平成21年度 学士論文

分散動的アスペクト指向言語の
アラウンドアドバイスの
実装方法

東京工業大学 理学部 情報科学科
学籍番号 06-1978-0
早船 総一郎

指導教員
千葉 滋 教授

平成22年2月3日

概要

今日、分散環境でアプリケーションを利用することが増加していることで、分散アプリケーションに対して横断的関心事をまとめる技術として分散動的アスペクト指向プログラミングが注目されている。分散動的アスペクト指向プログラミングは分散アスペクト指向プログラミングと動的アスペクト指向プログラミングの特徴を備えた技術である。本技術の実用化を考えると標準の Java 仮想機械を使うため Hotswap を利用し、単に各マシンでアスペクトを織り込むのではなく、我々のグループで提案した remote pointcut のような機構を利用し、静的と同じ機能が提供されるべきである。

しかし、従来の分散動的アスペクト指向プログラミングでは around advice に関して複数の制限があり、一部の機能が未実装や不可能である処理系がほとんどである。around advice とは、ある期間全体の処理の代わりに実行される処理である。まず、多くの処理系では、execution ポイントカットを選び proceed を呼び出す around advice が利用できない。proceed は元のジョインポイントの期間に対応する処理を渡された引数を使って実行するものである。この制限の原因は、メソッドボディにアドバイスの呼び出しが織り込まれるため、織り込み先のメソッドボディを退避できず、proceed が元々の処理を呼び出す手段をなくしてしまうからである。メソッドボディの退避を行うためにメソッドの追加が必要だが、これは Java の Hotswap の制限によって動的に新たなメソッドの追加ができない。また、従来の分散動的アスペクト指向プログラミングの proceed の実装方法では、そのまま remote pointcut を選択し proceed を呼び出す around advice を実装できない。そこで、proceed 呼び出しを remote pointcut に対応させる必要がある。remote pointcut とは、遠隔ホスト上のジョインポイントをポイントカットとして選択することができる技術である。

本研究では、分散動的アスペクト指向言語の around advice の実装方法として、前者の課題について、ロード時にメソッドラップを行う手法を提案する。これにより、動的に行えないメソッドボディの退避をロード時に行うことで Hotswap の制限を回避して、メソッドボディの退避を行い、ラッパーにフックを織り込むことで解決している。また、後者の課題について、リモート呼び出しが実行コンテキストを含むように改造し、それ

によりメソッドディスパッチを行う実装方法を提案する。proceed が直接的に呼び出したいメソッドを呼び出せるように、本システムにインスタンスを登録し、インデックスなど情報を利用して本システムが proceed 呼び出しのメソッドディスパッチを行う。各 proceed 呼び出しは与えられた次の呼び出しに必要な情報を引数に持つことでメソッドディスパッチを行う。これにより、proceed がリモート呼び出しを行えるようになった。さらに、同じジョインポイントに結びついたアラウンドアドバイスが複数ある場合の処理が複雑とされていた。しかし、本システムが次の呼び出しに関する情報に、アプリケーションのメソッドを指定するだけでなく、織り込んだアスペクトのアドバイスを指定を行うことで、同じジョインポイントに結びついた around advice の場合にも対処できるようになった。

また、実験により本システムを用いた場合のオーバーヘッドは、ラップされたメソッドを呼び出すことや Java エージェントによるオーバーヘッドが約 3 %であり、これに加えて起動時に行われるラップのオーバーヘッドがかかることがわかった。織り込みと unweave には通信によるオーバーヘッドが大きく、バイトコードの変換よりもかかる時間が大きいことがあることがわかった。不必要な通信を行わない実装を行わなければならないことが重要である。

謝辞

本研究は以下の方々なくして、存在しえなかったでしょう。研究の提案や方針などになにかと心を砕いていただき、多くの指導をしていただいた指導教員の千葉滋教授に感謝致します。

そして、論文のスタイルファイルを作成していただいた光来健一氏に感謝致します。森田悟史氏は、研究についての多大な知識を与えていただき、様々な意見と助言をいただきました。さらに実装方法や論文の書き方なども指導していただきました。心から感謝致します。

そして研究室のみなさん。心より感謝しています。

目 次

図 目 次

表 目 次

第1章　はじめに

今日、分散環境で、つまり複数の計算機上で動作するアプリケーションが注目されている。分散環境にある横断的関心事をモジュール化する Distributed Aspect-Oriented Programming (分散 AOP) が研究されてきた。Aspect-Oriented Programming (AOP) は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横断的関心事 (crosscutting concerns) をモジュール化することができる技術の一つである。まず、分散環境に関する特有の問題に対処しなければならないが、オブジェクト指向や AspectJ などで記述する場合、分散に関する記述をモジュール化することはできなかった。そこで、分散に関する記述もモジュール化できる分散 AOP が研究されてきた。

加えて、プログラムの実行を止めずに振る舞いを変更する技術として、Dynamic AOP が注目されている。これを DAOP という。DAOP では、アスペクト指向のモジュール単位であるアスペクトを動的に織り込み (weave)・削除 (unweave) をすることができる。通常のプログラムでは、事前に準備しておいた振る舞いにしか変更できないため、任意の振る舞いを変更する場合には、プログラムの実行を止め、再コンパイルし、実行しなおす必要がある。そして、この二つを合わせた分散動的 AOP も研究されている。それは、分散環境で動いているプログラムを実行を止めることなく、動的に変更を加え、振る舞いを変更することができるものである。

しかし、分散動的アスペクト指向言語では機能として制限をかけていることが多い。特に around advice は AOP の中でも特殊なものとしてとらえられている。他のアドバイスとは異なる部分が多く、実装を避けたり、実装が不可能であったりする。そのため、機能に様々な制限がかかっていることもある。まず、DAOP の観点から考えると実装を困難にしている原因となるのは、Java の Hotswap による制限がある。これによって動的なメソッドボディの退避が行うことができない、この制限により静的で行われているような単にボディの退避を行う方法は万能とは言えない。標準の JVM でなく、Jikes Research Virtual Mathine (Jikes RVM) [4] などを拡張したものを利用した VM レベルの織り込み方法を用いた場合は Hotswap とは異なるため、メソッドボディの退避を行えるものがある。しかし、それは実用性を考えるとは言えない。なので、今回の議論の対象か

ら外す。

分散 AOP の観点から考えると remote pointcut を用いた場合に proceed に対応することができないことが論点となる。これは proceed によるメソッド呼び出しがリモートによるものであるため、特別に対処する必要がある。それを避けるために、around advice の利用方法としては、制限をかけローカルホストのみに限定している処理系がほとんどである。このように機能の制限のかけ方としても、そもそも実装を行わず利用させないような実装や、proceed を必ず一度呼び出すような実装、さらに同じジョインポイントに対しての織り込みを許さない実装などがある。

本稿の残りは、次のような構成からなっている。第 2 章では、分散動的アスペクト指向言語においての around advice の問題点を述べ、第 3 章では、本研究で開発したフレームワークによるシステムでの提案、第 4 章では、システムの実装方法、第 5 章では、システムの性能を測る実験、そして、第 6 章では、本研究のまとめと今後の課題について述べる。

第2章　分散動的アスペクト指向言語のaround adviceの問題点

分散アプリケーションを実行している時に、そのアプリケーションの振る舞いを変更することや機能的に拡張することが必要とされる場合がある。そのような要求に対して既存の方法での解消方法を説明し、その既存の解決方法では解消することができない問題点を述べる。

2.1　分散アプリケーション

ネットワーク技術が向上している中で、複数のホストを利用する分散アプリケーションの需要が高まっている。Java では分散アプリケーションを開発するためのモジュールが多く用意されている。その中には、Java 標準の Remote Method Invocation (RMI) を用いたプログラムなどが開発され、これによりネットワーク越しでリモートに参照を行えるプログラムが作られている。そのようなアプリケーションはネットワークを用いて、莫大な計算を分散させているアプリケーションが多く、その実行を止めることは今までの計算を捨ててしまうことである。計算量の多いプログラムは、分散をしていても実行にかかる時間が多い、そのため、実行を止めることはできるだけ避けたい。

2.1.1　N 体問題

分散アプリケーションを考えるにあたり、例として上げられる一つが、N 体問題である。N 体問題とは、N 個の質点間の重力を計算することで、質点の運動をシュミレーションする問題である。これは N 個の質点がある時刻に存在する位置を初期位置として、T 単位時間後の位置を求めるものであり、質点が同じ座標に二つ以上存在することを許し、重なって消滅することや分裂することがないというものである。2 体問題までは計算として解くことが可能であり、3 体問題以上であると式を解くことによる計算が不可能となるため、ステップごとに各質点間の相互作用を計算することが必要とされる。ステップごとにすべての質点間の相互作用を計算す

るので、質点の数が多いほど計算に時間がかかる。この計算をより速く解くために分散させたり、近似を行ったりすることが行われている。

近似方法

特にこの近似の仕方には有名なものがあり、互いの質点による相互作用が局所性をもっているというという性質から、ある一定距離以上の遠くからの相互作用がないととらえることで近似をおこなっている。しかし、それでも質点が N 個の場合に逐次計算した場合では $O(N^2)$ の計算量がかかってしまう。この近似を利用し、改良したものが粒子登録法と領域分割法である。

領域分割法

領域分割法は、図 2.1 のようなものである。シュミレーションする空間を一定距離 r でメッシュ状に分割し、隣接する領域上にある質点のみ相互作用をするものととらえるものである。まず、シュミレーション空間をメッシュ状に分割します。分割されたそれぞれの空間をセルと呼び、セルのサイズをある一定距離以上にすることによって、相互作用する可能性がある範囲は、自分の存在するセルとその近傍セルに限定する方法である。

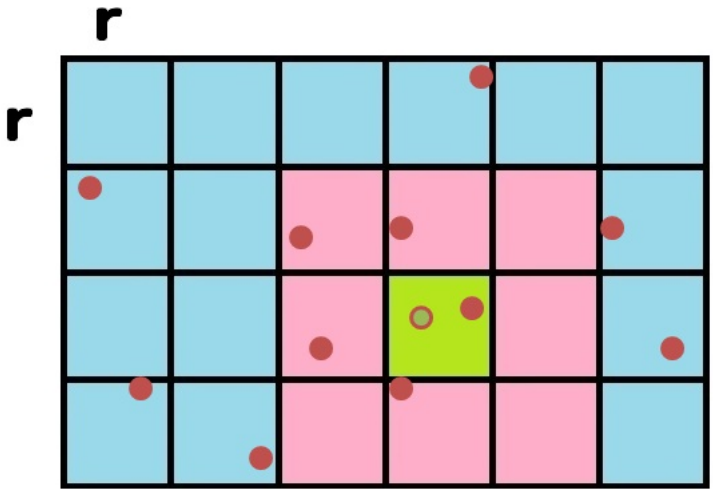

図 2.1: 領域分割法

粒子登録法

粒子登録法は、図 2.2 のようなものである。ある一定距離 r 以上のものは対象外にするという考えで、相互作用をする可能性がある粒子を各粒子が予めピックアップしておこうというものである。このことにより、各質点に対して計算を行うのではなく、リストにある質点だけで、計算を行える。このピックアップ自体は $O(N^2)$ の計算量がかかるが、毎回行わず、ある一定距離 $d/2$ を移動したときにリストを更新することで計算コストを軽減している。

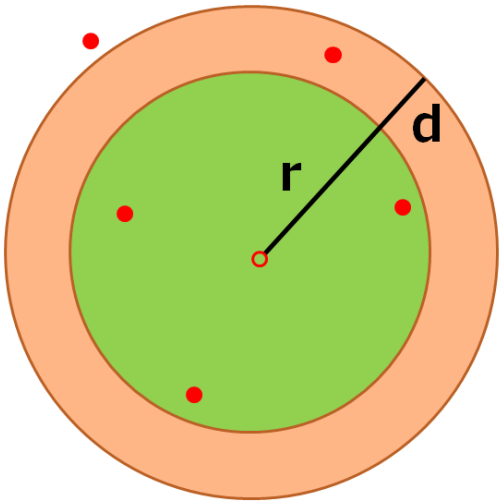

図 2.2: 粒子登録法

振る舞いの変更

今回、この N 体問題を近似せずに Java を用いて分散環境で計算させていることを考える。質点の初期座標、質量、速度などの初期データを各ホストに送り、各ステップごとに各ホストで担当する質点の計算を行わせる。このとき、各ステップが終了したあと各ホストで同期をとり、質点の座標の位置を更新し、計算が行われていく。そこで、このプログラムに先程の近似を行うことや、粒子登録法、領域分割法を取り入れようとした場合、今までの計算を放棄することなく、実行時に振る舞いの変更を行える。このように実行時に振る舞いを変更するアプリケーションの例であると言える。

近似を行うことは計算結果が変わってしまうため、あまり実用的な実行時の振る舞いを変更するアプリケーションの例とはいえないかもしれな

い。しかし、近似を行っても問題がないという状況で近似を入れることは構わないので、振る舞いを変更する例として捉える。この場合には、振る舞いを変更するユーザーがそのことについて責任をもつ必要がある。それに比べて、近似を行っているアプリケーションに対して、粒子登録法や領域分割法を取り入れることは問題は少ない。それは計算結果が変わるものでなく、各質点の計算にかかる時間が減るだけだからである。粒子登録法や領域分割法は前の説明通り、計算方法の変更を行っているわけではなく、計算を行っても近似によって影響を受けない質点に対する計算を省略しているものである。その性質はプログラムのキャッシュであると言える。

2.2　既存技術による解消法

アプリケーションの実行を止めることなく、振る舞いを変更する方法というものは、今までに多く研究されてきたが、それには制限があり変更できない部分も存在する。分散動的アスペクト指向言語では around advice に未実装や不可能な部分を抱えている処理系が多く、先程のような変更には対応できない。

2.2.1　アスペクト指向言語

オブジェクト指向では、プログラムを小さなモジュール (部品) に分割することで、保守性と拡張性を高めています。しかし、オブジェクト指向ではうまくモジュール化できない部分が存在する。処理が複数オブジェクトにまたがり、処理内容が他の関心事に強く依存している場合などである。そのような関心事を横断的関心事 (crosscutting-concern) と呼ばれ、これをアスペクト指向ではうまくモジュール化を行っている。そのため、アスペクト指向では、オブジェクト指向の欠点を補う、モジュール化技術と言える。AOP には以下のような概念がある。[21]

- ジョインポイント

  アスペクトで実装されたモジュールが、他のモジュールと結びつく接点である。例としては、あるメソッドが呼び出されたとき、コンストラクタが呼び出されたとき、フィールドに値が代入されたとき、例外ハンドラが実行されたとき、などが典型的なジョインポイントの種類である。

- ポイントカット

ジョインポイントの集合から目的のジョインポイントの集合を指定するもの。条件を指定することで、目的のジョインポイントの集合が作られ、アスペクトと結びつけるポイントを記述する。

- アドバイス

  ポイントカットで指定されたタイミングで実行する処理。

- アスペクト

  クラスやインタフェースに加えて追加されたモジュール単位。ポイントカットとアドバイスの組み合わせを指定することで、横断的関心事をまとめたモジュール。

- weave (織り込み)

  アスペクトで記述されたモジュールを統合する処理。ポイントカットで指定されたジョインポイントの集合でアドバイスが実行されるように結びつけること。

- unweave

  weave とは反対にジョインポイントに結びついているアドバイスを取り除くこと。

AspectJ

AspectJ [9, 1] はアスペクト指向言語として最も有名であり、広く使われている言語である。AspectJ は Java 言語の拡張を行い、アスペクトの機能を利用出来るようにした言語である。AspectJ で用いることができるポイントカットについて代表的なものを以下に示す。

- call ポイントカット

  メソッド呼び出し時、コンストラクタ呼び出し時。

- execution ポイントカット

  メソッド実行時、コンストラクタ実行時。

- set ポイントカット

  フィールドに値を代入する時。

- get ポイントカット

  フィールドの値を参照する時。

さらに AspectJ で用いることができるアドバイスについて代表的なものを以下に示す。

- before advice

  ジョインポイントが実行される直前に実行されるアドバイス。

- after advice

  ジョインポイントが実行される直後に実行されるアドバイス。

- around advice

  ジョインポイントの期間全体の処理の代わりに実行されるアドバイス。

around advice は他の二つのものと異なる部分が存在する。まず、戻り値の型が必要になり、ジョインポイントの戻り値の型と一致しなければならない。さらに、proceed 呼び出しがあり、around advice の内部から、元のジョインポイントの期間に対応する処理を実行することもできる。proceed 呼び出しは、そのために用意された特別な構文である。proceed の引数はその around advice の引数と、戻り値の型は around advice の戻り値の型と同じように指定する必要がある。proceed は呼び出すと、元のジョインポイントの期間に対応する処理を、渡された引数を使って実行します。同一のジョインポイントに対して、複数のアドバイスがポイントカットによって結びついている時には、proceed には、まだ実行されていない、残りのアドバイスの実行が含まれる。

```
1  public class Sample{
2    public static void main(String[] args){
3      Sample s = new Sample();
4      s.foo();
5      s.hoge();
6    }
7    public void foo(){
8      System.out.println(“foo”);
9    }
10   public void hoge(){
11     System.out.println(“hoge”);
12   }
13 }
```

図 2.3: Sample クラス

```
1 public aspect HelloAspect{
2   pointcut helloMethod():
3     execution(void Sample.hoge());
4   before(): helloMethod(){
5     System.out.println(“Hello”)
6   }
7 }
```

図 2.4: AspectJ のアスペクトクラス

ここで、AspectJ を用いた簡単なプログラムを説明する。AspectJ を用いると図 2.3 のようなプログラムに対して、図 2.4 のようなアスペクトを実装することができる。元のプログラムでは foo() と hoge() を用いて、

```
foo
hoge
```

と出力するクラスである。これに対してアスペクトの方では execution ポイントカットにより、hoge() メソッドの実行が選ばれている。そのポイントカットを用いて before advice が利用されている。このアスペクトを織り込みを行った後、Sample クラスを実行すると、

```
foo
Hello
hoge
```

と出力されるように振る舞いが変わる。hoge() メソッドが実行される直前に HelloAspect のアドバイスが実行されるのでこのような出力になる。

2.2.2　動的アスペクト指向言語

アプリケーションの振る舞いを変えて、機能追加を行いたいことなどがある。このとき、普通であれば、アプリケーションの実行を止めて、プログラムを変更した後で再度コンパイルしなおし、改めて実行する必要がある。しかし、計算量が多いアプリケーションは実行時間が長く、実行を止めることなく計算を行いたい。このようにアプリケーションの実行を止めることなく、実行中にアプリケーションの振る舞いを変更したいという要求がある。

そこで、アスペクトを実行中のプログラムに織り込むことができるようにした技術である、動的 AOP が研究された。Java の拡張で行われる代表的な動的織り込みの実現方法としてはいくつか種類がある。

Hotswap を用いた織り込み

この方法では、VM 上にロードされたバイトコードを動的に新しいバイトコードに変換することで、織り込みを実現している。java.lang.Instrument API などを用いることで Hotswap を用いたアスペクトの織り込みを行っている。通常の JVM の制限によりメソッドボディしか動的に変更できない。

この方法を用いているシステムには、HotWave [19] , Wool [16], 最も新しく出された PROSE [11] がある。java.lang.InstrumentAPI などを用いてバイトコードを操作して、アドバイスの呼び出しが行われるように変更する。HotWave では around advice をインライン展開を利用することでアドバイスを織り込んでいる。他の実装としてはアドバイスを呼び出すフックを織り込む方法もある。

フックを用いた織り込み

この方法では、ジョインポイントに対してあらかじめフックを挿入しておく方法である。このフックにより、各ジョインポイントがポイントカットとして選ばれているのかを判断し、アドバイスを実行することができる。

この織り込み方法を用いているシステムには、JAC [13] 、二番目に出された PROSE [14] がある。JAC ではクラスをロードする際に、ジョインポイントにフックを挿入している。PROSE では Jikes Research Virtual Machine (RVM) に基づき、その拡張した RVM の JIT コンパイラによってフックが挿入される。

フックを用いた織り込みにも方法がいくつかある。まず、すべてのジョインポイントにフックを挿入する方法ある。この方法では、選択されていないジョインポイントでもフックが挿入されているので、ポイントカットによって選択されているかを判断する必要があり、大きなオーバーヘッドとなる。もうひとつは、あらかじめ指定したジョインポイントのみにフックを挿入する方法である。この方法では、想定してないジョインポイントに対してアスペクトを織り込むことができない。

イベント通知を用いた織り込み

この方法では、元のコードを編集することなく、ジョインポイントで発生するようにしていたイベント通知を元にアスペクトの織り込みを行う。この織り込み方法では、イベントの発生してから元のプログラムに復帰するまでの時間的コストが大きく、オーバーヘッドが大きくなってしまう。

この方法を用いているシステムには、最初に出された PROSE [15] 、Wool [16] がある。これらのシステムは、JPDA (Java Platform Debugger Architecture) などのデバッガを用いてイベント通知するようにし、プログラムを停止させてアドバイスを実行するようにしている。

VM レベルでの織り込み

この方法では、Virtual Machine をアスペクト指向用に拡張し、動的なアスペクトの織り込みを実現している。 [6] この方法を用いたシステムには、Steamloom [8] があり、RVM[4] を拡張することで、アスペクトの織り込みが行えるようにしている。

2.2.3　分散アスペクト指向言語

計算量が多いプログラムの解決方法として、計算を分散させることで負荷を減らし、計算を速く行う方法がある。この場合、各ホストで計算を行っているため、アプリケーションの振る舞いを変更する場合には各ホストに対してアスペクトを織り込む必要がある。図 2.6 とは異なり、図 2.5 のように別のホスト上にあるジョインポイントを選択できるようになれば、各ホストにアスペクトを配置する必要がなく、アドバイスを一つのホストで実行することができ、集中管理することができる。このようなジョインポイントの選択を実現したのは DJcutter [12] の remote pointcut である。その後、AWED [10] でも利用されている。

2.2.4　分散動的アスペクト指向言語

上の二つの特徴をまとめた分散動的アスペクト指向言語も存在する。これは、分散環境で動いているアプリケーションに対して、動的な振る舞いの変更することを実現している。

このようなシステムとしては、JAC, JAsCo [17], DyRes [18] がある。JAC は遠隔ホストに対してアスペクトを織り込むことができるが、remote pointcut のような機能がない。JAsCo は AWED を用いて分散対応にしているので、remote pointcut を備えている。DyRes は織り込みやアンウィーブを安全に行うための機構を備えたシステムであり、Spring AOP [5] を拡張して実装されている。

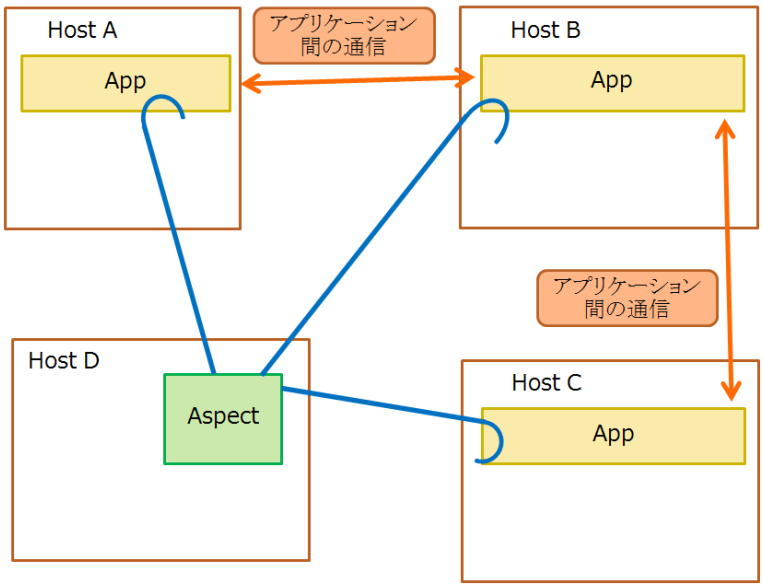

図 2.5: remote pointcut を用いたアスペクトの織り込み

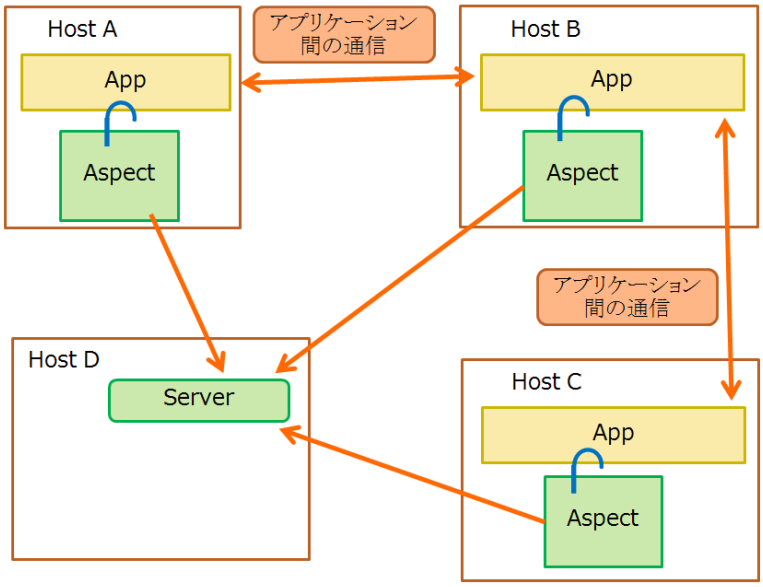

図 2.6: 通常のアスペクトの織り込み

2.3　既存技術の問題点

この研究では、Hotswap を用いた分散動的アスペクト指向言語について議論している。そこで Hotswap の制限は大きな問題になる。さらに、

remote pointcut を選択した場合には proceed 呼び出しはリモート呼び出しになるため適切な分散処理を行う必要がある。

2.3.1　Hotswap の制限

本システムでは、動的な振る舞いの変更を行うために Hotswap を利用している。メソッドのボディを退避することが必要なことであることを確認し、その方法が Hotswap の制限を受けることを示す。

メソッドボディの退避

静的にアスペクトを織り込むシステムなどでは、メソッドボディを退避することで around advice を実現する実装がほとんどである。図 2.7 のようにメソッドボディを退避することで、proceed 呼び出しはその退避先を呼び出せばよい。しかし、このようなメソッドボディの退避を行えないと図 2.8 のように、メソッドボディの退避が行わないと元々の処理内容が消されてしまう。

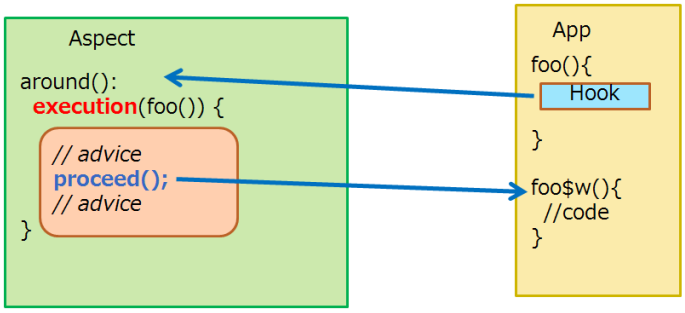

図 2.7: メソッドボディを退避した場合

図 2.9 のように、メソッドボディを退避することができる織り込み方法では、同じジョインポイントに対して織り込みが行われた場合でも有効な織り込み方法であることがわかる。同じジョインポイントに対して複数の around advice が織り込まれた場合には proceed 呼び出しはアスペクトのアドバイスを呼び出すこともある。複数回のメソッドボディの退避したが行えればこの問題は解決ができる。図 2.9 のように各 around advice が各メソッドを利用して proceed 呼び出しの実現が行える。

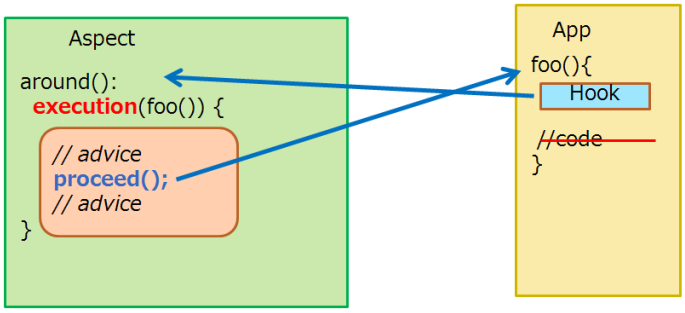

図 2.8: メソッドボディの退避を行っていない場合

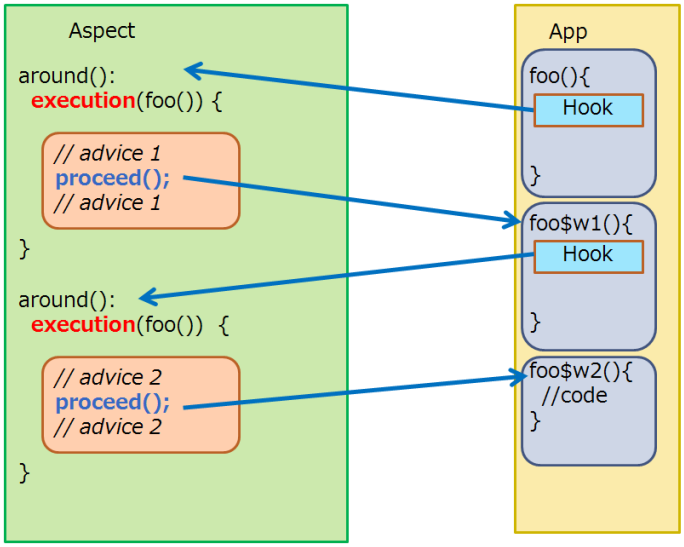

図 2.9: 複数回メソッド退避を行った例

メソッドラップ

このようなメソッドボディの退避をメソッドラップとして行う方法がある。メソッドラップを行う方法は VM レベルでの織り込みを容易にするための方法として提案されている。[7] メソッドラップを行うことで織り込み時に考慮すべき項目が減り、織り込みが簡単に行える利点があることがいわれている。コンパイル時にメソッドラップを行うことで、アスペクトを織り込みやすい環境になったクラスを用意することができ、それを実行することでコンパイル時の負担を軽減し、アスペクトを織り込むときのオーバーヘッドも小さくすることができる。しかし、around advice の実装のために利用されたものでなく、ラップの方法としても実装方法が定

まっているものではない。

制限の影響

注意すべきはメソッドボディの退避を行うことに制限が存在することである。Hotswap では動的なメソッドの作成ができず、メソッドボディの退避が行えない。従って、メソッドボディの退避が行えないので、別の実装方法を用いることが必要なる。Hotswap ではメソッドの追加やフィールドの追加ができないが、メソッドボディのみは変更できる。フックを動的に織り込む方法では、このような制限が存在しても影響がない織り込みがほとんどであり、既存の研究でも、一部の機能に制限があるくらいで実装されている。

しかし、execution ポイントカットを選択した proceed を呼ぶ around advice についてはメソッド追加ができないことが影響される。あるメソッドをポイントカットで指定した場合、そのメソッドの処理全体の代わりにアドバイスが実行される。これは、そのメソッドボディがアドバイスの呼び出しを行うフックに変換することで実現している。このような場合、around advice には proceed 呼び出しという特別な構文があるが、その内容は元々のメソッドボディである。先程の説明通りにメソッドボディがアドバイスの呼び出しに変換されると元々の処理内容は消えてしまう。

既存の技術の例としては、HotWave があげられる。HotWave ではインライン展開を利用し、around advice をエミュレートしている。この方法では around advice は before, after advice の組み合わせであるととらえ、この二つを利用しエミュレートしている。メソッドボディにフックを織り込むのではなく、アドバイスそのものを織り込み、同じメソッドにインライン展開されたアドバイス間でデータをやりとり出来るような機構の inter-advice communication を利用している。これはローカル変数をインライン展開させることで、before, after advice の間でその値をやりとりできるようにしている。しかし、HotWave では proceed 呼び出しを複数回呼び出すことなどが行えない。あくまで before, after のアドバイス間でのデータのやりとりであって、proceed 呼び出しについての制御は行うができない。よって、around advice の完全なエミュレートが出来ていない。

2.3.2　Java 標準の RMI を用いたリモート呼び出し

仮に何らかの方法で、Hotswap によってメソッドボディの退避が行えたとしても問題はまだ残っている。本システムには remote pointcut を機構として取り入れるため、Hotswap で織り込まれたフックはリモート呼

び出しを利用している。execution ポイントカットだけでなく、call ポイントカットでも proceed 呼び出しもリモート呼び出しで行われる。

そこで、Java 標準の RMI を利用して proceed 呼び出しを行うことを考える。しかし、Java 標準の RMI では動的に作成したメソッドのインタフェースが必要となり、送られてきた stub をそのインタフェースを利用し、キャストすることが必要となる。つまり、図 2.10 のように proceed 呼び出しを記述することで proceed 呼び出しがリモート呼び出しとなる。しかし、そのような記述を行うには、stub が何であるかを判断する方法を予め準備し、明確に stub をキャストする必要がある。proceed 呼び出しはポイントカットの指定の仕方により様々なメソッド呼び出しとなるため、キャストを正確に行っていたとしてもユーザーが定義するアスペクトに厳密な条件分けが必要となり、その条件分けを経て、明示的なキャストを記述する必要がある。

このような処理を記述すれば可能になるが、それは本システムが目指しているシステム構成とは異なってしまう。なので、Java 標準の RMI を素朴に用いる方法では実現困難である。

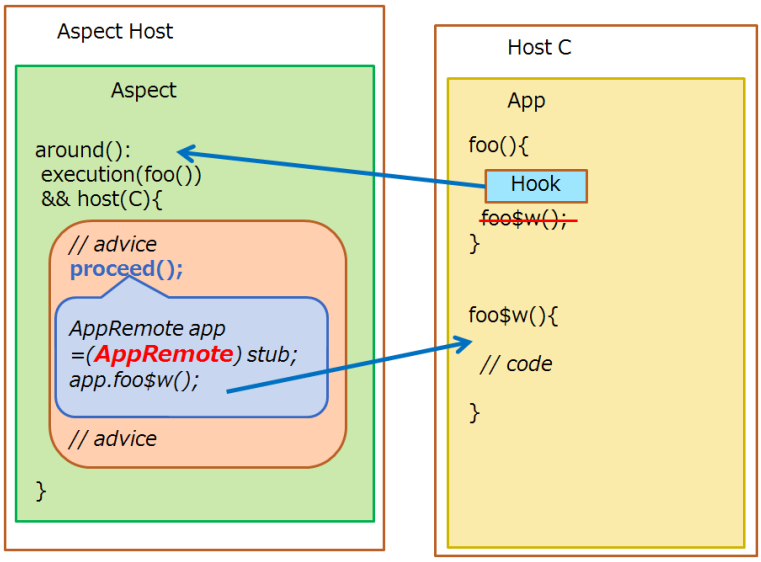

図 2.10: Java RMI を使った例

第3章　本研究での提案

Hotswap を用いた織り込み方法では、動的なメソッドボディの退避ができない。そのため、ロード時にメソッドラップを行うことで解決をする。さらに、around advice が分散環境で実現されるには proceed の呼び出しの実装方法に注意する必要がある。proceed がリモート呼び出しであること、同じジョインポイントに対して織り込まれたアドバイスがある場合、まだ実行していないアドバイスを呼び出すようにすることである。

3.1　特徴

本システムは around advice の execution ポイントカットに対応し、remote pointcut の機構を備え、同じジョインポイントに対して around advice が織り込まれた場合にも対処する。これらはフレームワークとして行われる。

２章で述べたように既存の技術では以下のような問題点がある。

- 動的な織り込みで execution ポイントカットを選択し proceed を呼ぶ around advice が利用できない。
- proceed 呼び出しがリモート呼び出しになるには素朴な実装ではできない。
- 同じジョインポイントに around advice を織り込むにはフックではできない。

3.1.1　本システムの利点

最初に問題点をあげたが、本システムを用いれば、これらの問題はすべて解決することができる。アプリケーションを作成するときにはあとから機能を追加させることを考慮せずに作成することができ、注意するべきはアプリケーションを実行する時にオプションを付ける必要があることだけである。

本システムを用いて機能を追加する際にアスペクトを作成するときには、アプリケーションがどのように実行されているかを把握している必要

がある。特に実行されているホストなども把握していることを前提にしている。しかし、これはアスペクトを作成し、織り込み、プログラムの振る舞いを変更することを考慮すれば、必要な制約である。

3.1.2　ロード時のメソッドラップ

通常、メソッドボディを退避させる方法でアスペクトを織り込む場合には同じジョインポイントに結びつく around advice の数だけメソッドを作成し、フックによる呼び出しが作られる。これと同様な振る舞いを目指してメソッドラップを行う方法では、同じフックの織り込み場所に入る around advice の数だけメソッドラップを行うことが必要とされている。そのような処理を行わないと proceed 呼び出しを行った時に必要なアドバイスの呼び出しが飛ばされてしまうことがある。これはメソッドラップのみで around advice を実装しようとしている場合である。

この研究ではこの解消法を二つに分けることで動的なメソッド作成なしに around advice を実現する。まず、一つ目としては、元々のメソッドの処理内容を退避するためにメソッドラップを行う。これをロード時に行うことで、Hotswap の制限を回避しつつ、メソッドボディの退避を行う。続いて、同じフックの織り込み場所に対して around advice が織り込まれる場合にはメソッドラップではなく、他の方法を用いて対処する。具体的には後で説明するメソッドディスパッチで独自の処理を行う。

3.1.3　RMI の実行コンテキストを含める改造

実行コンテキストが含まれることで、独自のメソッドディスパッチを行う。その理由としては、任意のアプリケーションへの対応を行うための問題と同じフックの織り込み場所に対して複数の around advice が織り込まれたときへの対応を行うためである。

remote pointcut を利用することで proceed 呼び出しがリモート呼び出しになる。従って、利用している Java 標準の RMI では、スタブをキャストする必要がある。つまり、動的に作成したメソッドに対してはクラスをロードする時に java.rmi.Remote を継承したリモートインターフェースを実装する必要がある。

proceed 呼び出しは様々なホスト、クラス、メソッドを呼び出す可能性があり、フックに対して proceed 呼び出しが一対一で対応しなければならない。そのことから、様々なキャストを行うこと求められ、それを行うことができない。

強引にキャストを行って呼び出しを行うようにしても、まだ問題が残っている。インターフェースがないと作成したアスペクトクラスがコンパイ

ルできないので、ロードされていないクラスに対してのアスペクトが作成できない。

以上の複数の問題から、Java RMI を使った既存の方法で proceed 呼び出しが行えない。従って、本システムではメソッドディスパッチを独自で行う方法を取っている。

フック

around advice のフックでは以下の三つを順に行う、実際に織り込まれるフックは図 3.1 のようなコードである。before, after advice の場合には proceed 呼び出しが行われることがないので、アドバイスの呼び出しのみを行うフックが織り込まれる。

- インスタンスをエージェントに登録
- proceed に必要な情報をアスペクトに送る
- アドバイスの呼び出し

```
1  if(index$w < 0){
2    index$w = run.AppHost.addinstance(this);
3  }
4
5  run.AppHost.sendproceed(index$w, /* methodName */);
6
7  $_ = ($r)((run.aspect.AspectRemote)run.AppHost.aspectlist.
8  get(/* aspect index */)).advice(/* aspect index */, $args);
```

図 3.1: Javassist による織り込まれるフックの記述例

まず、インスタンスの登録についてである。エージェントはアプリケーションを監視しているシステムであり、本システムそのものである。エージェントについては後で述べる。インスタンスの登録はインスタンスに対して一度行えばよい、しかし登録されているかどうかを調べるには適した方法をとらないとオーバーヘッドがとても大きい。なので、フィールドにインスタンスの登録状況を残せるようにしておく。このフィールドは Javassist を用いてクラスのロード時に追加で作成するものであり、メソッドラップと同じタイミングで行われる。追加されたフィールドは登録状況とインデックスを保持を行う。初期値は-1 に設定し、登録後は 0 以上の値をとるので、値を調べることで登録されているかを判断することができる。

proceed の準備

エージェントは proceed 呼び出しが行われる可能性がある around advice のフックが織り込まれ、フックが呼び出された時にインスタンスの登録を受ける。エージェントはアプリケーションとともに動いているシステムであり、今回は本システムである。つまり、本システムはアプリケーションの振る舞いを変更できるようにアプリケーションを監視を行っていて、このインスタンスの管理も行っている。このようにインスタンスの登録が行われた時には登録したインデックスを返し、フックが指定のフィールドに値を代入する。

フックがインスタンスの登録を終えたら proceed 呼び出しを行うのに必要な情報をおくる。インスタンスの登録が必要なのは一度だけだが、proceed の情報を送るのは織り込まれたフックの数だけ行う必要がある。proceed に必要なのはインスタンスを登録した場所を示すインデックスとメソッド名とホスト名である。これでどのフックから呼び出しが行われたかを一意に定めることができる。それに加えて、proceed がアドバイスを呼び出すか、インスタンスを呼び出すかを送る。これについては後で述べる。

ここで、フックと proceed の関係は一対一であることを確認しよう。ジョインポイントとポイントカットをつなげる方法としてフックを利用している。フックの基本的な性質としてアドバイスの呼び出しがあるが、本システムでは、proceed 呼び出しがリモートで行われる場合があるため、そのための準備が必要となることについては論じた。さら、一つのポイントカットから複数のフックが作成されるのは言うまでもない。クラスやメソッドが異なれば、その分だけ織り込まれるフックが必要となる。さらに、remoto pointcut を用いた場合にはホストによっても異なる。

さらに、問題にするのは、織り込まれたフックが同じであっても、proceed 呼び出しが異なる場合が存在する。それは複数のインスタンスが作成された場合である。同じクラスから作成されるため、織り込まれているフックは同じである。このように同じ記述によるフックでも proceed 呼び出しは異なるとして扱いたい。

ここで proceed 用の情報を管理するためにフックの判別を行いたい。必要なのは以下の三つで有る。

- ホスト
- インスタンスのインデックス
- メソッド

remote pointcut の問題点であるホストは、呼び出しを行う相手を選ぶ際に解決することができる。なので、ホストへの命令を受けるようにリモー

トオブジェクトを持っているようにする。ポイントカットでは、クラスを指定するようになっているが、それでは proceed の呼び出しの時には不足する。なので、エージェントで保存を行ったときのインデックスを用いてインスタンスの指定を行えるようにしていた、そのインデックスを用いて proceed 呼び出しにも対応できるようにする。ポイントカットによってメソッドも指定した場合には呼び出すメソッドも複数あり、選択するときに問題が生じる。メソッドごとに行えるようにしておく。

このようにフックを判別を行えるようにしておくことで、その識別のインデックスをアドバイスの呼び出しのときに引数で持たせ、さらに proceed 呼び出しのときに引数を持たせて呼び出しを行うようにすることで指定ができるようになる。

フックによる呼び出しが行われるときにその proceed 呼び出しのためのデータをアスペクトクラスが保持するするので、フックの判別を行い、存在しなかったら追加で作成が行えるようにすればよい。

アドバイスの呼び出し

図 3.1 のようにフックからアドバイスを呼び出しが行われる。Javassist の内蔵コンパイラによってコンパイルされます。このコンパイラは$ で始まるいくつかの特別な意味を持つ識別子によって、言語を拡張しています。$_ は戻り値であり、$r は戻り値の型で、これはキャストに使用される。$args はメソッドの引数をオブジェクト型の配列として受け取ることができる。図の中での aspect index はアドバイスの登録されたインデックスを入れるようにするため、フックを作る際に数値が代入される。従って、実際に織り込まれるコードでは aspect index ではなく、適した値が織り込まれる。メソッド名も同様である。

proceed 呼び出し

proceed の呼び出しを受けたとき、エージェントはリフレクション API を利用して、アドバイスやインスタンスのメソッド呼び出しを行っている。このときに proceed の準備で追加していたアドバイスかインスタンスかの情報を利用することができる。このときインスタンスか、アドバイスかによって呼び出し方法が異なる。これを、事前に準備していた情報を元に判断する。図 3.2 のようにエージェントにリモート呼び出しが行われ、そこでメソッド呼び出しが行われる。

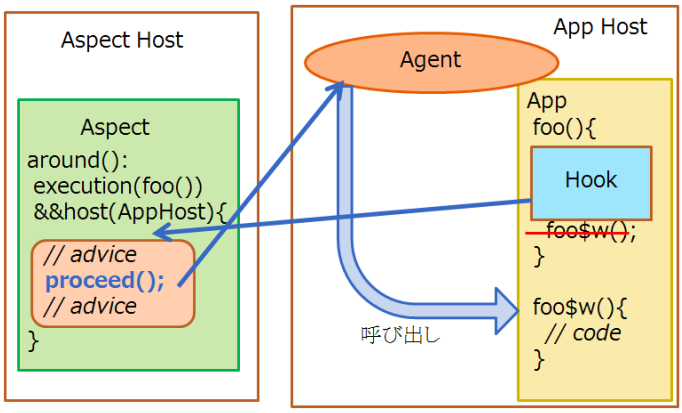

図 3.2: エージェントによる proceed 呼び出し

3.2　本システムが提供する機能のまとめ

本システムが提供するアスペクトを記述するための機構を確認する。

3.2.1　ポイントカット

本システムでは、execution ポイントカットと call ポイントカットをサポートしている。両方とも引数として、織り込むホスト名の配列、織り込み方法、クラス名、戻り値の型とシグネチャを持たせる必要がある。

3.2.2　アドバイス

本システムでは、before advice, after advice, around advice をサポートしている。around advice は、proceed 呼び出し関して制限のないものにも対応していて、remote pointcut を利用することができる。もちろん、before advice, after advice も remote pointcut が利用可能である。

ユーザーがアスペクトクラスを作成するときには本システムの Aspect クラスを継承する必要がある。Aspect クラスは下のような advice メソッドをリモート呼び出しできるようになっている。なのでユーザーが作成するときは、図 3.3 のようにオーバーライドすることでアドバイスの内容を記述すればよい。戻り値の型も引数の型もオブジェクト型であるので、キャストを行うことでどのようなメソッドに対しても対処できる。

```
Object advice(Object...object);
```

```
1  Object advice(Object...object){
2    /*
3     * advice
4     */
5    return null;
6  }
```

図 3.3: アドバイスの記述例

around advice において proceed 呼び出しを行う場合には、織り込んだフックとの一対一対応が必要になる。フックを織り込む際にインデックスを振り、それに対する proceed の情報を保存しておく。アドバイスの呼び出しはそのインデックスを引数として追加して呼び出しを行う。本システムの Aspect クラスは下のような advice メソッドもリモート呼び出しできるようにしている。around advice では図 3.4 のようなアドバイスをオーバーライドとして記述する。

```
Object advice(int index,Object...object);
```

```
1  Object advice(int index,Object...object){
2    /* advice */
3    Object o = proceed(index,object);
4    /* advice */
5    return o;
6  }
```

図 3.4: proceed 付アドバイスの記述例

第4章　実装

本システムの実装に必要な技術についてと、実際の実装方法について述べる。

4.1 Instrument API

本システムは、java.lang.instrument パッケージを用いて実装されている。このパッケージは、Java エージェントと呼ばれる Instrument API を使用するクラスが、観察される側のアプリケーションと同じ JVM 上で動作するエージェントを実現する仕組みである。そのプログラムのバイトコードにアクセスするための枠組みを提供するものである。対象のアプリケーションのバイトコードを他のバイトコードに置き換える枠組みが提供されているが、変換方法としては以下の二つがある。

- クラスがロードされる過程に割り込み、そのバイトコードをほかのものに置き換える。 (以下、「ロード時の変換」と呼ぶ)
- ロード済みのバイトコードを新しいバイトコードに再定義する。 (以下、「再定義による変換」と呼ぶ)

ロード時の変換は、クラスがロードされるときにのみ変換可能なため、動的にバイトコード変換することはできない。しかし、Hotswap ではないので、メソッドの追加やフィールドの追加が可能である。したがって、メソッドラップを行うことができる。アスペクトの織り込みは、メソッドにフックを織り込むことで行えるので、こちらは再定義による変換を用いて実装されている。

4.1.1 Instrumentation インタフェース

このインタフェースはJava コードを監視を行い、変換するための機構を利用するためのバイトコードの処理の窓口となっている。Instrumentation インタフェースが提供しているメソッドは以下のようなメソッドがある。

ロード時の変換に用いられるメソッドとしては、addTransformer, removeTransformer などがある。ClassFileTransformer オブジェクトを登

録したり、削除を行うことができる。登録された ClassFileTransformer がクラスロードの過程に割り込み、バイトコード変換の処理を行う。

再定義による変換に用いられるメソッドとしては、redefineClasses がある。このメソッドにクラスの新しい定義を渡すことでクラスの再定義し、バイトコードを変換する。また、クラスのバイトコード変換をサポートしているかどうかは、isRedefineClassSupported メソッドを呼ぶことで再定義による変換がサポートされているか否かを返す。

4.1.2　premain メソッド

Instrument API を利用するには、Java エージェントにおいて Instrumentation オブジェクトを取得する必要がある。Java エージェントを実装しているクラスは、premain メソッドを用いて Instrumentation オブジェクトを取得している。このメソッドは、以下のようなシグネチャを持つ。

```
public static void premain (String agentArgs, Instrumentaion inst)
```

Java エージェントを実装しているクラスには、このメソッドを実装しておく必要があり、監視対象のアプリケーションを実行する際にオプションを指定して実行することで、監視対象のアプリケーションの main メソッドが呼び出される前に、この premain メソッドが呼ばれるようになる。

-javaagent オプション

-javaagent オプションは、次のように指定する。

```
java -javaagent: <JAR path> = <options>
```

Java エージェントのコードは JAR ファイルにまとめることになっており、-javaagent オプションによりその JAR ファイルのパスを指定しなければならない。JAR ファイルの中のどのクラスの premain メソッドが呼び出されるかは、マニフェストファイルの属性 Premain-Class で指定する。他にも、Boot-Class-Path オプションでエージェントが必要とするクラスのクラスパスを指定することができる。さらに、Can-Redefine-Classes オプションでエージェントが再定義による変換機能を必要とするか否かを指定することができる。Can-Redefine-Classes はデフォルトでは false となっているため、本システムでは true にする。従って、本システムのコードはすべて JAR ファイルにまとめてあり、-javaagent のオプションで指定することで利用できる。

4.1.3　ロード時の変換

premain メソッドで、Instrumentation オブジェクトの addTransformer メソッドを呼び出し、トランスフォーマーを登録することでロード時の変換を行うことができる。それ以降は、アプリケーションのクラスがロードされるたびに ClassFileTransformer の transform メソッドが呼び出されるようになる。

例としては、図 4.1 のような Agent クラスを実装した場合がある。16 行目のように新しいバイトコードを返すことで、ロード時のバイトコード変換が行われる。バイトコードの変換は javassist を用いる。21 - 23 行目の isTransform で変換を行いたいクラスであるかを判断している。変換を行わない場合には 18 行目のように null を返せばよい。この例では Sample クラスを書き換える場合を示している。

```
1  public class Agent implements ClassFileTransformer {
2    public static void premain(String agentArgs,
3  Instrumentation inst){
4      Agent agent = new Agent();
5      inst.addTransformer(agent);
6    }
7
8    public byte[] transform(ClassLoader loader, String
9  className, Class<?> classBeingRedefined,
10 ProtectionDomain protectionDomain, byte[]
11  classfileBuffer) throws IllegalClassFormatException {
12     if(isTransform(className)){
13       /*
14        * create bytecode
15        */
16       return /* new bytecode */;
17     }
18     return null;
19   }
20
21   public boolean isTransform(String className){
22     return className.equals(‘‘Sample’’);
23   }
24 }
```

図 4.1: ロード時の変換例

4.1.4　再定義による変換

premain メソッドで得た Instrumentation オブジェクトで redefineClasses メソッドを呼ぶことで、クラスの再定義を行うことができる。再定義による変換は実行中にいつでも行うことが可能であるため、Instrumentation インスタンスを Java エージェントのクラスのフィールドに保持することで、このクラスを通じて書き換えをおこなうことができる。しかし、Sun VM (Version 6.0) では、メソッド内部のバイトコードしか動的に書き換えることができないという制限がある。これが Hotswap の制限である。

例としては、図 4.2 のような動的な変換を行うアプリケーションがある。これは GUI を用いてバイトコード変換する。premain メソッドでは 6 行目で動的な変換を行うための GUI を作成している。このとき引数に Agent クラスを渡すことで、AgentFrame クラスが Agent クラスを利用して、バイトコード変換を行えるようにしている。12 行目で Instrumentation インスタンスの参照がフィールドに保持される。これで Agent は動的な変換がいつでも行えるようになる。作成した GUI は 15 - 24 行目の weave メソッドを呼び出すことで、書き換えを行う。このメソッドでは、クラス名、メソッド名から変換する対象を見つけ出し、コードを元に新しいバイトコードを生成し、33 - 40 行目の処理に移行する。redefineClass メソッドでは、ClassDefinition を生成し、この配列を引数とし、Instrumentation の redefineClasses メソッドを用いることで動的なバイトコード変換が行われる。

4.1.5　Javassist を用いたバイトコード変換

Javassist は、Java のバイトコードを操作するための Java ライブラリであり、構造リフレクションの機能を持つ。これにより、クラス構造を抽象化した API があるため、バイトコードに関する知識を持たなくてもプログラマはバイトコードの変換を行うことができる。Javassist は [20] で配布されている。

図 4.3 で、Javassist を用いたバイトコード変換の例を示す。さきほどの図 4.2 の 29 - 31 行目でのバイトコード生成は、Javassist を用いると図 4.3 のように書ける。この例では、指定されたメソッドを新たに取り出し、その頭にコードを挿入するようになっている。まず、4,5 行目で引数のクラス名から、クラスを表す CtClass (compile-time class) インスタンスを生成している。9 行目では、そのクラスで定義されているメソッドの中から引数のメソッド名であるメソッドを表す CtMethod インスタンスを生成している。次に、11 - 14 行目で新しく作成するバイトコードのための CtMethod インスタンスを生成している。16 行目でコードを挿入し

```
1  public class Agent {
2    private Instrumentation inst;
3    public static void premain(String agentArgs,
4  Instrumentation inst){
5      Agent agent = new Agent(agentArgs, inst);
6      // create redefine GUI
7      AgentFrame agentFrame = new AgentFrame(agent);
8      agentFrame.setVisible(true);
9    }
10
11   public Agent(String agentArgs, Instrumentation inst) {
12     this.inst = inst;
13   }
14
15   public void weave(String className, String methodName,
16 String code) {
17     try {
18       byte[] woven = getNewByteCode(className, methodName,
19  code);
20       redefineClass(className, woven);
21     } catch(Exception e){
22       e.printStackTrace();
23     }
24   }
25
26   private byte[] getNewByteCode(String className, String
27 methodName, String code) throws NotFoundException,
28 CannotCompileException, IOException {
29     / * create bytecode */
30     return /* new bytecode */;
31   }
32
33   private void redefineClass(String className, byte[]
34 byteCode) throws ClassNotFoundException,
35 UnmodefableClassException {
36     ClassDefinition def =
37 new ClassDefinition(Class.forName(className), byteCode);
38     ClassDefinition[] defs = new ClassDefinition[] { def };
39     inst.redefineClasses(defs);
40   }
41 }
```

図 4.2: 再定義による変換例

ている。18 行目で元の CtMethod インスタンスと新しい CtMethod インスタンスを入れ替えて、最後にクラス全体を toBytecode メソッドでバイトコード配列にして返している。

```
1  private byte[] getNewByteCode(String className, String
2  methodName, String code) throws NotFoundException,
3  CannotCompileException, IOException {
4    ClassPool cp = ClassPool.getDefault();
5    CtClass cc = cp.get(className);
6    if(cc.isFrozen()){
7      cc.defrost();
8    }
9    ctMethod cm = cc.getDeclaredMethod(methodName);
10
11   ClassPool newCp = new ClassPool();
12   newCp.appendSystemPath();
13   CtClass newCc = newCp.get(className);
14   CtMethod newCm = newCc.getDeclaredMethod(methodName);
15
16   newCm.insertBefore(code);
17
18   cM.setBody(newCm, null);
19
20   return cc.toBytecode();
21 }
```

図 4.3: Javassist によるバイトコードの変換例

4.2　Java RMI

ここでは、本システムでは Java RMI を利用している。Java RMI の利用方法として制限がある部分があり、その特徴的な部分を解説しておく。

4.2.1　概要

Java Remote Method Invocation (Java RMI)[3] は Java アプリケーション環境で動作させるために設計されたもので、Java 言語の RMI システムは、Java 仮想マシン (JVM) から構成される同種環境を想定しているため、このシステムでは Java プラットフォームのオブジェクトモデルの特性を最大限に活用できるようになっている。

Java 言語では、通信のための基本的な機構としてソケットをサポートしているが、クライアントとサーバは情報交換メッセージを符号化と復号化を行うためにアプリケーションレベルでのプロトコルに集中しなければならず、プロトコルの設計が複雑になり、エラーを招きやすい。そこで RPC (リモートプロシージャ呼び出し) があり、通信のインターフェースをプロシージャ呼び出しのレベルまで抽象化している。しかし、分散オブジェクトシステムに適合させるには難点がある。

4.2.2　インターフェースとキャスト

リモート呼び出しを行うためには、java.rmi.Remote クラスを直接または間接的に継承したリモートインターフェースを作成する必要がある。リモートインターフェースはリモート Java 仮想マシンから呼び出される可能性があるメソッドのセットを宣言するです。リモートインターフェース内で各メソッドの宣言が行われ、throws 節に java.rmi.RemoteException が含まれる必要がある。

リモートオブジェクトのスタブは、リモートオブジェクトに対するクライアントのローカルでの代理、つまりプロキシの役割を果たします。クライアントは、リモートオブジェクトのクラスが定義するのと同じセットのリモートインタフェースを持ったスタブ (代理) オブジェクトに対して呼び出しを行います。スタブは、リモートオブジェクトクラスと同じリモートインタフェース群を実装するため、スタブはサーバオブジェクトの型のリモート部分と同じ型を持つことになります。このため、クライアントは、Java 言語の組み込み演算を利用してリモートオブジェクトの型チェックや、あるリモートインタフェースから別のインタフェースへのキャストを行うことができます。リモート呼び出しを行うにはスタブをキャストすることが必要となる。

4.3　本システムの設計

本システムは、Instrumentation API を使用しているクラスの Agent クラスを Java RMI のリモートオブジェクト呼び出しを用いて呼び出し、分散動的なバイトコード変換を行っていない。バイトコード生成は Javassist を用いて行っている。以下、本システムの実装方法について述べる。

4.3.1　本システムの使用方法

本システムを利用したアプリケーションに対するアスペクトの織り込みがどのように記述できるかを示す。アスペクトの記述例としては図 4.4 の

ようなに行える。

```
1  public class HelloAspect extends Aspect {
2    pubic HelloAspect(String name) {
3      /* unweave のために名前を付ける*/
4      super(“HelloAspect”, name);
5
6      /* ホストの指定*/
7      String[] hosts = { /* chiba100, chiba101... */};
8
9      /* アスペクトの織り込み方法*/;
10     WeavePosition wp = /* control app aspect */
11
12     /* Execution, call ポイントカット*/
13     Pointcut pc = new Execution(hosts, wp, /* className */,
14 /* 戻り値の型とシグネチャ*/);
15
16     /* before, after, around アドバイス*/
17     Advice advice = new Before(pc);
18
19     /* 織り込むフックの作成*/
20     Hook hook = new Hook(advice);
21     setHook(hook);
22   }
23
24   public Object advice(Object... objects) {
25     System.out.println(“Hello”);
26     return null;
27   }
28
29   public static void main(String[] args) {
30     /* 引数によって unweave のための名前を指定する*/
31     HelloAspect helloaspect = new HelloAspect(args[1]);
32     /* 引数による指定で weave か unweave を行う*/
33     aspect.run(args[0]);
34   }
35 }
```

図 4.4: 本システムによるアスペクトクラスの例

ホストの指定

分散環境であるので、織り込む対象のホストを指定する。アスペクトを保持するホストでなく、アスペクトを織り込まれるホストであるので、注

意する必要がある。String の配列を用いて直接ホスト名を記述することもできるが、本システムでは他の指定の仕方を用意している。形式にしたがったテキストファイルから文字列を読み込み、ホスト名の配列を作成するような機能を作った。

アスペクトを織り込む際にすべてのホストに対して織り込みを行いたいことが多い。その時にすべてのホスト名を毎回記述することは非常に手間がかかる。従って、”ALL” と記述するだけで、すべてのホスト名の配列の代わりを担うことができれば、記述が簡単になり、ホストの指定するときに記述漏れが起こるなどのミスが減る。

ここではアスペクトの織り込み命令を行うホストをコントロールホスト、アプリケーションが実行されているホストをアプリケーションホストと呼ぶこととする。すべてのアプリケーションホストをどのように把握するかがここでの問題となる。コントロールホストが、実行しているアプリケーション状態を把握していれば、すべてのアプリケーションホストを指定することは簡単である。把握の仕方として、二つの方法を示す。一つはアプリケーションホストからコントロールホストに通知を行い、コントロールホストはその通知を受け、アプリケーションホスト一覧を作成する。もう一つはコントロールホストを利用するユーザーがすべてのアプリケーションホストについて把握していて、ユーザーが一覧を作成する。

アプリケーションホストからコントロールホストへ通知を行う方法について。アプリケーションを実行するときにはアスペクトの織り込みに関して、-javaagent のオプションを付けることのみ考慮する。従って、アプリケーションの実行時にコントロールホストが機能してなくてもよい。このことからアプリケーションホストは起動後コントロールホストへの通知が行われるまで、通知を送り続ける必要がある。もしそのような方法を採用した場合には、かなりのオーバーヘッドがかかってしまう。

従って、本システムではコントロールホストを利用するユーザーがアプリケーションホスト一覧を作成する方法を使う。ユーザーが一覧を作成するので、すべてのホスト名というグループというようにとらえることもできる。従って同じ方法を利用して一部のホストをグループとしてまとめることができる。テキストファイルには図 4.5 のような形式でホストのグループを作成する、下の例ではすべてのアプリケーションホストが chiba100 から chiba102 の場合である。

アスペクトでホスト名を指定するときにホスト名を直接指定しているのか、ユーザーが作成した一覧を利用するかを判断するために String の配列の先頭を”user” として、二つ目をグループ名となるようにする。図 4.6 はホスト一覧の利用方法を示したものである。図 4.6 の一行目のようなユーザーが作成した一覧を利用しすべてのホストを指定している場合で

```
1  {ALL/* グループ名*/,
2  /* ホスト名一覧*/
3  chiba100.intrigger.nii.ac.jp,
4  chiba101.intrigger.nii.ac.jp,
5  chiba102.intrigger.nii.ac.jp
6  }
7  {chiba100/* グループ名*/,
8  /* ホスト名一覧*/
9  chiba100.intrigger.nii.ac.jp
10 }
```

図 4.5: ホスト名のテキストファイルの記述方法

ある。さらにローカルホストを指定する場合には、図 4.6 の二行目のように指定すればよい。

```
1  String[] hosts = {"user","ALL"};
2  String[] hosts = {''local''};
```

図 4.6: ホスト名一覧の利用方法

この機能によってホストの指定がグループ単位で行えるようになっている。

アスペクトの織り込み方法

アスペクトを織り込むにはどこかのホストがアスペクトのインスタンスを保持している。図 4.7 ように三通りの方法があり、以下のような方法である。

- 織り込まれる側にインスタンスがある場合
- 織り込み側にインスタンスがある場合 (remote pointcut)
- 第三者のホストにインスタンスがある場合 (remote pointcut)

本システムは remote pointcut の利用を選択することもできるため、同じ織り込むホストと同じ織り込まれるホストであっても、二通りある。さらに第三者のホストにアスペクトのインスタンスを持たせるように織り込むこと機能も作成した。これを行うにはコントロールホストからの織り込

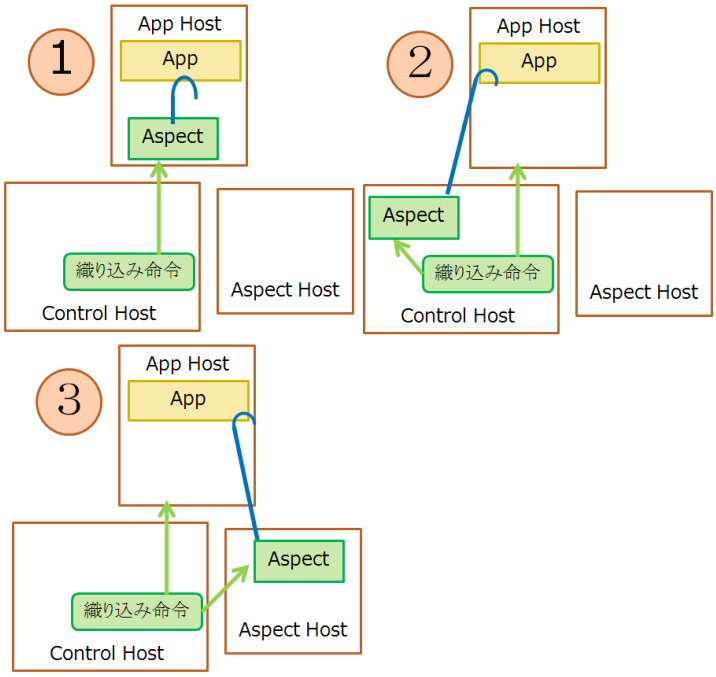

図 4.7: アスペクトの織り込み方法

み命令を受けるための準備をしておく必要がある。第三者のホストはアスペクトを織り込むユーザーが実行させておく。従って、織り込み命令が届かない場合を想定する必要はない。

この機能により、織り込みをコントロールするホストとアドバイスが実行されるホストとアプリケーションが実行されるホスト、この三種類に分けることができるようになっている。

インスタンスに名前を付ける

アスペクトを織り込む時には、そのクラスのインスタンスを生成し、weave メソッドを呼び出すことで行われる。同じアスペクトを何度も織り込んだときには各インスタンスを保持していなければ、各インスタンスの区別を付けることができない。従って、このままではクラス単位での unweave しか行えない。

そこで織り込む時に名前を付けることで区別を付けることができるようにする。アスペクトを unweave したい時には、そのクラスの unweave メソッドを呼び出すことで行える。引数に名前を持たせることでインスタンスの判別を行う。今まで通りにクラス単位での unweave を行いたいとき

には名前に ”ALL ”を指定することで行えるようにした。なので、アスペクトを織り込むときに付ける名前には ”ALL ”を避けなければならない。別な利用法としては、これから織り込むアスペクトを unweave するときには必ず同じクラスのアスペクトも unweave したいときにはあえて名前に ”ALL ”を指定する方法をとることもできる。このような利用方法はアスペクト作成者が判断して利用することができる。

4.3.2　weave

本システムでは、アスペクトのインスタンスやスタブを受け取り、それに対してメソッド呼び出しを行うためにリストにして管理しておく。フックでは管理しているアドバイスに対しての呼び出しを織り込むことでアスペクトの織り込みが行える。順序良くアドバイスが実行するには、順番通りにフックの織り込まれるようにすることで before, after advice は解決できる。

フックの置き換え

around advice はフックを織り込む場所が同じものに対してフックを織り込む方法だけでは行えない。アスペクトの織り込まれ方を考えると図 4.8 のようになる。これは最初の状態で Aspect A が foo() メソッドの execution ポイントカットに対して around advice を織り込まれている場合である。そこに Aspect B が織り込まれた場合である。すでに織り込まれているアドバイスも含めたものが Aspect B の proceed にあたる。従ってまずは Aspect B のアドバイスが実行されて、proceed 呼び出しで Aspect A のアドバイスが実行され、さらに proceed 呼び出しでラップされた元々のメソッドの内容が実行される。

しかし、これから織り込むフックの呼び出すアドバイスが around であり、フックを織り込む先にすでに around advice のフックが織り込まれていた場合には、フックの置換えが必要になる。順序よくアドバイスを呼び出していく場合には、一番最後に織り込まれたアドバイスのフックが実際のアプリケーションに織り込まれ、織り込まれたアドバイスで proceed の情報として今までそこに入っていたフックに対応するアドバイスの情報が送られる。

proceed に関する情報は次の呼び出しを覚えている方法で、リンクドリストのようになっており、それに従って proceed 呼び出しが行われ、アドバイスが実行される。このつながりを作る方法として二つの方法が考えられる。

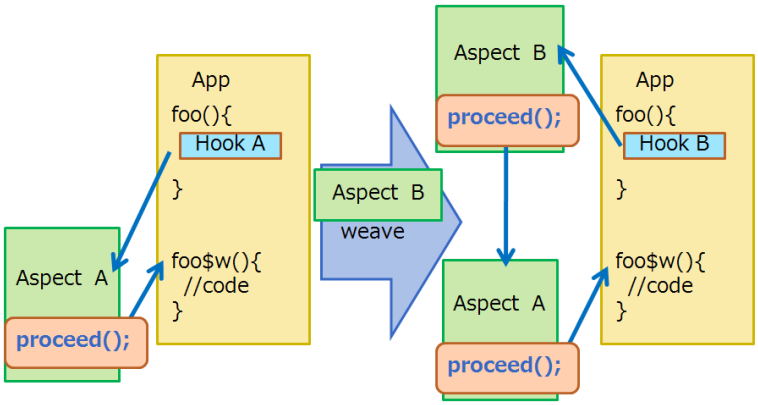

図 4.8: around advice の織り込まれ方

- 図 4.8 のようにすでにリストが作られているところにアスペクトを織り込む方法
- フックによる呼び出しで、最初から順にリストを作成する方法

純粋に考えれば、前者の方法を選択してしまうが、その方法では解消できない場合が存在する。それは、インスタンスが作成されていない状況で、フックの織り込み場所が同じ場合である。織り込みが終わった後インスタンスが作成された場合には織り込まれているフックは一つである。しかし、アスペクト自体は複数ある可能性がある。このような場合には proceed の設定がなされていないので、フックによる呼び出しでリストを作成する必要がある。フックによる呼び出しが行われた時には、フックの情報から同じフックのアスペクトを取り出し、織り込まれたのと逆順にアドバイスが実行されるので、インデックスを元にリンクドリストになるように proceed の情報を設定する。

proceed 呼び出しがアドバイスの呼び出しの場合には、アスペクトが織り込まれた時に登録した時のインデックスを持たせ、アドバイスの呼び出しであることを送る。

未ロードのクラスに対する織り込み

織り込み対象のアプリケーションのクラスがロードされていないときは織り込むことができないので、ロードを行いメソッドラップを必ず行う。通常より早くロードが行われるが、実行されたりするわけではないので、これを行うことによる問題はない。

4.3.3　unweave

アプリケーションの元々のメソッドを用意し、必要なアスペクトを織り込み直す。まずはアスペクトのフックがどこに織り込まれるべきかを特定し、その織り込み場所に織り込まれているアスペクトをピックアップし、織り込みしなおす。織り込みをやり直す場合にはそのメソッドに対する部分のみを行うようにすることが必要である。もちろん、weave で考慮した実装方法を今回も利用してポイントカットの一部の織り込みをやり直す必要がある。

図 4.9 まずはフックが織り込まれていないメソッドを用意し、メソッドラップは行う。Javassist を利用した独自に作成したメソッドなので、それを利用してメソッドラップやインスタンスの登録を行っているかを示すフィールドを作成する。あとは Javassist を用いたバイトコード変化する利用しておこなう。

```
1  /* アプリケーションの元々のクラスを取り出す*/
2  ClassPool newcp = new ClassPool();
3  newcp.appendSystemPath();
4
5  ClassPool defcp = ClassPool.getDefault();
6
7  CtClass newcc;
8  try {
9    /* 利用するクラスを取り出す*/
10   newcc = newcp.get(className);
11   CtClass defcc = defcp.get(className);
12
13   /* newcc に含まれるメソッドをラップする*/
14   /* newcc にフィールドを追加する*/
15
16   /* 変更する method を取り出す*/
17   CtMethod newcm = newcc.getDeclaredMethod(methodname,
18 params);
19   CtMethod defcm = defcc.getDeclaredMethod(methodname,
20 params);
21
22   /* 定義されているメソッドを新しいものに置き換える*/
23   defcm.setBody(newcm, null);
24   agent.redefine(className, defcc);
25
26   /* 必要な before, after アドバイスのフックを入れなおす。*/
27   /* around アドバイスのフックを入れる*/
28 } catch (NotFoundException e) {
29   e.printStackTrace();
30 } catch (CannotCompileException e) {
31   e.printStackTrace();
32 }
```

図 4.9: unweave の概要

第5章　実験

InTrigger [2] を用いて通信を行い、その通信を行っているアプリケーションに対するオーバーヘッドを計測した。

5.1　実験環境

本システムの性能を計測するため、分散環境を構成する必要がある。分散環境として巨大な計算資源を提供している InTrigger を利用した。

InTrigger とは、情報爆発時代に向けた新しい IT 基盤技術の研究のための分散プラットフォームである。それは、日本中に分散されたクラスタのクラスタである。並列処理、分散プログラム、ネットワーク、信頼性と復旧性、セキュリティなどのシステムソフトウェア開発者。自然言語処理、データマイニング、ウェブマイニング、ウェブ分析などの様々なタイプの大規模情報処理の研究者。その二つのどちらにも対応するシステムである。InTrigger は、これらの広い分野の研究者が容易に協力して作業できるプラットフォームを確立している。

プログラム　ソケットを用いて、通信を行うアプリケーション

コンパイラ　JDK 1.6.0

マシン　InTrigger 国立情報学研究所の chiba 、公立はこだて未来大学の mirai 、東京大学の hongo を利用する。

実行環境

表 5.1: 実行環境

Site	OS	CPU	Memory
chiba	Debian	Core2Duo 2.13GHz	4GB
mirai	Debian	XeonE5410 2.33GHz	16GB
hongo	Debian	Core2Duo 2.13GHz	4GB

5.2　本システムを用いることによるオーバーヘッド

本システムは、-javaagent オプションを利用して、JAR ファイルを指定して実行を行う。ここで指定する JAR ファイルは本システムを利用する。このような Instrument API の Hotswap を用いてフックを織り込むことでアスペクトの織り込みを行っている本システムは、-javaagent を利用したことによるオーバーヘッドとメソッドラップを行うことによるオーバーヘッドがある。ここではその両方を計測する実験を行った。

5.2.1　javaagent

ソケットを用いた通信アプリケーションの一回ごとにかかる時間を、-javaagent をつけることで本システムを利用した場合とそうでない場合でそれぞれ 500 回分を計測した。結果は、表 5.2 と図 5.1、図 5.2 のような結果となった。結果からわかるように javaagent によるオーバーヘッドは約 3% である。これは他の織り込み方法が数十% であることに比べると Hotswap のオーバーヘッドが小さいことがわかり、本システムのオーバーヘッドはないと言える。

表 5.2: javaagent によるオーバーヘッド (ms)

	平均	標準偏差
オプションなし	30.304	0.88971
オプションあり	31.484	0.823252

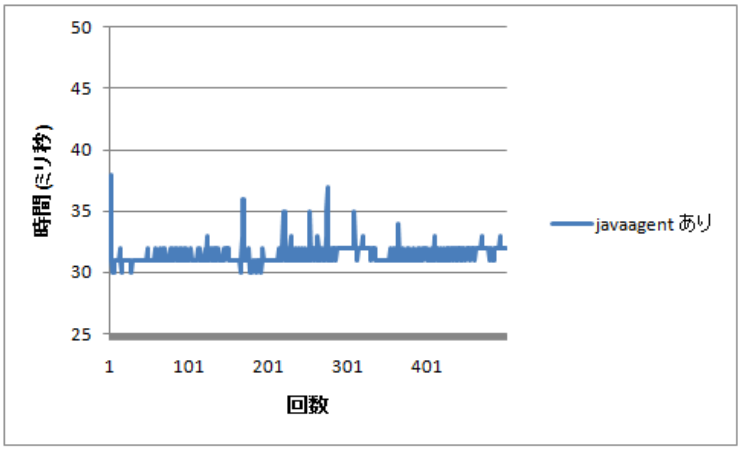

図 5.1: javaagent オプションを付けた場合

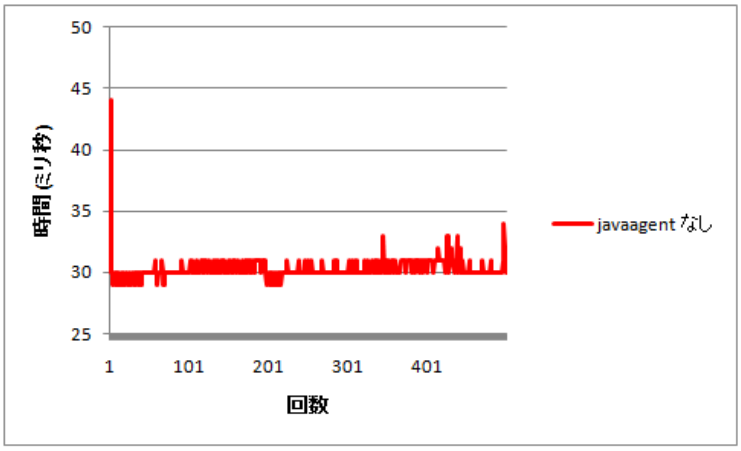

図 5.2: javaagent オプションを付けなかった場合

5.2.2　ロード時のメソッドラップ

Javassist を用いたメソッドラップを行っている。必要がないクラスを除外することで効率的なメソッドラップを行えるようにしているが、そのオーバーヘッドは計測する必要がある。特に大規模なアプリケーションに対するメソッドラップには時間がかかる。まず表 5.3 はメソッドラップを行わないクラスに関するデータである。これは条件分岐によってメソッドラップを行わないがその条件分岐にかかる時間を計測した。計測誤差もあることから、ラップを行わないクラスに関するオーバーヘッドはないと言える。

表 5.3: メソッドラップによるオーバーヘッド

	平均 (ms)	標準偏差 (ms)
ラップを行わないクラス	0.565644	0.393824
	サイズ (KB)	時間 (ms)
アプリケーションのクラスファイル	59	171

5.3　織り込みなどのオーバーヘッド

次に、本システムを用いたアスペクトの織り込みによるオーバーヘッドについて計測を行う。ソケットを用いた通信アプリケーションのメッセージに対してメッセージの追加を行うようなアプリケーションを実行するこ

とで、一回のメッセージ通信にかかる時間を計測した。メッセージの通信とメッセージの追加を分けて記述したプログラムを用意し、メッセージの追加はアスペクトで記述し、あとから織り込むことで作成したプログラムと、元々合わせて記述したプログラムを用意した。この二つのプログラムを利用して比較する。比較の条件として以下のような設定をする。

- 予めメッセージの追加を行っているアプリケーションを javaagent オプションなしに実行した場合 (図 5.3)
- 上記のアプリケーションを javaagent オプションを付けて実行した場合 (図 5.4)
- メッセージの通信のみを記述したアプリケーションを実行し、メッセージの追加を記述したアスペクトを織り込み実行した場合 (図 5.5)
- 上記のアスペクトを削除した場合 (図 5.6)

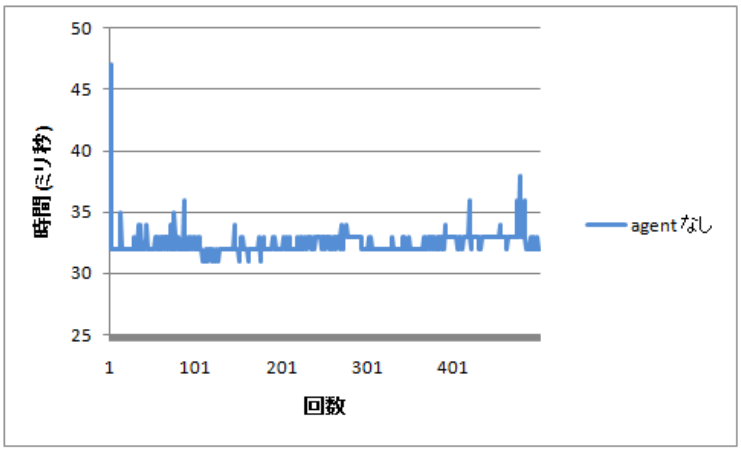

図 5.3: javaagent オプションを付けていないアプリケーション

比較を行うため、同じ振る舞いをする部分である 101 回目から 199 回目で行う。その結果は表 5.4 である。

まず、javaagent による影響を比較するため、最初と二番目を比較する。これは本システムを用いることによるオーバーヘッドで論じた通りのオーバーヘッドである。

続いて、二番目と三番目の結果を比較することで、すでに考慮してメッセージの追加を実装しているプログラムとフックを用いた呼び出しを織り込む場合を比較することができる。完全に同じ振る舞いではないが、そのプログラムの振る舞いは基本的に同じあり、オーバーヘッドはないとと

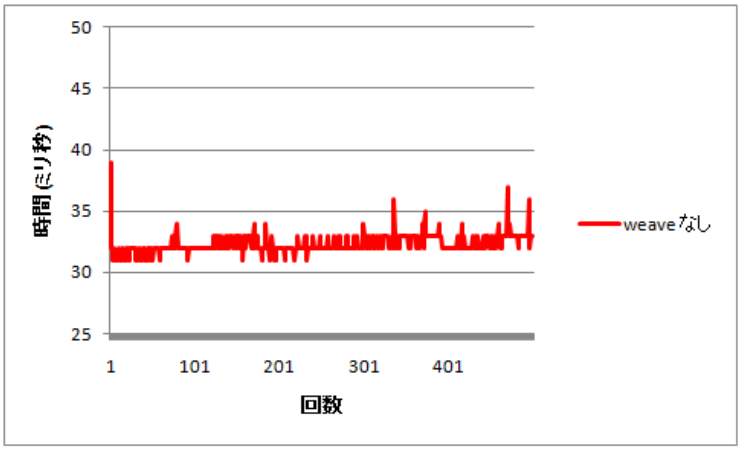

図 5.4: javaagent オプションを付けたアプリケーション

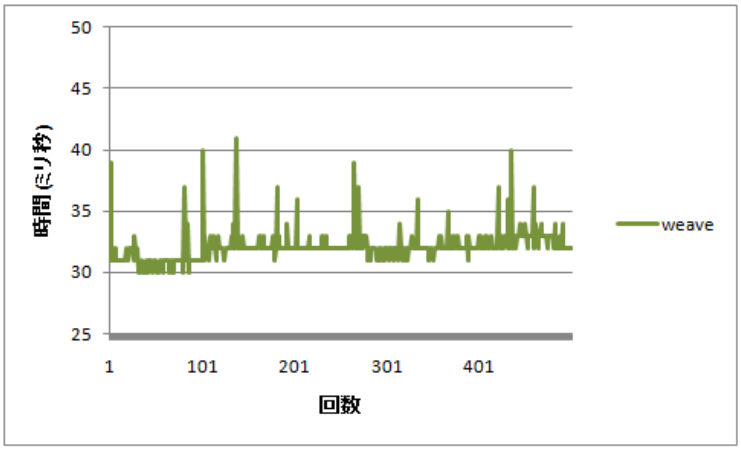

図 5.5: 途中で織り込みを行ったアプリケーション

らえて良い。しかし、注目すべきは織り込み時のオーバーヘッドである。フックの織り込みなどのバイトコード変換が行われるため、遅くなっている。

さらに、三番目と四番目の結果を比較する。ここでは、unweave がポイントとなる。unweave するときにフックの除去の処理がかかり、多少遅くなっている。さらに、フックを取り除いたあとは織り込む前の振る舞いと同じなので、オーバーヘッドがなくなっている。

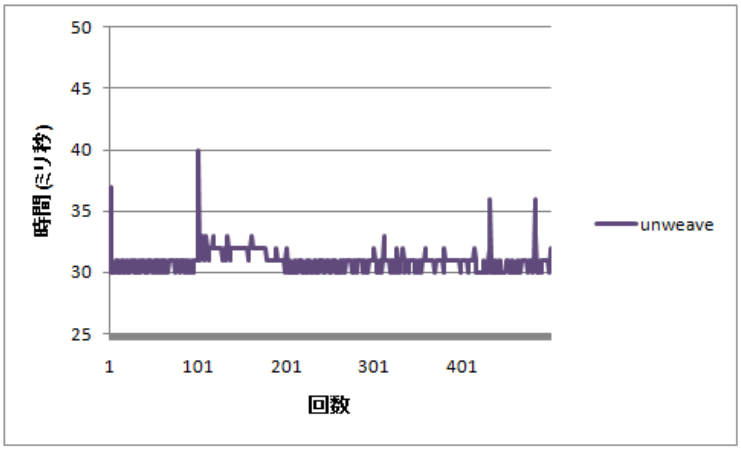

図 5.6: 織り込み後に unweave を行ったアプリケーション

表 5.4: 織り込みによるオーバーヘッド (ms)

	平均	標準偏差
agent なし	31.9898	0.524791
agent あり	32.344	0.61606
aspect weave	32.114	1.36982
aspect unweave	30.954	0.986002

5.3.1　remote pointcut と proceed 呼び出し

本システムでは、remote pointcut の機構をとりれているため、proceed 呼び出しがリモートで行われることがあり、それによるオーバーヘッドについて考察する。当然ながら、リモート呼び出しが行われることによって一つの通信で行っていたところが Java RMI によって行われる通信が増えるため極端にパフォーマンスは落ちるように見える。しかし、必要な通信を行っているためである。従って、遠くのホストに対する remote pointcut であれば遅延は大きく、近くのホストに対する remote pointcut であれば、遅延は小さい。その結果が図 5.7 図 5.8 図 5.9 である。これは 100 回目でアスペクトを織り込み、300 回目で織り込んだアスペクトを取り除いている。織り込み時や削除する時にはフックの設定があるため、かなりの時間がかかり、オーバーヘッドとなっている。

ここで、図 5.7 についての詳細を述べる。まず、chiba と mirai がそれぞれサーバとクライアントになり通信を行っているプログラムである。アスペクトの織り込み命令を mirai が行い hongo にあるアスペクトを利用

して mirai で動いているアプリケーションに対して織り込みを行っている。従って、mirai から織り込み命令が送れれ、hongo のアスペクトから mirai のアプリケーションに対して織り込み命令が送られる。そのため、織り込みの命令が 100 回目で出されているが、スレッド化を行っているので、9 ms のロスで織り込み命令を送るようになっている。通信のラグがあるため反映されているのは 129 回目であり、あとは実際のバイトコード変換にかかるオーバーヘッドがある。unweave を行うときは織り込まれているアプリケーションに対して行われるため、通信による誤差がなく削除が行われる。フックを取り除き登録していたアスペクトを除外を行う。このアスペクトを除外するかどうかを判断するために通信が行われるため、オーバーヘッドが大きい。

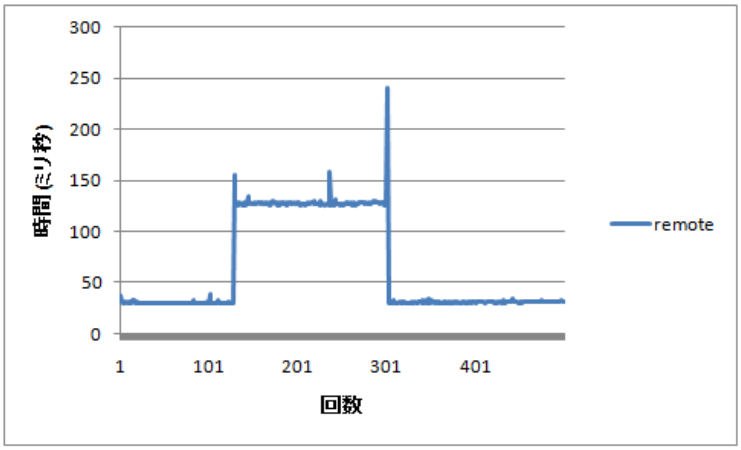

図 5.7: hongo のアスペクトが mirai にあるジョインポイントを選択

さらに、図 5.8 についての詳細を述べる。こちらは先ほどとは異なり、サーバとクライアントが逆になっている。アスペクトの織り込み命令を chiba が行い hongo にあるアスペクトを利用して chiba で動いているアプリケーションに対して織り込みを行っている。従って、chiba から織り込み命令が送れれ、hongo のアスペクトから chiba のアプリケーションに対して織り込み命令が送られる。従って、さっきほどのものより通信にかかるオーバーヘッドが減り、織り込みが反映されるまでの時間が短い。

最後に図 5.9 についての詳細を述べる。これは二番目と同じように chiba がクライアントで mirai がサーバとなって通信を行っているアプリケーションである。これに対して、通信によるオーバーヘッドが少ないように chiba から chiba に対して織り込みを行う。同じサイトであるだけであり、ホストが異なるようにしているので、アプリケーションやアスペクトの織り込み方法は同じである。織り込みが行われるまでの差がほとんどない。

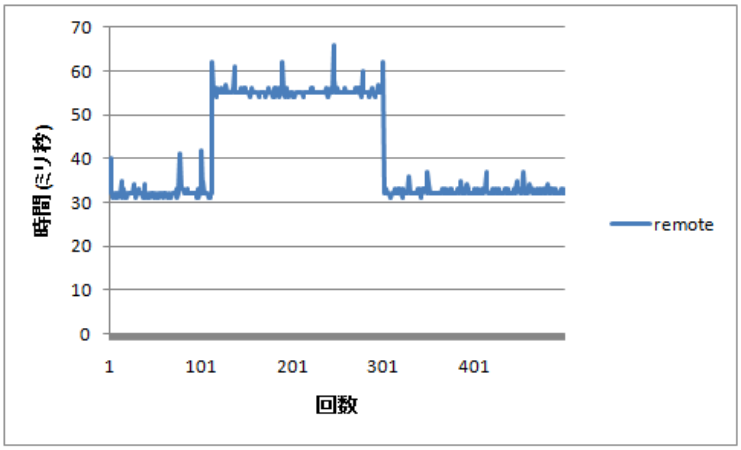

図 5.8: hongo のアスペクトが chiba にあるジョインポイントを選択

このような結果からわかるように、ホスト間の通信に影響する部分が大きいが本システムによるオーバーヘッドでなく、通信に必要なコストである。従って、不必要な通信を取り除くような実装を行うことが、オーバーヘッドを大きく削減できるとわかり、そのような実装が望まれる。

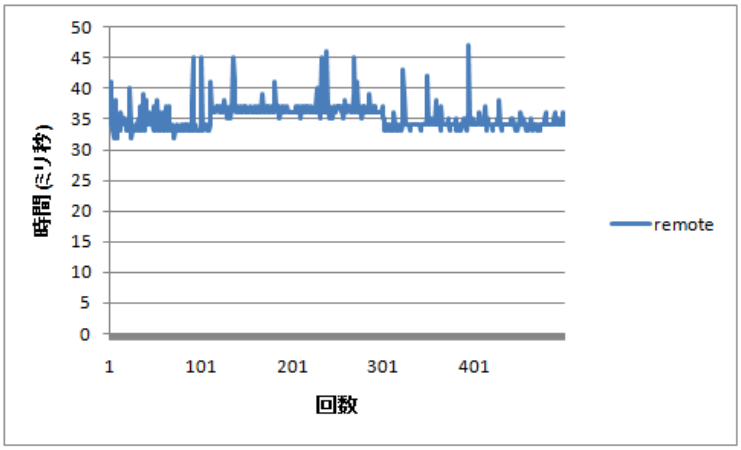

図 5.9: chiba のアスペクトが chiba にあるジョインポイントを選択

第6章　まとめと今後の課題

6.1　まとめ

本研究では、分散動的 AOP における around advice の実装方法を提案した。

分散動的 AOP では、around advice について制限をかけているシステムが多く、利用することができない。DAOP では Hotswap の制限によって実装が困難になっている。新たにメソッドをラップすることができず、proceed 呼び出しを行うことができない。分散 AOP では remote pointcut が利用できるが、そのことによって proceed 呼び出しがリモート呼び出しになり、必要な処理が多くなる。その他に、同じジョインポイントに対してアドバイスの呼び出しを行いたいような場合、フックを複数織り込むことで before, after advice は実現できるが、around advice の場合はそのような実装では実現できない。

本システムでは、ロード時のメソッドラップを行い、元々のメソッドの処理内容を退避することができた。メソッドラップの解決方法の一部採用しているが、残りの部分は本システムでの RMI の実行コンテキストを含めた改造によるメソッドディスパッチを行うことで解決している。まず、リモート呼び出しをラップを行ったメソッドに対して行うことができないため、一度本システムのエージェントが受け取り、そこでリフレクション API を用いて呼び出しを行うようにしている。さらに around advice が同じジョインポイントに対して織り込まれている時には、アドバイスの実行される順序を考慮し、その順序通りに proceed 呼び出しが行われるようにすることが必要であることを確認し、proceed 呼び出し自体は必ずエージェントに対して行われるので、エージェントが proceed の情報を元にメソッド呼び出しを行うようにしている。

実験では本システムでのオーバーヘッドを計測することができた。ロード時のラップ、動的なアスペクトの織り込みや unweave に関してはオーバーヘッドが多少なりともかかることがわかった。それよりも通信に必要な部分でのコストは大きかったため、それによって織り込みや unweave にかかるオーバーヘッドが大きくなることがわかった。さらに実験の結果から織り込まれたコードによるオーバーヘッドは少なく実装できたことがわかった。

6.2　今後の課題

本システムにはまだ解決されるべき問題が残っている。

6.2.1　現在のシステムの拡張

現在のシステムでは不完全な部分は多く残っている。他のシステムで実現されていない部分に関する実装方法について行ってきたが、他のシステムで実現されているが本システムでは実現されていない部分も多い。アスペクトは複数のアドバイスを持てるようにするなどより機能を拡張することで、よりよいモジュール化を行うことができるシステムにすることが今後の課題である。

大規模な分散アプリケーションを実行させた時に、ロード時におけるメソッドラップのオーバーヘッドや利用するマシン数が増えた場合にアクセス集中による問題が発生する可能性もある。既存の研究として織り込みの一貫性を保つ織り込み手法を取り入れることも今後の課題である。

6.2.2　他の around advice の実装方法の提案

本システムではメソッドラップにかかる時間がかかってしまう。従って、メソッドラップの方法を変えることや RMI の改良を検討して本システムの性能を上げる余地がある。しかし、より飛躍的な改良を行うには根本的な実装方法を変更する必要があり、新たな around advice の実装方法を取り上げる必要がある。メソッドラップを行わない方法を用いる around advice の実装方法として動的プロキシを用いる方法や呼び出し側での変換を行う方法などが提案としてある。

動的プロキシを用いたラッパークラスでの実装

本システムではメソッドラップを Javassist を用いて行ったが、動的プロキシを用いた方法でラッパークラスを作成することができる。この方法ではメソッドを判別する時のオーバーヘッドが大きいことが予想される。

呼び出し側の変換による実装

呼び出される側である execution ポイントカットを考える際に call ポイントカットとの違いが問題となる。この方法での実装での問題点はいくつか上げられて、本システムとの実装方法と比較したところ、他のシステム

も取り入れられているメソッドラップを分散動的環境で利用できるように拡張することを選んだ。

6.2.3　各 around advice の実装方法の性能比較

先程のように実装方法としていくつかの選択肢がある。それらの実装方法を比較し、考察することでよりよい実装を提案できるのではと思う。これらを行うことが今後の課題である。

参考文献

[1] : The aspectj project, http://www.eclipse.org/aspectj/.

[2] : InTrigger. Intrigger platform, https://www.intrigger.jp/registration/.

[3] : Java Remote Method Invocation, http://java.sun.com/j2se/1.5.0/ja/docs/ja/guide/rmi/index.html.

[4] : The Jikes Research Virtual Machine, http://jikesrvm.org/.

[5] : Spring website. Spring framework, http://www.springframework.org/.

[6] Bockisch, C., Arnold, M., Dinkelaker, T. and Mezini, M.: Adapting virtual machine techniques for seamless aspect support, *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, New York, NY, USA, ACM, pp. 109–124 (2006).

[7] Bockisch, C., Haupt, M., Mezini, M. and Mitschke, R.: Envelope-Based Weaving for Faster Aspect Compilers, *NODe/GSEM*, pp. 3–18 (2005).

[8] Bockisch, C., Haupt, M., Mezini, M. and Ostermann, K.: Virtual machine support for dynamic join points, *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, New York, NY, USA, ACM, pp. 83–92 (2004).

[9] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G.: An Overview of AspectJ, *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, London, UK, Springer-Verlag, pp. 327–353 (2001).

[10] Navarro, L. D. B., Südholt, M., Vanderperren, W., De Fraine, B. and Suvée, D.: Explicitly distributed AOP using AWED, *AOSD*

'06: Proceedings of the 5th international conference on Aspect-oriented software development, New York, NY, USA, ACM, pp. 51–62 (2006).

[11] Nicoara, A., Alonso, G. and Roscoe, T.: Controlled, systematic, and efficient code replacement for running java programs, *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, New York, NY, USA, ACM, pp. 233–246 (2008).

[12] Nishizawa, M., Chiba, S. and Tatsubori, M.: Remote pointcut: a language construct for distributed AOP, *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, New York, NY, USA, ACM, pp. 7–15 (2004).

[13] Pawlak, R., Seinturier, L., Duchien, L., Florin, G., Legond-Aubry, F. and Martelli, L.: JAC: an aspect-based distributed dynamic framework, *Softw. Pract. Exper.*, Vol. 34, No. 12, pp. 1119–1148 (2004).

[14] Popovici, A., Alonso, G. and Gross, T.: Just-in-Time Aspects: Efficient Dynamic Weaving for Java, *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pp. 100–109 (2003).

[15] Popovici, A., Gross, T. and Alonso, G.: Dynamic weaving for aspect-oriented programming, *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, New York, NY, USA, ACM, pp. 141–147 (2002).

[16] Sato, Y., Chiba, S. and Tatsubori, M.: A selective, just-in-time aspect weaver, *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering*, New York, NY, USA, Springer-Verlag New York, Inc., pp. 189–208 (2003).

[17] Suvée, D., Vanderperren, W. and Jonckers, V.: JAsCo: an aspect-oriented approach tailored for component based software development, *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, New York, NY, USA, ACM, pp. 21–29 (2003).

[18] Truyen, E., Janssens, N., Sanen, F. and Joosen, W.: Support for distributed adaptations in aspect-oriented middleware, *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, New York, NY, USA, ACM, pp. 120–131 (2008).

[19] Villazón, A., Binder, W., Ansaloni, D. and Moret, P.: Advanced runtime adaptation for Java, *GPCE '09: Proceedings of the eighth international conference on Generative programming and component engineering*, New York, NY, USA, ACM, pp. 85–94 (2009).

[20] 千葉滋: Javassist Home Page, `http://www.csg.is.titech.ac.jp/~chiba/javassist/index.html`.

[21] 千葉滋: アスペクト指向入門　-Java・オブジェクト指向から AspectJ プログラミングへ, 技術評論社 (2005).