は，Wool の効果を検証するために，いくつかの性能測定を行った．実験から，本研究で提案するジャストインタイム・フック埋め込み技術が既存手法に比べて性能向上を達成したことを確かめた．

以下では，まず次章で，従来の DAOP システムを概説し，その問題点を議論し，続いて 3 章で提案するジャストインタイム・フック埋め込み技術を説明し，4 章で Wool の設計，実装を述べ，5 章で性能測定を行い，6 章で本稿をまとめる．

2. Dynamic AOP

本研究では，AOP を実現するシステムの中でも実行時にアスペクトをプログラムに合成する DAOP システムを対象にした．本章では，まず，DAOP システムについて説明する．

2.1 Dynamic AOP システム

アスペクトとプログラムを合成するタイミングはコンパイル時，クラスロード時，実行時の 3 つに分類される．コンパイル時の AOP システムは，一般に処理系のプリプロセッサとして実装される．良質のコードを生成できるという利点があるが，分割コンパイルができず，特定のソース言語に依存した実装となってしまう．Java 言語にアスペクト指向を取り入れた，汎用アスペクト指向言語の 1 つである AspectJ[9] はそのような問題を解決しているが，実行時の合成は許していない．クラスロード時の AOP システムは，BCA[8] や Javassist[4] のようなロード時のバイトコード変換ツールを用いてアスペクトとプログラムを合成する．ソースコードが不要で分割コンパイルが可能だが，合成がクラスのロード時に制限される．実行時にアスペクトとプログラムの合成が行われる AOP システムを，特に Dynamic AOP システムと呼び，過去に様々な研究がなされてきた．

DAOP システムでは，アスペクトを用いた開発サイクルの高速化や実行時情報に基づく適応的なサービスをアスペクトで記述できるなどの利点がある．

開発サイクルの高速化 AspectJ の場合，ソースコードレベルでアスペクトとプログラムを合成する．したがって，実行中のアプリケーションを停止し，アスペクトとプログラムを合成，再コンパイルし，アプリケーションを起動し直すといった作業が必要となる．DAOP システムではアスペクトとプログラムの開発が分離され，またアプリケーションの実行を止めることなくアスペクトが合成されるので，開発サイクルが高速化される．

適応的なサービスアスペクト クライアントの要求や環境の変化に応じてサービスを動的に切り替える適応的なアプリケーションをアスペクトを使って構築できる．たとえば，セキュリティ処理のような汎化が脆弱化につながる危険性を持つ[19] 処理では，それに関わるコードが往々にしてモジュール間に散らばってしまう．たとえば，ローカルリソースを利用

するような汎用ルーチンを作成し，それをいろいろなところから再利用する場合，セキュリティホールになる場合がある．Java プロテクションドメイン[5]では，システムが提供するライブラリを介することで，一時的に権限をより大きなものに切り替える特権と呼ばれる機能（doPrivileged()）がある．しかし，システム固有のリソースを取得するために以下のようなメソッドを定義するのは危険である．

```
public Object getProp(String key) {
 return AccessController.doPrivileged(
   new PrivilegedAction() {
     public Object run() {
      return System.getProperty(key);
     }});
}
```

doPrivileged() を使用するライブラリは，権限の検査を行う checkPermission() の直前で，各リソースごとに doPrivileged() を呼び，権限を与えなければ，その特権を悪用されるかもしれない．つまり，一般にセキュリティ上危険が存在する処理は，再利用性のあるコードと共存しにくい．DAOP システムではユーザに応じたアクセス制限をアスペクトに記述し，実行時に合成することで動的にセキュリティポリシーを変更可能なアプリケーションを容易に開発できる．また，分散 GUI アプリケーションのコンポーネント配置などを，動的に変更する機能を持つ様々な ORB（Object Request Broker）が提案されているが[7),16)]，実行時に再構成する処理をアスペクトに記述し，使用するデバイスに応じて動的に切り替えるといったことも記述できる．

2.2 典型的な DAOP の実装手法

AOP を実現するモデルはダイナミックジョインポイントモデルと呼ばれ，AspectJ において最初に提案された．このモデルでは，フィールドアクセス，メソッド呼び出し，例外処理などの実行フロー中のあらゆる場所が join-point という形で定義される．そして，その join-point でプログラムの実行が横取りされ，アドバイスとして表現された処理が行われる．

オブジェクト指向言語でダイナミックジョインポイントモデルを実装するための典型的な方法は，join-point にあたる場所にフックを挿入し，フックした場所で適切なアドバイスを実行するという手法である．

2.3 既存の Java DAOP システム

実行時にアスペクトとプログラムを合成する DAOP システムは多数存在するが，リフレクションを処理系がサポートしている言語では，実行時に容易にフックを挿入することができる．たとえば，動作リフレクション[17),18)]は，メソッドのような演算を横取りして，その動作を変更することができる．しかし，リフレクション機能は CLOS や Smalltalk ではサポートされているが，Java や C++では完全にサポートされていない．本研究で対象とする Java では，制限されたリフレクション機能（イントロスペクション）しかサポートされていないため，過去に様々な DAOP を実現するための手法が提案されてきた．

リフレクションをサポートしない言語で，DAOP を実現する手法は，大きく分けて 2 つの要素技術に基づいている．1 つは，実行時にフックを挿入する機能を言語処理系にサポートさせる方法である．もう 1 つの方法は，どの join-point でアドバイスが実行されてもよいように，静的コード変換ですべての join-point にフックを挿入する手法である．この手法では，すべての join-point でアドバイスを実行するかどうかの判定がつねに行われる．

処理系の改造は，最も素朴な実装手法であるが，たとえば，Java のような可搬性が重要である言語には適用できず，現実的ではない．アスペクト指向を Smalltalk で実装した AOP/ST[2)] では，バイトコードインタプリタを改造し DAOP を実現している．また，PROSE[13)] のようにデバッガを利用した手法もあるが，大きなオーバヘッドが報告されている．PROSE は join-point に対応する場所で，デバッガがアドバイスを実行する．アドバイスの実行がデバッガによって行われるため，そのコンテキストスイッチがもたらすオーバヘッドは非常に大きい．

JAC[12)] や HandiWrap[1)] は，静的コード変換により，プログラム中のすべての join-point へフックを挿入するため，不必要なフックが多数埋め込まれてしまう．フックは join-point でアドバイスを実行するかどうかを判定するためのものである．つまり実行中にアドバイスを有効にするというフラグを立て，それをすべての join-point でつねにチェックする．この手法は，AspectJ でも用いられているが，静的な AOP では，フックを挿入する位置が静的に決定されるため，不必要なフックは挿入されない．DAOP で同様の手法を用いると，AspectJ で生成されるコードに比べ不必要なフックを数多く含み，多大なオーバヘッドが生じる．

3. ジャストインタイム・フック埋め込み

効率的な DAOP システムを実現するために，ジャストインタイム・フック埋め込み技術を提案する．処理系を改造する手法は，可搬性が重要な言語には適用

できず，PROSE のようにデバッガを用いても，大きく性能が劣化してしまう．静的コード変換による手法は，不必要なフックが数多く挿入されてしまう．ジャストインタイム・フック埋め込み技術は，この不必要なフックの挿入という問題を解決するための手法である．本章では，提案するジャストインタイム・フック埋め込み技術が，なぜ効率的な DAOP システムを実現するのかについて，その手法の詳細とともに示す．

3.1 遅延したアドバイスフックの埋め込み

我々は新たに，コード変換を実行時に行い，アドバイスフックの挿入を遅延させ，効率的な DAOP を実現する手法を開発した．これにより，フックを用いて join-point を表現する DAOP システム全般の速度向上が期待できる．この手法では，アスペクトとプログラムの合成，すなわちプログラム再定義が実行時に行われ，かつ実際の合成は遅延され，オンデマンドに行われる．つまり，再定義する必要のあるクラスが判明したときになって初めて行われる．したがって，実際にはアスペクトを合成しないプログラムの join-point に，アドバイス実行を判定するための無駄なフックを追加するということはない．実行時に，プログラムを再定義するので，フックはプログラム内に直接埋め込まれる．一度プログラム内に埋め込まれたフックはプログラムにそのまま実行されるので，コンテキストスイッチが起こるのは，初めにアスペクトとプログラムの合成が行われるときのみである．

多くの静的コード変換を採用する DAOP システムと同様に，コードを埋め込む際にアドバイスコード自体ではなく，アドバイスを実行するためのフックを埋め込むことでアドバイスを容易に取り外したり切り替えたりすることが可能となり，また不正なアドバイス処理を大域的にチェックすることができる．

3.2 active フレームへの対応

実行時にアドバイスフックを埋め込むと，そのときに実行中のメソッドを書き換えてしまう場合がある．DAOP システムにはこのような場合でも正しくアドバイスを実行する仕組みが必要である．たとえば，本棚（BookShelf）から本（Book）を検索するプログラムに，検索履歴をとる機能を追加する場合を考える．この機能を追加するために，次のように，BookShelf の search() を呼び出す前に，検索語を記録し，呼び出し後に，結果を記録するアスペクト（HistoryAspect）を書くとする．

```
aspect HistoryAspect{
 before(String keyword) :
  call(* BookShelf.search(String))
  && args(keyword) {
    record keyword
  }
 after() returning(Book book) :
  call(* BookShelf.search(String)){
    record found bookname
  }
}
```

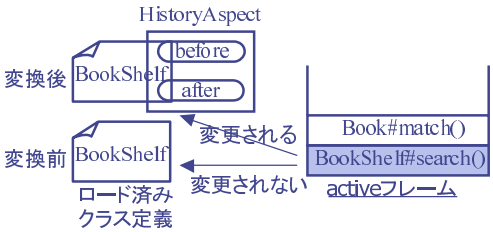

図 1 メソッド実行中のスタック
Fig. 1 State of stack in execution.

図 1 は，BookShelf の search() を呼び出し，本棚に属する本を表す Book の match() を呼び出し，キーワードにマッチする本を探すプログラムの実行の途中の状態を表す．HistoryAspect を合成する側では，このような途中の状態で合成する場合でも，正しく検索履歴が保存されることを期待する．しかし，search() を呼び出したときに積まれたフレームが参照していた BookShelf は，Book の match() が終了し，実行が戻ったときに，HistoryAspect が合成したものに書き換えられているため，正しくアドバイスが実行されない．

通常，言語処理系はスタックマシンとして実装され，メソッド呼び出しは，スタックにそのメソッド用の環境を表すフレームを積むことで実現される．実行時のアドバイスフック埋め込みで書き換えられるのはクラス定義である．クラス定義が書き換えられると，そのときにスタックに積まれているフレーム（active フレーム）でのメソッド呼び出しでは，正しくアドバイスが実行されない．そこで，DAOP システムは active なフレーム，すなわち実行の途中のメソッドでも正しくアドバイスを実行しなければならない．active フレームで，メソッドが参照するクラス定義が変更されるかどうかで 2 つの挙動が考えられる．

クラス定義が変更されない場合 active フレームに制御が移ったときにアドバイスが実行されないため，アスペクトの合成するタイミングによって同じ join-point でアドバイスが実行される場合とされない場合が生じてしまう．

クラス定義が変更される場合 保持しているプログラ

図 2 active フレームでのアドバイスの実行
Fig. 2 The execution of advices on active frame.

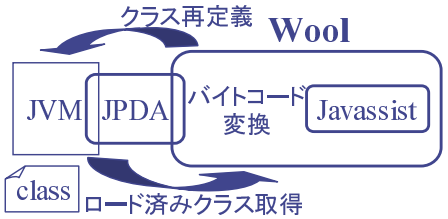

図 3 Wool のアーキテクチャ
Fig. 3 An architecture of Wool.

ムカウンタと，フックが埋め込まれたクラス定義とで不整合が生じてしまう．メソッド呼び出しが起こり，新しいフレームをスタックに積むとき，現在実行中のフレームを中断し，呼び出しが終了した後にその場所から再開するために戻り番地（プログラムカウンタ）の情報を新しいフレームに記憶する．しかし，before アドバイスなどで，中断前のコードの位置にフックが埋め込まれると，中断しているプログラムカウンタの値が古いままなので，積まれたフレームが持つ戻り番地とで狂いが生じてしまう．

提案するジャストインタイム・フック埋め込み手法では，active フレームの問題に対応するためにデバッガを利用する．多くの言語に用意されるデバッガを使用することで，上記 2 つのどちらの場合でも active フレームの問題に対応できる．クラス定義が変更されない場合，active フレームでは，アドバイスを実行するために，前述した PROSE と同じくデバッガを用いる．クラス定義が変更される場合，クラス定義を元に戻さなければ，正しく動作しないので，デバッガを用いてメソッドの終了をインターセプトし，クラス定義を元に戻し，同様にデバッガがアドバイスを実行する．この場合，新しくメソッドを呼び出すときには，再びアスペクトの合成されたクラスを再定義しなければならない．デバッガを利用して active フレームにおいてアドバイスを実行する様子を**図 2** に示す．

アドバイスの間に依存関係がある場合，必ずしも active フレームでアドバイスを実行することが望まれるというわけではない．たとえば，`BookShelf` クラスで，検索にかかる時間を測定する機能をアスペクトで追加する場合を考える．このようなアスペクトでは，`search()` の直前と直後で時間を計り，その差分が検索時間となる．図 1 の状態でアスペクトを合成したとき，`search()` の直前で時間が計測されていないため，誤った検索時間が計測されてしまう．このような場合は，active フレームで after アドバイスだけが実行されてはいけない．Wool は，この問題に対して，4.3 節で述べる weaver を制御するための記述を，アスペクト開発者に許すことで柔軟に対応する．

4. Wool

ジャストインタイム・フック埋め込み技術を Java に適用するために実装した weaver である Wool について説明する．

4.1 Wool の概観

Wool は，アスペクトを記述するための API と，アスペクトとプログラムを合成する weaver，プログラム外部からの weave を受け付けるサブシステムからなる．実行時にプログラムとアスペクトを合成するためのコード変換に，バイトコード編集ツール Javassist を用いた．また，JVM にすでにロード済みのクラスを再定義するために，JDK1.4 の JPDA の HotSwap 機能[15] を利用した．Wool が行う合成処理は**図 3** に示されるように，ロード済みクラスの取得，Javassist 使ったバイトコード変換，JPDA を用いた改変クラスの再定義，という流れで行われる．

本章では，まず Wool でのアスペクトを例示し，そこで DAOP システムに必要であると我々が考える，weave のタイミングの制御を取り上げる．その後，ジャストインタイム・フック埋め込み技術の詳細な実装について論じる．

4.2 アスペクトの記述

Wool はアスペクトを Java 言語で記述するための API を提供する．記述性に重点を置くような多くのアスペクト指向言語では，柔軟かつ直観的なポイントカットを記述するために独自の言語を提供しているが，Wool は実行性能の改善を目的としているため，ダイナミックジョインポイントモデルを実現するシンプルな Java のクラスを提供する．**図 4** に 3.2 節で示した AspectJ のコードを Wool で記述した場合のコード例を示す．

ポイントカットの記述 AOP では，プログラム中に適切に定められた join-point を抽出するための書き方が用意されている．join-point とは，モジュール間をある関心事に従って横断するときの基準となる場所であり，それらの join-point の集合を，いかに柔軟に記

```
1: public class HistoryAspect extends WlAspect {
2:  Pointcut record
      = Pointcut.methodCall("*","BookShelf","search"
                "(Ljava/lang/String;)V");
3:  public void weave(Wool wool) throws WeaveException {
4:   wool.insert(new WlBeforeAdvice(record) {
5:     public void advice(TargetInfo t) {
6:       recordBook(t.retValue()); } });
7:  }
8:   wool.insert(new WlAfterAdvice(record) {
9:     public void advice(TargetInfo t) {
10:       recordKey(t.args(1)); } });
11:  }
12:  public void initWeave(Wool wool)
13:   throws ThreadNotWeavableException {
14:    wool.filterClass("^java.*|^sun.*", false); }
15:  public void beforeWeave(Wool wool)
16:   throws ThreadNotWeavableException {
17:    if (wool.hasActiveFrame("main")) wool.skip(); }
18: }
```

図 4 Wool のアスペクト例（HistoryAspect）
Fig. 4 An example of aspect for Wool.

述するか，またはいかに動的な情報に基づいて抽出するかといった研究も数多くなされている[3),6)]．join-point の集合を抽出する操作をポイントカットと呼ぶ．Wool では，たとえば，図 4 の 2 行目のように，`Pointcut` クラスの `methodCall()` メソッドなどを用いて，その引数から決まるメソッドに関連する join-point の集合を抽出する．メソッドに関連する join-point を抽出する場合は引数が修飾子，クラス名，メソッド名，シグニチャをとる．そのほか，フィールドやコンストラクタに関しても同様に用意されているが，それ以外のコントロールフローに依存した join-point などは現段階ではサポートしていない．

アドバイスの記述 アドバイスはあらかじめポイントカットされた join-point の集合で行われる処理を表す．join-point はプログラム中のあらゆる場所を網羅するよう設計されるため，通常アドバイスは，join-point の直前（before）での処理，直後（after）での処理，または join-point での処理を横取りして行う処理（around）の 3 つがある．Wool では，図 4 の 3〜11 行目のように `weave()` というコールバック関数を実装し，その中で 4, 8 行目の `insert()` の体裁に合わせてアドバイスを記述する．`insert()` は，引数として `WlAdvice` クラスのサブクラスである `BeforeAdvice` や `AfterAdvice` をとり，`advice()` に join-point で実行する処理を記述する．`advice()` には，join-point の情報や呼び出したオブジェクトの情報，引数の情報などのコンテキスト情報を格納した `TargetInfo` オブジェクトが渡される．また，現段階では Wool は around アドバイスをサポートしていない．

イントロダクションの記述 既存のクラスにフィールド，メソッド，コンストラクタなどの要素を新たに追加したり，クラスの継承関係や派生関係のようなクラス階層を変更したりする機能をイントロダクション機能と呼ぶ．Wool ではこの機能をサポートしていない．なぜなら，JVM の HotSwap 機能が，実行時のクラス要素の追加やクラス階層の変更を許していないからである．しかし，イントロダクションによって追加されるクラス要素は，既存のクラスからは使用されず，アドバイスで定義されたコードから間接的に参照される．したがって，クラスロード時に各クラスにイントロダクションを格納するためのマップをインスタンスとして追加し，新たに追加されたクラス要素へのアクセスは，そのマップを介して行うようにすれば，容易に実行時のクラス要素の追加を実装することができると考えている．

4.3 weaver を制御する記述

Wool でアスペクトとプログラムを合成するには，プログラム内部に，Wool を起動するコードを記述する方法と，プログラムの外部から，Wool を起動する方法との 2 種類がある．具体的には下に示すように，Java プログラムで記述する場合とコマンドラインから起動する場合がある．

内部から

```
WlAspect azpect
    =WlAspect.forName("CertainAspect");
Wool wool
    =Wool.connect("my.domain.com",5432,
azpect);
wool.weave(azpect);
```

外部から

```
% wool -weave CertainAspect -address 5432
```

プログラム内から Wool の weaver を起動する場合，アスペクトを表すオブジェクトを生成し，Wool を表すオブジェクトへ，サブシステムが動作している URI（Uniform Resource Identifiers）とポート番号，合成するアスペクトを渡して接続する．その後，接続で得られた Wool オブジェクトの `weave()`, `unweave()` を使ってアスペクトの合成，切り離しを行う．コマンドラインで別プロセスから起動する場合は，コマンドライン引数として-weave オプションと合成するアスペクト，サブシステムが動作する URI とポート番号を指定する．

Wool は，開発者にアスペクトプログラムから，

表 1 weaver を制御するためのメソッド
Table 1 Methods for controlling weaver.

weaver 制御用関数	呼ばれるタイミング	主な利用法
`initWeave()`	weave 直後の初期化	`filtering()` でアスペクトが影響するクラス集合をフィルタリング
`beforeWeave()`	新しいクラスを再定義する直前	`skip()` でクラス再定義処理をスキップ
`afterWeave()`	新しいクラスの再定義した直後	`regardActiveFrame()` で active フレームへ対応の有無を決定
`beforeUnweave()`	クラス定義を元に戻す直前	`skip()` でクラス再定義処理をスキップ
`afterUnweave()`	クラス定義を元に戻す直後	I/O ストリームを閉じるなどの finalize 処理

weaver を制御する機能を提供する．weaver の制御とは，アスペクトとプログラムが合成したり，切り離されるタイミングなどを調整したりすることである．そのために，アスペクトプログラムの中で weaver の初期化（`initWeave()`），アスペクトの合成の前後（`beforeWeave()`，`afterWeave()`），アスペクトの切り離しの前後（`beforeUnweave()`，`afterUnWeave()`）での処理を記述することができる．具体的には，`WlAspect` クラスのコールバックメソッドとして提供され，weaver を制御するために Wool を表すオブジェクトが渡される．

3.2 節で示したように，DAOP は静的な AOP 言語と違って，アスペクトとプログラムを合成するタイミングがプログラムの実行に大きく影響を及ぼす．しかし，プログラム外部から weaver を起動する場合，いつ weave するのか，いつプログラムに反映されるのかはまったく不定である．また，実行中のプログラムの active フレームで，アドバイスを実行するかどうかは実行時情報をもとに柔軟に決定されなければならない．したがって，Wool は，プログラムが実際に変更されるタイミングを制御する仕組みが必要である．図 4 の 12〜17 行目は weaver を制御するコードを記述したものであり，**表 1** に Wool が提供する制御ポイントとそこで可能な処理の例を示す．

4.4 Java でのジャストインタイム・フック埋め込み

3 章で，我々が提案したジャストインタイム・フック埋め込み技術を Java で実装するための詳細な手法を説明する．

実装方針 Wool は，ジャストインタイム・フック埋め込み技術を Java で実装するために JPDA を利用した．JPDA を使うことで，Java の重要な利点である可搬性を損なわずに実装することが可能となる．

別な実装方法として，Just-in-Time（JIT）コンパイラ技術を利用する手法が考えられるが，Wool では採用しない．OpenJIT[11] のような自己反映的な JIT コンパイラを利用すれば，実行中に JIT の動作を変更することができる．アスペクトによってコンパイルコードを変更するような実装を行えば，ジャストインタイム・フック埋め込みを実現できるとが考えられる．その方法では，メソッド単位でアスペクトとの合成が行われるので，より多くの不必要なフックの埋め込みを避けられる．JIT を使ってコードを再コンパイルし，フックを埋め込む場合でも，active フレームの対応処理は必要である．可搬性の問題から今回の実装では採用しないが，さらなる高速化を望める手法であると考えている．

遅延したアドバイスフックの埋め込み 実行時にアドバイスフックをクラス定義に埋め込み，その処理をできるだけ遅延させるために JPDA の HotSwap 機能を利用した．Wool は，プログラムとアスペクトの weave 処理では合成はせず，ロード済みクラスを走査し，ポイントカットで指定された join-point にあたる場所へ `BreakpointEvent` をセットする．プログラムの実行が breakpoint まで及ぶと，適切にプログラムがバイトコード変換される．そして，`ThreadReference` クラスの `redefineClass()` を使って JVM にロード済みのクラス定義を変更する．`redefineClass()` は，ロード済みクラスを実行時にデバッガの制御のもとで変更する機能である．また，`BreakpointEvent` も JPDA で提供されるクラスであるが，JDK1.4 の JPDA では breakpoint をセットしていても高速な HotSpot 実行（JIT とインタプリタの共存）が可能となっているため，合成を遅延させてもオーバヘッドはほとんどない．

active フレームへの対応 Java の実行系である Java 仮想マシンはスタックマシンであり，JPDA の仕様によれば `redefineClass()` を行った後では，新しく積まれるフレームは新しいクラス定義を参照するが，そのときに，active であるフレームが参照するクラス定義は変更されない．つまり，単純に `redefineClass()` でクラス定義を変更しても，3.2 節で論じたクラス定義の一貫性が崩れるという事態に陥ってしまう．

3.2 節で述べたように，Wool は，active フレームでアドバイスを実行するために，デバッガを用いる．クラス定義を書き換えた後に，すべてのスレッドのスタックフレームを走査し，不整合が生じたフレームを

表 2 プログラムの実行時間とアスペクトの合成処理時間
Table 2 Execution time of program and time of aspect weaving.

	AspectJ	PROSE の手法	Wool	join-point 抽出 +変換再定義
(N) (1)	8590	7388812	19638	1680 + 4057
(N) (2)	8522	23307	11832	1680 + 806
(M) (1)	1063	45817	11003	1680 + 4057
(M) (2)	1003	3833	3993	1680 + 806

単位：[msec]

探索する．次に，そのメソッド中のポイントカットで指定された join-point に `BreakpointEvent` をセットする．そのフレームでは，再定義前のクラス定義が実行されるため，アドバイスの実行は breakpoint でデバッガが実行する．すべての active フレームがなくなったら breakpoint を削除する．

5. 性能測定

本稿で我々が提案したジャストインタイム・フック埋め込み技術の有効性を示すために，我々は，Wool のプロトタイプを実装し，それを用いて従来手法との比較実験を行った．実験に用いた計算機は，Sun Blade1000（CPU: UltraSPARC-III 750 MHz×2, memory: 1 GB）である．OS は Solaris8, JVM として Sun Java2 SDK v1.4.0 の HotSpot^{TM} Client VM を用いた．実験結果はすべて 5 回繰り返して測定し，その最小値を選んでいる．

1 つ目の実験では，アスペクトの合成処理にかかる時間とプログラムの実行時間を測定するために，SPECjvm98 [14] の jess という Java で書かれた CLIPS のエキスパートシステムでナンバーパズル問題（N）とモンキーバナナ問題（M）を測定した．それぞれに対して実行時に次の 2 種類の異なるアスペクトを合成した．

(1) すべての protected メソッドの先頭に null アドバイスを挿入するアスペクト．146 クラス中の 4 つのクラスが実行時に再定義される．
(2) 特定のクラスの指定する 1 つのメソッドの先頭に null アドバイスを挿入するアスペクト．

weave するタイミングはプログラムの開始直後としたため，weave 時に考慮すべき active フレームはない．

1 つ目の実験結果を**表 2** に示す．参考に AspectJ の結果も載せる．比較のため，PROSE で採用されているデバッガがすべてのアドバイスを実行する手法も測定した．(1) のアスペクトが weave するメソッドは 734,615 回使用されていて，(2) の場合 1,338 回使用されている．デバッガがすべてのイベントを実行する手法では，アスペクトが weave されるメソッドが使用されるたびにコンテキストスイッチが起こりデバッガがアドバイスを実行するため，極端なオーバヘッドが出ている．Wool では (1)，(2) のアスペクトの合成処理にかかった時間がそれぞれ 5,737 ミリ秒，2,486 ミリ秒であり，それを含めた総実行時間でもかなりの性能改善が見られた．特にアスペクトが weave するメソッドの使用回数が増えるほど，Wool ではフックがコード中に埋め込まれているため，良い結果が期待できる．

2 つ目の実験では，active フレーム処理のオーバヘッドを計測するために，自然数の和を求める再帰メソッド `sum()` で，アスペクトが合成されるときに active なフレームの個数を増やしていき，そのときの実行時間を測定した．合成するアスペクトは，`sum()` の前後に null アドバイスを挿入する．

2 つ目の実験の結果を**図 5** に示す．Wool は active フレーム数が増加するほどするほど，性能が劣化している．active フレーム数が増えるとアドバイスを実行するためにデバッガに頻繁に制御が移るため，このような結果が出た．また，現段階ではフレームが参照するクラス定義が変更されたかどうかをデバッガがつねにチェックするため，遅くなっている．たとえば再帰呼び出しの 20 番目でクラス定義が書き換えられたとしても，その後 180 回再帰呼び出しをして，呼び出しから 200 回戻るときに，200 個のフレームすべてで，デバッガがチェックするような実装になっている．それに対して，イベントフックに基づいた手法は，active フレームは影響せず，純粋にアドバイスを実行するためのコンテキストスイッチが減少するので，active フレーム数増加とともに処理時間が短くなっている．

6. まとめ

本稿では，効率的な DAOP システムを実現するための手法として，ジャストインタイム・フック埋め込み技術を提案し，それを Java 用に実装した weaver である Wool について述べた．Wool は，アスペクトとプログラムの合成のためのフックの挿入を可能な限り遅延させ，バイトコード変換でアドバイスを実行するフックをプログラムに埋め込み，クラスを実行時に再定義する．したがって，対象プログラムに無駄なコードを埋め込まない．実験から，アスペクトの合成後の対象プログラムは，実行時に生成された不必要なフックのない良質のコードになっており，高速に実行されることを確認した．

Wool は，JPDA の HotSwap 機能でアドバイスフッ

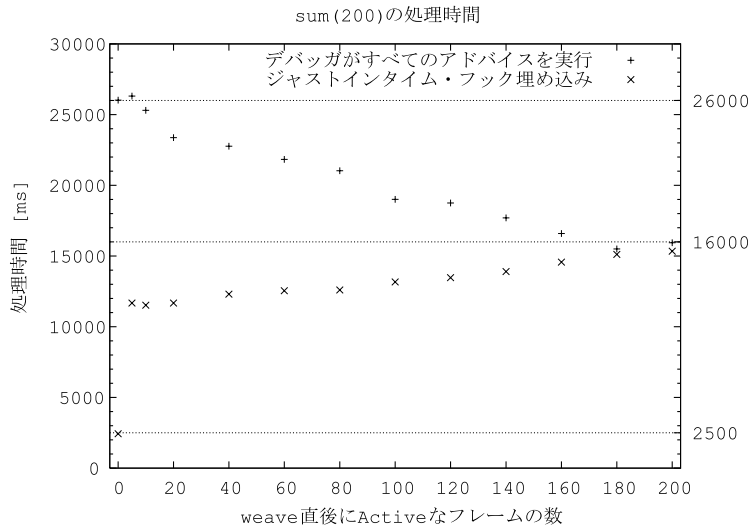

図 5 active フレーム処理のオーバヘッド
Fig. 5 Overhead of handling active frames.

クを実行時にプログラムに埋め込み，active フレームのメソッドでは，デバッガがアドバイスを実行する．HotSwap 機能は，JDK1.3 以前にはなかったものであり，同技術による類似研究は我々が知る限りまだない．active フレームに対応する場合，実験で大きな性能劣化が確認された．しかし，Wool は weaver を制御する機構を用意し，active フレームの対応や合成のタイミングを，アスペクト開発者に柔軟に決定させるので，アスペクト開発者は状況に応じてより良い処理を選択することができる．

今後は，未実装のポイントカット指定やイントロダクション記述などの AOP 言語として必要な機能を実装し，さらに性能を向上させる工夫を施していく必要があると考えている．具体的には，すべてのフックをコード書き換え・再定義しないような仕組みを考えている．実験を行った結果，アドバイスの実行が少ないときにコード変換・再定義がボトルネックとなるため，実行回数に基づきアドバイス実行方法を手動，あるいは自動で選択するような実装が望ましいと考える．

参 考 文 献

1) Baker, J. and Hsieh, W.: Runtime Aspect Weaving Through Metaprogramming, *AOSD 2002*, pp.86–95 (2002).
2) Böllert, K.: On Weaving Aspects, Position paper at the ECOOP'99 workshop on Aspect-Oriented Programming (1999).
3) Brichau, J., Mens, K. and Volder, K.D.: Building Composable Aspect-Specific Language with Logic Metaprogramming, *Generative Programming and Component Engineering*, LNCS 2487, pp.93–109, Springer-Verlag (2002).
4) Chiba, S.: Load-time structural reflection in Java, *ECOOP 2000*, LNCS 1850, pp.313–336, Springer-Verlag (2000).
5) Gong, L. and Schemers, R.: Implementing Protection Domains in the Java Development Kit 1.2, *Internet Society Symposium on Network and Distributed System Security* (1998).
6) Gybels, K.: Using a logic language to express cross-cutting through dynamic joinpoints, *Workshop on Aspect-Oriented Software Development* (2002).
7) Joergensen, B.N., Truyen, E., Matthijs, F. and Joosen, W.: Customization of Object Request Brokers by Application Specific Policies (2000).
8) Keller, R. and Hëlzle, U.: Binary Component Adaptation, *ECOOP'98 — Object-Oriented Programming*, LNCS 1445, pp.307–329, Springer-Verlag (1998).
9) Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G.: An Overview of AspectJ, *ECOOP 2001*, LNCS 2072, pp.327–353, Springer-Verlag (2001).
10) Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J.: Aspect-Oriented Programming, *Proc. European Conference on Object-Oriented Programming*, Vol.1241, pp.220–242, Springer-Verlag, Berlin, Heidelberg, and New York (1997).
11) Ogawa, H., Shimura, K., Matsuoka, S., Maruyama, F., Sohda, Y. and Kimura, Y.:

OpenJIT Frontend System: an implementation of the reflective JIT compiler frontend, *ECOOP 2000*, LNCS 1850, Springer-Verlag (2000).
12) Pawlak, R., Seinturier, L., Duchien, L. and Florin, G.: JAC:A Flexible Framework for AOP in Java, *Reflection'01*, pp.1–24 (2001).
13) Popovici, A., Gross, T. and Alonso, G.: Dynamic Weaving for Aspect-Orinented Programming, *AOSD 2002*, pp.141–147 (2002).
14) Spec — The Standard Performance Evaluation Corporation: SPECjvm98 (1998). http://www.spec.org/osg/jvm98/
15) Sun Microsystems: Java^{TM} Platform Debugger Architecture (2001). http://java.sun.com/j2se/1.4/docs/guide/jpda/index.html
16) Truyen, E., Jrgensen, B.N. and Joosen, W.: Customization of Component-Based Object Request Brokers through Dynamic Configuration (2000).
17) Welch, I. and Stroud, R.: Kava — Using Bytecode Rewriting to add Behavioural Reflection to Java, *USENIX Conference on Object-Oriented Technology* (2001).
18) Wu, Z.: Reflective Java and A Reflective Component-Based Transaction Architecture, *OOPSLA'98 Workshop on Reflective Programming in C++ and Java* (1999).
19) 高木浩光：オブジェクト指向とセキュリティ脆弱性，情報処理学会ソフトウェア工学研究会，オブジェクト指向 2002 シンポジウム (2002).

(平成 14 年 12 月 24 日受付)
(平成 15 年 7 月 1 日採録)

佐藤　芳樹

1977 年生．2000 年東北大学工学部情報工学科卒業．2002 年同大学院情報科学研究科情報基礎科学専攻修士課程修了．2002 年より東京工業大学大学院情報理工学研究科数理・計算科学専攻博士課程．言語処理系，分散処理システム，プログラミングに関する研究に従事．ACM 学生会員．

千葉　　滋（正会員）

1968 年生．1991 年東京大学理学部情報科学科卒業．1993 年同大学院理学系研究科情報科学専攻修士課程修了．1996 年同専攻博士（理学）取得．同専攻助手，筑波大学電子・情報工学系講師，東京工業大学情報理工学研究科講師を経て，現在同助教授．言語処理系およびオペレーティングシステム等システムソフトウェアの研究に従事．日本ソフトウェア科学会，ACM 各会員．