平成19年度 学士論文

同期的にアドバイスを活性化できる分散動的アスペクト指向システム

東京工業大学 理学部 情報科学科
学籍番号 04-2514-2
森田 悟史

指導教員
千葉 滋 准教授

平成20年2月20日

概要

近年、プログラムの実行を止めずにその振舞いを変更する技術としてダイナミックアスペクト指向プログラミング (DAOP) が注目を浴びている。アスペクト指向プログラミング (AOP) は、オブジェクト指向では分離しきれない関心事をアスペクトというモジュールにまとめるための技術である。アスペクトを定義すると、クラスに新たな処理が織り込まれ、クラスの振る舞いが変更される。クラスに織り込む処理はアドバイスというモジュールに記述し、いつアドバイスを実行するかをポイントカットという指定子を用いて決定する。DAOP では、動的にアスペクトを織り込むため、アドバイスが活性化されるタイミングがプログラムに存在する。これは、織り込まれたアスペクトが有効になる瞬間である。

分散環境で DAOP を用いる場合、各ホストで活性化のタイミングがずれてしまう問題がある。これは、各ホストの CPU 使用率の違いやネットワーク間の遅延などにより、全てのホストに対してアスペクトが一斉に織り込まれる保証がないためである。既存の分散 DAOP システムでは、アスペクトを織り込んだ後に各ホスト間でアドバイスが活性化されるタイミングを調整するための処理を、アドバイスボディーに記述する必要がある。しかし、このような実装方法では、アドバイス内に活性化のタイミングを考慮する記述が入り込むため、可読性が低下してしまう。これは、プログラムの再利用性の低下につながりバグの原因になりやすい。また、調整のために必要なホスト側のコンテクストを既存システムでは利用できない場合もあるため、活性化のタイミングを考慮することが非常に困難である。例えば、アプリケーションの同期に合わせて各ホストに織り込まれたアドバイスを活性化したい場合には、アプリケーションの実行状況を考慮して活性化のタイミングを調整する必要があるが、アプリケーションの実行状況をアドバイスでは把握できないことがある。

本研究では、活性化のタイミングを特別なポイントカットとして記述することができる分散 DAOP システム D&D を開発した。D&D は、クラスに織り込んだアドバイスコードを呼び出す段階で、活性化すべきかをこのポイントカットの指定により判断する。アドバイスボディーから活性化の調整処理を分離して記述することが可能になるため、アドバイス内に活性化のタイミングを考慮する記述を開発者が行う必要はない。D&D は、

メソッド呼び出しや、フィールドの値を基準にしてアドバイスの活性化を各ホスト間で同期させることができる。これを利用することで、アプリケーションの同期に合わせてアドバイスを活性化することが可能である。本論文では、同期が必要な分散アプリケーションの例として N 体問題を用いた。そして、実験として N 体問題アプリケーションに描画機能をアスペクトとして動的に追加した。D&D を用いることで、N 体問題の同期にあわせてアドバイスを活性化することができ、確実に正しいデータを描画することが可能となった。

また、D&D は HotSwap を用いて織り込みを行うため、最小限のオーバーヘッドでアドバイスを実行することができる。実験では、本システムを用いることによるオーバーヘッドは約 2%であることが分かった。また、アドバイスを呼び出すときには特にオーバーヘッドがないことが分かった。D&D を用いて織り込んだ描画のためのアドバイスコードは、アプリケーションの実行前に描画機能を予め実装した場合と比較して、実行時間がほぼ一致した。また、バリア同期を用いて活性化を同期させる方法を実験の比較対象として、活性化までにかかる時間を測定した。バリア同期では、アスペクトが織り込まれるタイミングやアプリケーションの実行時間に応じて活性化までにかかる時間が変動してしまったが、本システムを用いることで、安定した時間で活性化できることが分かった。

謝辞

本研究を進めるにあたり、研究の方針や論文の組み立て方について指導をしていただいた指導教員の千葉滋准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頂いた光来健一氏、論文の内容について様々な助言をしてくださった西澤無我氏、論文を構成するにあたり相談に乗っていただいた柳澤佳里氏に感謝致します。

また、堀江倫大氏には、論文の書き方、実験方法など親身になって指導していただきました。心より感謝致します。最後に、本研究を行う上で励まして頂いた同研究室の皆様に心から感謝致します。

目 次

図 目 次

表 目 次

第1章　はじめに

近年、Dynamic AOP (DAOP) がプログラムの実行を止めずに振舞いを変更する技術として注目を浴びている。これは、Aspect-Oriented Programming (AOP) という、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。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、予め設定された特定の振舞いしか変更できないため、任意の振舞いを変更したい場合は、プログラムを停止し、再コンパイルし、再スタートする必要がある。この、DAOP を用いると、プログラムの実行を止めずに、任意の振舞いを変更できる。

分散環境で動かすアプリケーションを、オブジェクト指向言語や AspectJ [5] 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い。この問題を解決するために、分散に関する記述もモジュール化できる AOP が研究されてきた [7]。そして、分散環境で動いているプログラムを動的に変更することができる分散対応の DAOP である、分散 DAOP も研究されてきている [6, 9]。

しかし、DAOP は自由度が高い分問題点も多い。DAOP は、動的にアスペクトを織り込むため、織り込まれたアスペクトが有効になる瞬間である、“アドバイスが活性化されるタイミング” が存在する。分散環境で DAOP を用いる場合、ネットワークの問題などで、アスペクトが全てのホストに対して一斉に織り込まれるとは限らない。そのため、活性化のタイミングがホストによってずれてしまうという問題がある。既存の分散 DAOP でアドバイスが活性化されるタイミングを調整するためには、アドバイスボディーの実装により、実行を無視するかどうかで調整しなければならない。このような実装をすると、アドバイスの中に活性化のタイミングを考慮する記述が入り込み、可読性が下がってしまうため、間違いの原因になりやすい。また、活性化のタイミングを考慮することは非常に困難であり、活性化のタイミングを調整するために必要なコンテクストが利用できない場合もある。例えば、アプリケーションの同期に合わせてアドバイスを活性化したい場合は、アプリケーションの状況を考慮して活性化のタイミングを調整しなければならないが、アプリケーションの状況をア

ドバイスでは把握できないかもしれない。

本研究では、活性化のタイミングを特別なポイントカットとして記述することができる、D&D という分散 DAOP システムを開発した。D&D では、アドバイスを活性化するかどうかの判断をアドバイスの呼び出し側で行っている。このポイントカットを利用することで、開発者はアドバイスの中に活性化のタイミングについての記述をする必要がなくなる。D&D は、メソッド呼び出しや、フィールドの値を基準にして同期したアドバイスの活性化ができるため、アプリケーションの同期に合わせてアドバイスを活性化することが可能になっている。また、D&D は HotSwap を用いて織り込みを行うため、最小限のオーバーヘッドでアドバイスを実行することができる。

本稿の残りは、次のような構成からなっている。第 2 章では既存のアプリケーションの振舞い変更方法とその問題点について述べ、第 3 章では、本研究で開発したシステムによる解決法を示す。第 4 章では、システムの実装方法について述べ、第 5 章では、システムの性能を測る実験について述べる。そして、第 6 章で、本研究のまとめと今後の課題について述べる。

第2章　実行時におけるアプリケーションの振舞いの変更方法とその問題点

実行時にアプリケーションの振舞いを変更することや機能を追加することが望まれる場合がある。本章ではまず、そのような要求に対する既存の解決方法を説明し、既存の解決方法における問題点を述べる。

2.1　実行時におけるアプリケーションの振舞いの変更

実行時にアプリケーションの振舞いを変更したり、機能を追加したりしたい場合がある。普通、アプリケーションの振舞い変更や機能追加をしたい場合は、アプリケーションの実行を止め、プログラムを変更した後で再コンパイルし、再実行しなければならない。科学技術計算などを扱う、実行に時間がかかる計算プログラムは、計算終了までに数時間から数日かかる場合もある。そのため、機能追加のためにプログラム止めて再実行することは望まれることではない。このように、アプリケーションを止めたくない場合があるため、実行中にアプリケーションの振舞いを変更したいという要求がある。以下、このような要求の例と既存の解決方法での問題点を述べる。

2.1.1　N 体問題

N 体問題の計算について考える。N 体問題とは、 N 個の質点間の重力を計算することで、質点の動きをシミュレートする計算のことである。この計算は、宇宙における星の動きをシミュレートするときなどによく考えられている。2 体問題までが計算として解くことが可能であり、3 体問題以上となると式で解くことが不可能となるため、計算のステップごとに各質点間の相互作用を計算していかなければならない。各ステップで全ての質点間の相互作用を計算するので、質点が多い場合計算に時間がかかってしまう。そのため、通常は様々な近似が行われながら計算される。

今回は、この N 体問題計算を近似などは行わずに java を用いて分散環境で計算させることを考える。質点の初期座標、質量、速度などのデータを各分散ホストに分配し、ステップごとにホスト間で質点のデータをやりとりして計算していく。全てのホストが同一ステップを計算している必要があるため、計算ごとに同期をとらなくてはならない。図 2.1 のように、calc() メソッドを呼ぶことで 1 ステップの計算が行われる。その間、ホスト同士で同期がとられ、n ステップ目の calc() と n+1 ステップ目の calc() は同じタイミングでは呼び出されないようになっている。

そこで、この N 体問題の計算プログラムに図 2.2 のような描画機能を追加することを考える。一般的に、このようなシミュレーションでは計算の視覚化が望まれることが多い。計算状況がどうなっているかというログ出力の意味でも重要である。しかし、はじめから最後まで描画し続けてしまうと計算スピードに大きな影響を与えるので、必要な時にだけ視覚化することが望まれる。このことから、この N 体問題も実行中に機能を追加したいという要求があるアプリケーションの例であると言える。

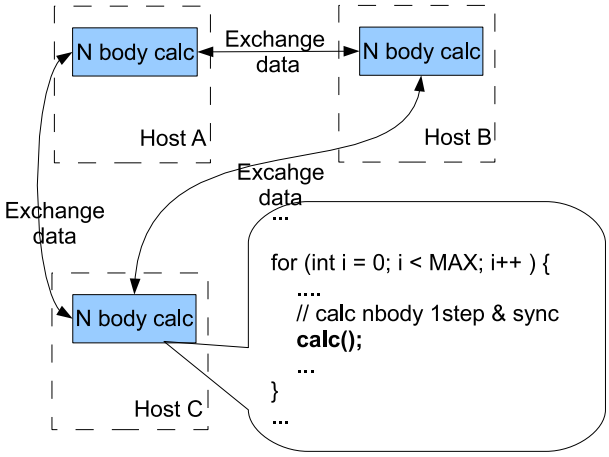

図 2.1: 分散環境での N 体問題計算

2.1.2　既存のアプリケーション変更方法と問題点

描画機能追加の方法として、第一に考えられる方法は、はじめから視覚化のためのコードを書いておき、ある瞬間から描画を開始するようにしておく方法である。しかし、この方法だと描画開始のタイミングや終了のタイミングを動的に変更することは難しい。また、描画のタイミングを変

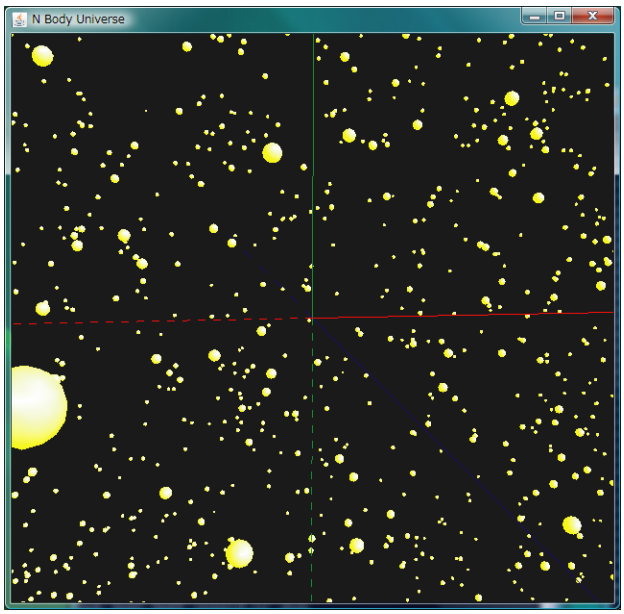

図 2.2: N 体問題計算の視覚化

更できるようになっていたとしても、描画の方法を変更することはできない。様々な視覚化の方法がある場合は、その中で要求される視覚化方法の全てを予め用意しておくのは困難だと言える。N 体問題の場合でも、3 次元座標空間にプロットしていくという方法以外にも、ある星からみた相対的な位置を描画したいかもしれない。

　次に考えられる方法は、アプリケーションの実行を止めてしまう方法だ。これは、完全にアプリケーションを再実行するわけではなく、チェックポインティング機能を付けておき、計算を再開するために必要な情報を保存しておくということだ。こうすれば、計算を止めても再度中断した時の状況に戻して計算を再開できる。そのため、計算を中断させてアプリケーションに機能を追加することが可能となる。しかし、この方法で問題となるのは、チェックポインティングで保存している情報だけで機能追加のための要求を満たすことができるかということだ。N 体問題の場合でも、計算再開のために必要な各質点の座標データを保存しているかもしれないが、その状況に至るまでに何ステップかかったかの情報は、計算再開には必要ないために保存してないかもしれない。こうなると、各質点の位置と現在までにかかったステップ数を視覚化したいという要求があった場合、視覚化できないことになってしまう。そこで、変化する要求に対応す

るため、計算アプリケーション中のすべての情報をチェックポインティングしておく設計をする方法も考えられるが、チェックポインティングにかかる時間的コストが大きくなってしまうため、あまり良い解決方法だとは言えない。

2.2　DAOP を使った動的変換

プログラムを動的に変換する方法として Dynamic Aspect-Oriented Programming (DAOP) という技術がある。以下、DAOP を用いた場合の解決方法と注意点を述べる。

2.2.1　アスペクト指向 (AOP) とは

アスペクト指向とは、オブジェクト指向ではうまくモジュール化できないモジュール間にまたがる処理（横断的関心事）をうまくモジュール化する技術である。AOP は以下のような概念を持っている。[18]

- ジョインポイント
  プログラム実行中のある点のことであり、アスペクトと結びつけることができるポイント。代表的なものとして、メソッドやコンストラクタの呼び出し・実行時点、フィールドの参照や代入などがある。

- ポイントカット
  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、必要なジョインポイントの集合をつくり、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
  ポイントカットによって構成されるジョインポイントの集合において実行したい処理を記述したもの。

- アスペクト
  ポイントカットとアドバイスの組み合わせを指定するモジュール単位。横断的関心事をまとめたモジュール。

- 織り込み (weave)
  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付けること。

- アンウィーブ (unweave)
  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

アスペクト指向言語として最も有名で、広く使われているのは、おそらく AspectJ [5, 12] である。AspectJ は Java を拡張してアスペクト指向の機能を使えるようにしたものである。

AspectJ で用いることができるポイントカットでは、以下のようなものが代表的である。

- call
  メソッド呼び出し時、コンストラクタ呼び出し時
- execution
  メソッド実行時、コンストラクタ実行時
- set, get
  フィールド代入時, フィールド参照時

AspectJ で用いることができるアドバイスは、before、after、around がよく知られており、それぞれ、ジョインポイントの事前、事後、その時点を表す。

```
1  public class HelloWorld {
2      public static void main(String[] args) {
3          HelloWorld h = new HelloWorld();
4          h.printHello();
5          h.printWorld();
6      }
7      private void printHello() { System.out.print(‘‘Hello ’’); }
8      private void printWorld() { System.out.println(‘‘world !’’); }
9  }
10
11 public aspect HelloAspect {
12     pointcut print(): execution(void printWorld()) ;
13     before(): print() { System.out.print(‘‘Aspect ’’); }
14 }
```

図 2.3: AspectJ を用いたアスペクト記述例

AspectJ を用いると図 2.3 のようにアスペクトを実装することができる。1-9 行目は、printHello() と printWorld() メソッドを用いて Hello world ! と出力する HelloWorld クラスである。そして、11-14 行目が AspectJ で記述された HelloAspect アスペクトとなっている。HelloAspect アスペクトでは execution ポイントカットにより、printWorld() メソッドの実行が選択されており、そのポイントカットで before アドバイスが宣言されている。

このアスペクトを織り込んで、HelloWorld クラスを実行すると、

Hello Aspect world !

と出力されるようになる。5 行目で実行される printWorld() メソッドの実行前に HelloAspect の before アドバイスが呼び出されるようになっているため、このような出力になる。

2.2.2　DAOP とは

DAOP とは、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技術である。Java ベースの代表的な DAOP での動的な織り込みの実現方法は大きく分けて以下のような種類がある。

- フックを用いた織り込み
- イベント通知を用いた織り込み
- HotSwap を用いた織り込み
- VM レベルでの織り込み

以下、これらについて簡単に説明する。

フックを用いた織り込み

この方法は、あらかじめジョインポイントにフックを挿入しておく方法である。このフックにより、そのジョインポイントがポイントカットとして選択されているかどうかを確認することができる。そのジョインポイントがポイントカットとして選択されているならばアドバイスを実行する。

この方法を用いた DAOP システムには、JAC [8, 9]、PROSE2 [10] がある。JAC ではクラスがロードされるときに、ジョインポイントにフックが挿入される。PROSE2 では Jikes Research Virtual Machine (RVM) を拡張していて、 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は大きく分けて 2 つの方法がある。1 つ目は、考えられる全てのジョインポイントにフックを挿入する方法であり、2 つ目は、予め指定したジョインポイントにのみフックを挿入する方法である。しかし、それぞれ問題がある。1 つ目の方法では、ポイントカットとして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ため、ポイントカットとして選択されているかどうかの確認が行われてしま

う。そのため、アスペクトが織り込まれていない場合でもオーバーヘッドが大きくなってしまうという問題がある。2つ目の方法だと、後から予め指定しなかったジョインポイントに対してフックを追加することができないため、予測できなかったジョインポイントにはアスペクトを織り込むことができないという問題がある。

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はなく、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。ジョインポイントでイベントが発生すると、システムはプログラムを停止させ、アドバイスを実行した後でプログラムに復帰する。

この方法を用いたシステムには PROSE [11]、 Wool [13] などがある。これらのシステムは、デバッガを用いて、ジョインポイントで発生するようにしたイベントを扱い、スレッドやプログラム全体を停止させてアドバイスを実行している。また、Java Platform Debugger Architecture (JPDA) を用いると、メソッドに入るときと出るとき・フィールドの参照や変更・例外の発生・特定の行への到達、そしてスレッドの開始や終了などでイベントを発生させるように要求できる。このように、JPDA を用いるとイベント通知を用いた織り込みを実現することができる。Wool ではこの JPDA を用いている。

しかし、この方法を用いる織り込みでは、イベント発生・通知・プログラムの停止・アドバイスの実行・プログラムへの復帰とイベント発生からアドバイスの実行終了までにかかる時間的コストが高く、アドバイスが実行される回数に比例してオーバーヘッドが大きくなってしまうという問題がある。

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置き換えることで織り込みを実現している。JPDA や java.lang.Instrument パッケージの API を用いることで HotSwap を用いたアスペクトの織り込みを行うことができる。この方法では、クラスのバイトコードを書き換えるので、書き換えられたクラスは、はじめから書き換え後のコードだった場合と同じ動きをすることができる。

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。また、バイトコード変換されたメソッドがアクティブなスタックフレームを持つ場合、アク

ティブなスタックフレームは元のメソッドを引き続き実行する。そのため、新しく定義されたメソッドは新しい呼び出しから使用される。

この方法を用いたシステムには Wool がある。Wool では、イベント通知を用いた織り込みとこの方法を組み合わせており、Just-in-Time にフックを埋め込み、埋め込んだフックが有効になるまではイベント通知を用いてアドバイスを実行するようになっている。これにより Wool は、イベント通知のみを用いたアスペクトの織り込みと比較して速度向上ができている。

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することでアスペクトの織り込みができるようにしている。

この方法を用いたシステムには、Steamloom [1] がある。Steamloom は、RVM を拡張している。また、Steamloom では、Thread local や instance local なアスペクトの織り込みも行うことができる。

2.2.3　DAOP を使った振る舞いの変更

DAOP を使うと動的にアスペクトを織り込むことができるので、N 体問題アプリケーションに描画アスペクトを自由なタイミングで織り込むことができ、視覚化を実現できる。また、視覚化のコードは計算プログラムに横断してしまうため、アスペクト指向を使うことでうまくモジュール化できるという利点もある。

2.1.2 章で確認したように、はじめから視覚化コードを実装しておく方法と、チェックポインティングをしておき、計算アプリケーションの実行を中断してから実装する方法では問題があった。DAOP を用いてアスペクトとして視覚化コードを織り込む場合、実行後に実装した視覚化アスペクトを織り込むことができる。これより、視覚化の方法が複数ある場合でも対応できる。また、実行中にアスペクトを織り込むので、視覚化のために必要なデータが足りないという心配もない。

分散環境への対応

今回、N 体問題を分散環境で計算することを考えていた。分散環境では、プログラムは各ホストに散らばって動いている。そのため、遠隔ホスト上のジョインポイントにアスペクトを織り込むことができない場合、図 2.4 のように、アスペクトも各ホストに配置しなくてはならない。このような、複数のホストにアスペクトが散らばっている状態では、アスペク

ト間のデータのやりとりなど、分散を意識してアスペクトを実装しなければならない。また、アスペクトがそれぞれのホストに散らばって配置されることも好ましくない。

図 2.5 のように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できれば、アスペクトを複数のホストに配置する必要がなくなり、アドバイスを 1 つのホスト上で実行することが可能になる。視覚化の場合でも、描画処理は 1 つのホストで実行されるべきである。

DJcutter の Remote Pointcut [7] 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。DJcutter は AspectJ と同じようにアスペクトを記述することができ、ジョインポイントのホストを指定するポイントカットが扱える。しかし、DJcutter は動的に織り込みを行うことができないため、今回の例では用いることができない。

JAC もまた、遠隔ホストに対する織り込みを実現することができる [9]。しかし、JAC は Remote Pointcut のような機構は備えていない。JAC を用いることでできる分散ホストに対する織り込みは、複数のホストに対してアスペクトを配置することで行われている。このことから、遠隔ホスト上のポイントカットで行われるアドバイスを 1 つのホストで実行しようとすると、図 2.4 のような分散を意識したアドバイスの実装が必要となる。

DJAsCo は JAsCo [14] を AWED を用いて分散対応にしたものであり、Remote Pointcut を備えた DAOP システムである [6]。DJAsCo は同じアスペクトの異なるインスタンス間でのステートシェアリングができるようになっている。また、アドバイスの実行方法を synchronous 実行と asynchronous 実行とで選択することができる。

DyRes [15, 4] 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分散 DAOP システムであり、Spring AOP [16] を拡張して実装されている。これは、アドバイス間に依存関係がある場合に有効であり、安全にアンウィーブするために必要な処理を記述することで、その記述をシステムが読み込み、安全な状態にしてからアンウィーブを行うことができる。

適した織り込み方法の選択

これまで、様々な種類の DAOP システムが存在することを確認してきた。ここからは、実際に N 体問題に視覚化機能を追加するために適している DAOP システムはどういったものなのかを議論する。

科学技術計算は時間がかかってしまうため、再実行することが望まれていなかった。よって、DAOP システムを用いることにより処理速度が極端に低下してしまうということは避けるべきである。このことから、アド

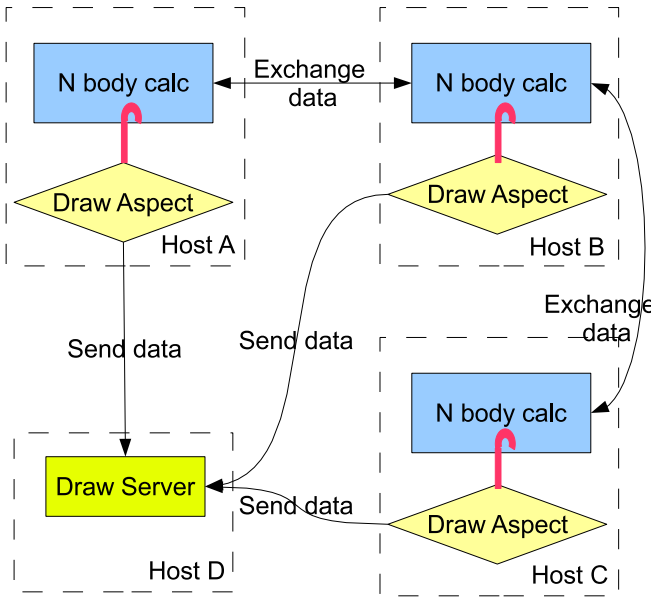

図 2.4: 描画機能追加を通常の DAOP で行った場合

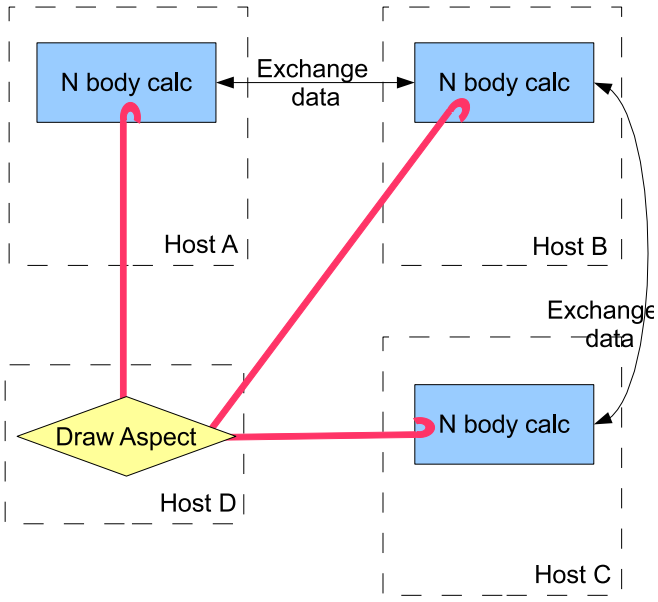

図 2.5: 遠隔ポイントカットを用いた場合の描画

バイスの実行のためにかかるオーバーヘッドは最小限に抑えたい。これをもとに先ほどあげた DAOP の種類を見る。

- フックを用いた織り込み
  アスペクトを織り込む前からフックが挿入されているため、アドバイスを実行していない間もオーバーヘッドが大きい。

- イベント通知を用いた織り込み
  アドバイス実行までにかかるタイムラグが大きく、アドバイス数に比例してオーバーヘッドが大きくなる。

- HotSwap を用いた織り込み
  バイトコードの書き換えに時間がかかるため、アスペクトの織り込みまでに時間がかかる。しかし、織り込まれると、はじめからコードが挿入されていた場合と同じスピードが期待できる。

- VM レベルでの織り込み
  扱う VM の性能によって処理速度は異なる。例えば steamloom の場合では、ある特定の条件では速いかもしれないが、アドバイスを実行しない間も拡張した VM で実行されるので、その場合は Sun VM よりは遅くなる。

以上により、HotSwap による織り込みが適していると考えられる。アスペクトの織り込みに時間がかかるという問題があるが、織り込みの瞬間に描画が開始される必要はないと考えられるため問題はない。

2.3　既存の DAOP の問題点

DAOP を用いた場合の視覚化機能の追加について議論してきた。分散環境のための機構が備わっていて、HotSwap を使った織り込みができる DAOP システムを用いればうまく視覚化機能を追加することができるように思える。しかし、既存の DAOP ではまだ問題がある。以下にその問題点について述べる。

2.3.1　活性化のタイミング

この論文では、アドバイスが実行される状態になることを活性化と呼ぶことにする。既存の DAOP では、アスペクトの織り込みが完了すると、すぐにアドバイスが実行されるようになる。織り込み完了のタイミングが活性化のタイミングになっているということだ。

分散環境におけるアスペクトの織り込みで問題となるのが、活性化のタイミングだ。分散環境では、織り込みの命令が、遠隔ホストの CPU 使用率やネットワークの問題などで、全てのホストに対して一斉に届くとは限らない。このことから、同じアスペクトの織り込みでも、アドバイスが活性化されたホストと準備できていないホストが共存してしまう可能性がある。

既存の DAOP で活性化のタイミングを操作するためには、図 2.6 のように、アドバイスの内容によりアドバイス呼び出しを無視するか実行するかを決めることで解決しなくてはならない。例えば、全てのホストにアドバイスが準備されてから活性化されるアドバイスが実装したいとする。この場合、アスペクトのフィールドに、どのホストでアドバイス呼び出しがあったかを記録しておき、全てのホストによってアドバイス呼び出しが起こってからアドバイスの中身を実行するようにすることで解決できる。しかし、これではアドバイスボディーに活性化のタイミングを考慮する記述が入り込んでしまう。これは、アドバイスの可読性を下げてしうことにつながり、間違いの原因になりやすい。また、アスペクトが織り込まれるタイミングを考慮してアドバイスを実装するのは困難であり、様々なタイミングが要求される場合はアドバイスの実装だけで解決することが不可能になる場合もある。また、アスペクトで、どこのホストからアドバイスが呼ばれているのか確認することができない場合、この実装は不可能ということになる。

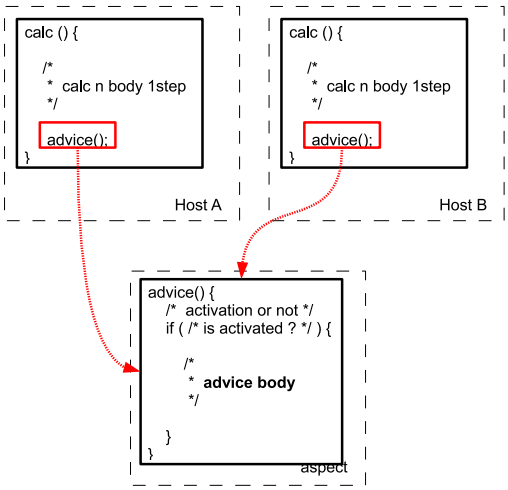

図 2.6: 活性化の判断をアドバイス内で行う方法

2.3.2　同期しているアプリケーションへの織り込み

N 体問題の計算は各ホストの間で同期をしながら計算していた。ここから、実際に N 体問題アプリケーションに描画機能を追加することを考える。

描画アスペクトのアドバイスの引数として用いるのは、質点の座標などのデータのオブジェクトだとする。これで、各ホストで計算中の座標データを送信するようなアドバイス呼び出しになる。アドバイスボディーでは、引数として与えられた座標のデータをアスペクトに保存しておき、保存した質点データの数が計算中の質点の数と一致したら (全ての質点データを受信したら) スクリーンにプロットするという処理を行う。次のデータがアスペクトに到着したら、再度保存を開始し、計算中の質点の数まで受信したのちスクリーンをリセットし、プロットし直す。これでうまく描画できるはずだが、アドバイスをこのような実装にした場合、アドバイスが活性化されるタイミングによっては正しいデータを描画できないことになってしまう。

ホスト A、B、C で N 体問題の計算を行っていたとする。各ホストは同期をとりながら 1 ステップずつ計算を行っている。ここに描画機能のアスペクトを織り込む。しかし、ネットワークの問題で A に対する織り込みが遅れてしまったとする。すると、B と C にはアスペクトが織り込まれているので、B、C のみがアドバイスを呼ぶことで質点のデータを送信する。アドバイス内では、B、C の座標データを保存する処理を行う。そして、A、B、C は同期をとり、次の計算ステップに入る。この時点で A に対する織り込みも完了しているとする。次のステップで行われるアドバイス呼び出しの順番が B、A、C だったとすると、まずアドバイス呼び出しで B の質点データが送信され、B の座標データがアスペクトに保存される。そして、この時点で 3 つの座標データがアスペクトに保存され、計算中のデータ数と一致した。よって、アスペクトに保存された座標データをスクリーンにプロットするが、保存された座標データは、n ステップ目の B、C のデータと n+1 ステップ目の B のデータであるため、正しく描画できていないことになる。そして次の A、C でのアドバイス呼び出しにより、アスペクトに保存されるデータは、ホスト A、C のデータということになる。このように (図 2.7 からも分かるように)、織り込みのタイミングがずれてしまうと、アプリケーションが同期していたとしても、アドバイス呼び出しがずれてしまうという問題が起こる。

この問題は、全てのホストにアドバイスが準備されてから活性化するという方法では解決できない。先ほどの例でこの方法を使ったとする。最初の B、C でアドバイスが呼ばれた段階では、まだ全てのホストにアドバイスが準備できていないため、呼び出しが無視される。そして、A、B、C

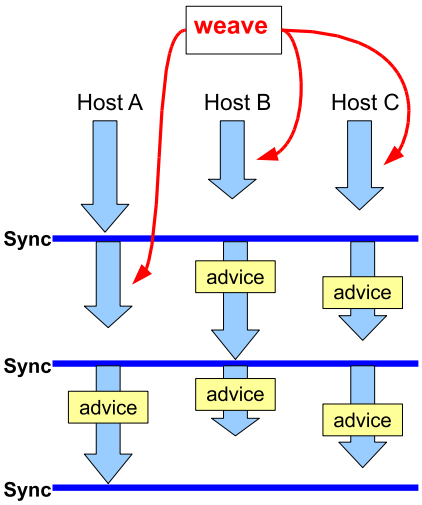

図 2.7: 織り込みがずれてしまった場合

が同期をとり、B によるアドバイス呼び出しが起こる。しかし、まだ全てのホストからの呼び出しがないため、このアドバイス実行も無視される。次に A によるアドバイス呼び出しが起こり、これで全てのホストからアドバイス呼び出しがあった。そして、次からのアドバイスが活性化されるため、この呼び出しも無視される。しかし、次の C による呼び出しも無視すべき呼び出しであるにもかかわらず、すでにアドバイスは活性化されているため座標データの保存が行われてしまう。このように、簡単にはこの問題を解決することはできないということが分かる。

　このことから、アドバイスが全てのホストに準備されるという活性化条件のほかに、織り込み先の状況によって活性化すべきかどうかを判断する必要がある。例えば、N 体問題の計算の各ステップでフィールドのカウンタの値を増やし、同期処理ではその値が一致しているかを調べているとする。この場合、アドバイス呼び出しの引数を質点データとそのカウンタにし、アドバイスの実装で、カウンタの値を使って活性化するタイミングを調整すればうまくいくかもしれない。しかし、このことでやはり視覚化以外の関心事がアスペクトに入り込んでしまい可読性が下がってしまう問題がある。また、もしそのようなフィールドが直接参照できないような場所にあったとき、この方法は実現できなくなる。

2.3.3　アドバイス間に依存関係がある場合の織り込み

アドバイス間に依存関係がある場合について考える。例えば、アドバイス A、B があったときに、A と B はセットで動作しているとする。このとき、もしアドバイス B のみが織り込まれてしまった場合、何かしら不具合が起こるかもしれない。このように、アドバイス間に依存関係がある場合は、活性化するために考えるべき条件がアドバイス間をまたがってしまい、ますます複雑になるという問題がある。

第3章　D&D

本研究では、活性化のタイミングを調整できる分散 Dynamic AOP システム D&D を開発した。D&D は、アドバイスが活性化されるための条件を特別なポイントカットとして記述することができる。以下、本システムについて述べる。

3.1　特徴

本システムは、既存の分散 DAOP の問題であるアドバイスが活性化される条件を特別なポイントカットとして記述することができる。以下、本システムの特徴を述べる。

3.1.1　分散への対応

本システムは、遠隔ホスト上のジョインポイントを選択してアスペクトを織り込むことができる。アスペクトを配布することで、遠隔ホスト上で実行させるという方法ではなく、Remote Pointcut を用いた時と同じように、1 つのホストでアドバイスを実行することができる。

遠隔ポイントカットを指定するために、ポイントカットの記述の中でクラス名、メソッド名、そしてホスト名の配列を渡す。そうすることで、指定したホストのジョインポイントでアドバイスが実行されるようになる。

以下のようにポイントカットを作成すると、nbody パッケージ NBodyCalc クラスの calc メソッドの実行がポイントカットとして選ばれる。そして、ホスト名の配列が指定されているので、遠隔ホスト上のジョインポイントが選択される。

```
String[] hosts = new String[] { hongo001, hongo002 .....  };
Pointcut pcut =
        Pcd.execution(“nbody.NBodyCalc”, “calc”, hosts);
```

3.1.2　活性化のためのポイントカット

本システムでは、既存の DAOP では難しかった、アドバイスを活性化するタイミングの判断を特別なポイントカットとして記述することで可能にしている。このポイントカットを *Activation Pointctu* と呼ぶ。

既存の DAOP システムを用いてアドバイスが活性化されるタイミングを同期させたりしたい場合は、アドバイスが呼ばれた後で、アドバイス呼び出しを実行するか無視するかをアドバイスボディーで判断しなければならなかった。その上、アドバイスが実際に実行される段階では実行すべきかどうかの判断材料が不足していて、判断することすらできない場合もあった。

本システムでは、図 3.1 のようにアドバイスを実行するかどうかを呼び出し側で判断するようになっている。活性化の条件が満たされない限りアドバイスが呼び出されることはない。したがって、アドバイスを実行すべきかどうかの判断を開発者がアドバイス内に記述する必要がなくなる。

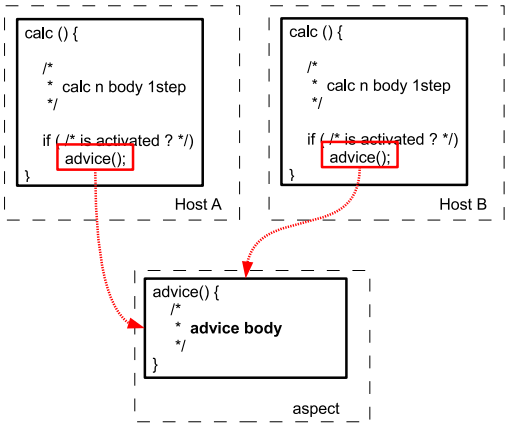

図 3.1: アドバイスを呼び出すかどうかを呼び出し側で判断する

現在、フィールドの値、メソッド呼び出しの状況を利用してアドバイスを活性化するかどうかの判断をさせるようにできる。また、アドバイスが活性化される条件として、あるアドバイスが活性化されているかという判断基準も用意してあるので、アドバイス間の依存関係にも対応できるようになっている。

3.1.3　Hotswap を用いたコード変換

本システムは、HotSwap を用いた動的バイトコード変換を用いて織り込みを実現している。よって、フックを用いた場合のようなアスペクトが織り込まれる前のオーバーヘッドや、デバッガを用いたときのような実行までのオーバーヘッドはない。アスペクトが織り込まれる前と織り込まれた後のものは、普通のクラスと同じ振る舞いをするので、最小限のオーバーヘッドでアドバイスを実行することができる。

3.2　D&D を用いたアプリケーションの振舞いの変更

既存の分散 DAOP システムでアプリケーションの振舞いを変更する場合、アドバイスが活性化されるタイミングがバラバラになってしまうという問題があった。以下、本システムを用いた解決方法を示す。

3.2.1　アプリケーションの同期に合わせたアドバイスの活性化

N 体問題の計算は、各ホスト間で同期をとりながら分散環境で計算していた。この計算プログラムに本システムを用いて描画機能をアスペクトとして追加する。

本システムを用いて描画アスペクトを実装すると、例えば、図 3.2 のようになる。本システムを用いてアスペクトを実装するためには、1 行目のように Aspect クラスを継承しなければならない。その上で、コンストラクタなどでポイントカットやアドバイスなどの設定を行う。

まず、6-9 行目のようにポイントカットを作成する。この例では、nbody パッケージの NBodyCalc クラスの calc メソッドの実行がポイントカットとして選択される。

次に、活性化の条件を記述する特別なポイントカットである、*Activation Pointcut* を 12 行目のように作成する。この例では、織り込み先のジョインポイントにおいて int 型の turn フィールドの値が一致したときにアドバイスが活性化されるようになる。N 体問題の計算がフィールドの値を用いて同期されている場合は、このようにすることで同期してアドバイスを活性化することができる。

そして、15-25 行目のように、アドバイスの定義と登録を行う。アドバイスは、15-16 行目のように、引数名、アドバイス名、ポイントカット、Activation Pointcut を引数として渡して定義している。Activation Pointcut を引数として渡さなかった場合は、アスペクトが織り込まれた瞬間にアドバイスが活性化されるようになる。アドバイスボディーは Advice クラスの advice() メソッドをオーバーライドして定義する。アドバイス

```
1  public class DrawAspect extends Aspect {
2
3      public DrawAspect() {
4
5          /* weave 対象のホストの配列を作る */
6          String[] hosts = new String[] {/*hongo001, hongo002, ..... */};
7
8          /* className, methodName, hostNames から pointcut を作成 */
9          Pointcut pc = Pcd.execution(‘‘nbody.NBodyCalc’’, ‘‘calc’’, hosts);
10
11         /* advice を活性化するタイミングを指定する pointcut */
12         ActivationPointcut apc = Apcd.field(‘‘turn’’);
13
14         /* advicebody を override して定義する */
15         Advice drawAdvice = new BeforeAdvice<LinkedList<Satellite>> (
16                 ‘‘satellite’’, ‘‘drawAdvice’’, pc, apc) {
17
18             @Override
19             public void advice(LinkedList<Satellite> satellites){
20                 drawSatellite(satellites);
21             }
22         };
23
24         /* advice を登録する */
25         setAdvice(drawAdvice);
26
27     }
28
29     private void drawSatellite(LinkedList<Satellite> satellites){
30         /*
31          * satellite オブジェクトの座標データをもとに描画処理を行う
32          */
33
34     }
35 }
```

図 3.2: D&D を用いた描画アスペクト 1

は、引数が、int、boolean などの基本データ型とその配列になっているものがオーバーロードされている。また、それ以外のオブジェクトの場合はジェネリックとして表現できる。今回は、引数として質点データのリストを受け取りたいので、ジェネリックを使いオーバーライドしてアドバイスを定義している。20,29 行目のアドバイスの中で呼んでいるメソッドで、実際の描画処理を行うようになっている。25 行目でアドバイスを登録し、アドバイスの定義を終えている。このアスペクトを、

```
new DrawAspect().weave();
```

のようにすると実際にアスペクトが織り込まれる。

```
1  public class DrawAspect extends Aspect {
2
3      public DrawAspect() {
4
5          /* pointcut を作成 */
6
7          ...
8
9          /* call Activation Pointcut を作成 */
10         ActivationPointcut apc = Apcd.call(‘‘nbody.NBodyCalc’’, ‘‘sync’’);
11
12         /* unweave するときのタイミングを指定するポイントカット */
13         InactivationPointcut iapc = Iapcd.call(‘‘nbody.NBodyCalc’’, ‘‘sync’’);
14
15         /* advicebody を override して定義する */
16         Advice drawAdvice = new BeforeAdvice<LinkedList<Satellite>> (
17                 ‘‘satellite’’, ‘‘drawAdvice’’, pc, apc, iapc) {
18
19             @Override
20             public void advice(LinkedList<Satellite> satellites){
21                 drawSatellite(satellites);
22             }
23         };
24
25         /* advice を登録する */
26         setAdvice(drawAdvice);
27     }
28     ...
29 }
```

図 3.3: D&D を用いた描画アスペクト 2

他の Activation Pointcut を用いたアスペクトとしては、図 3.3 のようにすることもできる。こちらの例では、10 行目のように Activation Pointcut として call が使われている。これを用いると、ポイントカットしている

ホストの全てで、指定したメソッド呼び出しが起こった後でアドバイスが活性化されるようになる。また、call の引数としてホスト名の配列を渡すことで、ポイントカットで指定したホスト以外のメソッドを指定することもできる。

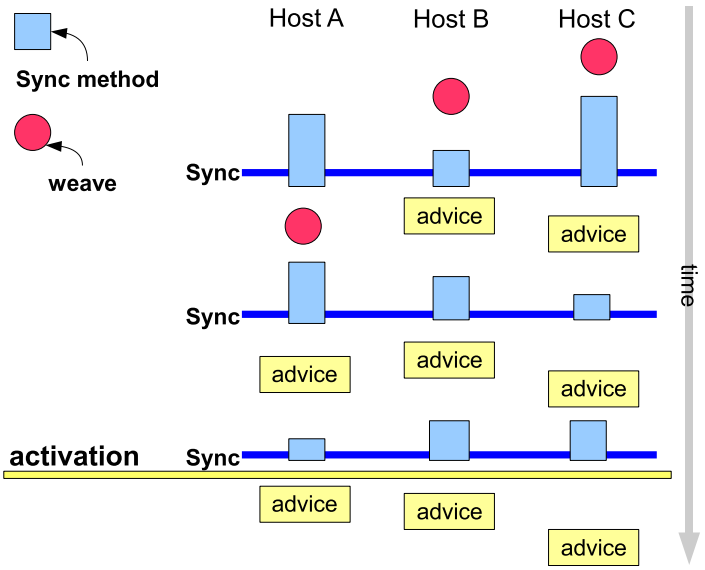

図 3.4: call Activation Pointcut での活性化までの流れ

図 3.4 は図 3.3 のように call Activation Pointcut を用いた時の N 体問題アプリケーションの流れを示している。各ホストでは、sync() メソッドを呼ぶことで、同期をしながら N 体問題の計算を行っている。call の引数として sync() メソッドを指定することで、アプリケーションの同期に合わせてアドバイスの活性化ができる。図 3.4 のように、各ホストは、アドバイスが全てのホストに準備されるのを待つ。もちろん、一部のホストだけにアスペクトが織り込まれても、アドバイスは活性化されていないので、実行されることはない。そして、全てのホストによって sync() メソッドが呼ばれた後に、アドバイスが活性化される。全てのホストで sync() メソッドが呼ばれていないのにアドバイスを呼ぼうとした場合、また 1 から sync() メソッドが全てのホストで呼ばれるのを待つ。このように、フィールドを参照する以外にも、メソッド呼び出しを利用することで、同期したアドバイスの活性化を実現できる。

そして、図 3.3 の 13 行目にあるように、アンウィーブするときの “非活性化” のタイミングを指定することもできる。アドバイスをアンウィー

ブするときに、Inactivation Pointcut が設定されていると、それに従ってアンウィーブされる。Inactivation Pointcut には、Activation Pointcut と同じように、フィールドを参照するものとメソッド呼び出しを基準にするものがある。Inactivation Pointcut を指定しなかった場合は、タイミングを気にせず、アンウィーブ命令が届いたホストからアンウィーブされる。アドバイスを定義するときは、Activation Pointcut を指定せずに、Inactivation Pointcut のみを指定することも可能である。

アンウィーブは、Aspect クラスの unweave() メソッドを呼ぶことで行われる。また、このメソッドの引数としてアドバイスを渡すことで、そのアドバイスだけがアンウィーブされる。指定しなければ、そのアスペクトの全てのアドバイスがアンウィーブされる。

3.2.2　アドバイス間の依存関係への対応

本システムを用いるとアドバイス間の依存関係があるアスペクトも実装することができる。

図 3.5 は 2 つのアドバイス間に依存関係がある場合の例である。8 行目で作成している、woven Activation Pointcut は、全てのホストにアドバイスが準備されてから活性化されるというものである。普通は、準備されたアドバイスから活性化されるので、アドバイスが活性化されたホストと準備されていないホストが存在するタイミングがある。この woven を Activation Pointcut とすることで、アドバイスが活性化された時には全てのホストでアドバイスが準備されているということが保証される。先ほど出てきた field や call は、活性化条件のために全てのホストにアドバイスが準備されていることが必要なので、明示的に woven を指定しなくても活性化された時には全てのホストでアドバイスが準備されている。

21 行目で作成している activated Activation Pointcut は、使用すると、アドバイスが活性化されるのが、引数として指定したアドバイスが活性化されてからになる Activation Pointcut である。引数として渡している intAdvice は、woven が Activation Pointcut となっているため、stringAdvice が活性化された時には、intAdvice は全てのホストで準備されていることが保証される。

また、32 行目では呼び出し側でアドバイスを囲むブロックを指定できる。この例のようにすると、挿入されるアドバイス呼び出しが、

```
for (int j = 0; j < 100; j++) {
    advice(str);
}
```

```
1  public class TestAspect extends Aspect{
2
3      public TestAspect(){
4
5          /* pointcut を作成 */
6          Pointcut pcut = Pcd.execution(className, methodName, hosts);
7
8          ActivationPointcut apc = Apcd.woven();
9
10         /* advicebody を override して定義する */
11         Advice intAdvice = new BeforeAdvice(argument,
12                 ‘‘intadvice’’, pcut, apc){
13             @Override
14             public void advice(int i){
15                 intMethod(i);
16             }
17         };
18
19         Pointcut pcut2 = Pcd.execution(className, methodName, hosts);
20
21         ActivationPointcut apc2 = Apcd.activated(intAdvice);
22
23         Advice stringAdvice = new BeforeAdvice<String>(‘‘str’’,
24                 ‘‘stringAdvice’’, pcut2, apc2){
25             @Override
26             public void advice(String s){
27                 stringMethod(s);
28             }
29         };
30
31         /* advice 呼び出しを囲む block を作成する */
32         stringAdvice.setAdviceBlock(‘‘for (int j = 0; j < 100; j++)’’);
33
34         /* advice を登録する */
35         setAdvice(stringAdvice);
36         setAdvice(intAdvice);
37     }
38
39     public void intMethod(int i){
40         System.out.println(i);
41     }
42
43     public void stringMethod(String s){
44         System.out.println(s);
45     }
46 }
```

図 3.5: アドバイス間の依存関係がある場合のアスペクト

のようになる。ブロックを複数作成すると、新しくセットしたものがアドバイス呼び出しの外側のブロックになる。

このように、stringAdvice は intAdvice が活性化されてから活性化される。アドバイス間に依存関係がある場合も本システムを用いることで記述することができた。

3.3　本システムが提供する機能のまとめ

以下、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。

3.3.1　ポイントカット

本システムでは、execution ポイントカットのみサポートしている。なぜなら、HotSwap を用いた DAOP システムであるため、メソッドの中身しか書き換えることができないからである。しかし、描画処理などは execution ポイントカットのみで十分対応できるので、2.3 章の問題点を解決するために支障はない。

```
1 String[] hosts = new String[] { /* hongo001, hongo002 ..... */ };
2 Pointcut pcut = Pcd.execution(/* className */, /* methodName */, hosts);
```

図 3.6: ポイントカットの定義例

ポイントカットの指定は、図 3.6 のように行われる。1 行目でホスト名の配列を作成し、2 行目でポイントカットを作成している。ポイントカットには引数としてクラス名、メソッド名、ホスト名の配列を渡す必要がある。

3.3.2　アドバイスの活性化条件

本システムでは、通常のポイントカットのほかに、特別なポイントカットである *Activation Pointcut* を定義することによりアドバイスが活性化されるタイミングを記述することができる。また、*Inactivation Pointcut* を定義することで、非活性化のタイミングを指定することもできる。

また、Activation Pointcut (Inactivation Pointcut) は論理演算子を用いて、図 3.7 のように結びつけることができる。

3 行目の and は、&& のような意味を持つ。このように and を用いると、アドバイス呼び出しの時にまず、and の左側である、1 行目の apc の

```
1 ActivationPointcut apc = Apcd.call(className, methodName);
2 ActivationPointcut apc2 = Apcd.field(fieldName);
3 ActivationPointcut apc3 = Apcd.and(apc, apc2 /* ,apc.... */);
4 ActivationPointcut apc4 = Apcd.or(apc, apc2, apc3);
```

図 3.7: Activation Pointcut の定義例

条件が満たされているかどうかが調べられる。そこで条件を満たさなければアドバイスは呼び出されず、次に呼び出しを試みるまで活性化はされない。そして、apc の条件が満たされてはじめて 2 行目で作成された apc2 の条件が調べられる。そこで apc2 の条件が満たされなければ、活性化はされず、次に呼び出しを試みるまで活性化されることはない。ここで注意しなければならないのが、すでに apc の条件は突破しているため、再度調べられることはないということだ。次の呼び出しでは apc2 の条件によって活性化されるかどうかが決まる。本システムの and の意味は、「同時に条件満たす」ということではなく、「両方満たしてから」という意味になっている。

4 行目の or は、|| のような意味を持つ。このように or を用いると、アドバイス呼び出しの時に、1 行目、2 行目の apc と apc2 の条件が調べられる。左から評価していき、どちらかを満たした場合アドバイスは活性化される。apc と apc2 の両方が満たされない場合にだけ、活性化は行われない。

Activation Pointcut

本システムが提供する Activation Pointcut は以下のような種類がある。

- **call**
  call Activation Pointcut は、引数として クラス名・メソッド名 を指定することで、ポイントカットしている全てのホストで指定したメソッド呼び出しが起こった後でアドバイスを活性化させるようにする。

  また、ホストを引数として指定することで、ポイントカットの一部のホストや全く別のホストのメソッドを基準とすることもできる。

- **field**
  field Activation Pointcut は、引数としてフィールド名を指定することで、指定したフィールドの値が一致した後でアドバイスを活性化させるようにする。

  現在は int 型のフィールドのみサポートしている。

- **woven**
  woven Activation Pointcut は、ポイントカットしているすべてのホスト上にアドバイスが準備されてから活性化されるようにする。

- **activated**
  activated Activation Pointcut は、引数としてアドバイスを指定することで、指定したアドバイスが活性化された後で、活性化されるようにする。

- **and, or**
  Activation Pointcut は論理演算で結びつけることができ、それぞれ以下のような意味になる。

  - **and**
    and で結びつけられた Activation Pointcut は、左から評価されていき、条件を満たさない限り右の条件を調べることはない。また、一度条件をクリアした Activation Pointcut は再度評価されることはなく、右の条件を調べるようになる。
  - **or**
    or で結びつけられた Activation Pointcut は、毎回全ての条件を調べることとなる。ひとつでも条件を満たすようになればアドバイスは活性化される。

Inactivation Pointcut

本システムが提供する Inactivation Pointcut は以下のような種類がある。

- **call**
  Activation Pointcut と同じように、引数としてクラス名・メソッド名を指定することで、ポイントカットしている全てのホストで指定したメソッド呼び出しが起こった後でアドバイスが無効となる。

  また、ホストを別に指定することも可能。

- **field**
  Activation Pointcut と同じように、引数としてフィールド名を指定することで、指定したフィールドの値が一致した後でアドバイスを無効にする。

3.3.3　アドバイス

本システムは、アドバイスとして before, after アドバイスが使用できる。around アドバイスは、現在サポートしていない。また、現在は、1 つのポイントカットにつき 1 つのアドバイスのみしか織り込むことができないようになっている。

```
1  Advice intAdvice = new BeforeAdvice (/* argument */, ‘‘intAdvice’’,
2          pcut) {
3
4      @Override
5      public void advice(int i) {
6          /* advice body */
7      }
8
9  };
10
11 Advice stringAdvice = new AfterAdvice<String> (/* argument */,
12         ‘‘stringAdvice’’, pcut, apc, iapc) {
13
14     @Overried
15     public void advice(String s) {
16         /* advice body */
17     }
18
19 };
```

図 3.8: アドバイスの定義例

アドバイスは、図 3.8 のように定義することができる。アドバイスを定義するときには、引数として、呼び出し側で用いられるパラメタ名、アドバイス名、そしてポイントカットを渡さなければならない。Activation Pointcut と Inactivation Pointcut は指定しなくてもアドバイスを定義することはできる。1, 2 行目のようにポイントカットのみを指定することもでき、11, 12 行目のように全て指定することもできる。また、Activation Pointcut か Inactivation Pointcut のみを指定してアドバイスを定義することもできる。

アドバイスボディーは、5-7 行目や 15-17 行目のようにオーバーライドして定義する。Advice クラスには、基本データ型や基本データ型の配列が引数となっている advice() メソッドがオーバーロードされているので、5 行目のように int 型が引数のアドバイスをオーバーライドできる。また、Object 型が引数となっている advice() メソッドもあり、こちらはキャストすることで任意のオブジェクトが引数として扱うこともできるが、11 行目のようにジェネリックを使うことで、15 行目のように String 型など

任意のオブジェクトが引数となっているアドバイスも定義できる。

第4章　実装

この章では、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 (以下エージェントクラスと呼ぶ) が、 JVM 上で実行されているプログラムを計測できるようにする機構を提供している。そして、対象となる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ための枠組みが提供されている。バイトコード変換の方法としては以下の二つの方法がある。

- クラスがロードされる過程に割り込み、バイトコードを変換する。(以下「ロードタイムの変換」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)

ロードタイムの変換は、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。本システムでは、動的にバイトコードを変換するため、再定義による変換を用いて実装されている。

以下、java.lang.Instrument パッケージの説明をする。

4.1.1 Instrumentation インタフェース

このインタフェースは、エージェントクラスが Java コードを計測するための機構を提供しており、バイトコード変換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で、ロードタイムの変換に用いるメソッドとしては、addTransformer(), removeTransformer() などがあり、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() があり、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を再定義し、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可否を返す。

4.1.2　premain メソッド

Instrument API を使用するためには、上で説明した Instrumentation インスタンスをエージェントクラスが取得する必要がある。そのために用意されているのが premain メソッドである。このメソッドは、引数として String 型と Instrumentation 型の変数を受け取る。エージェントクラスにはこのメソッドを実装しておかなくてはならない。そして、オプションを指定して計測対象のアプリケーションを実行することで、計測対象のアプリケーションの main メソッドが呼ばれる前に、この premain メソッドが呼ばれるようになる。

-javaagent オプション

エージェントクラスのコードは JAR ファイルにまとめることになっている。そして、-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。また、この JAR ファイルのマニフェストには、属性 Premain-Class が含まれる必要があり、この属性の値で premain メソッドが実装されているエージェントクラスを指定する。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パスを指定することができ、Can-Redefine-Classes オプションを true とすることで再定義による変換を許可するようにできる。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 としている。

-javaagent オプションは、次のように指定する。

java -javaagent:*jarpath[=options]*

jarpath は、エージェントの JAR ファイルのパスであり、*options* は premain クラスの一つ目の String 型の引数として渡される値となる。

4.1.3　ロードタイムの変換

premain メソッドで得られた Instrumentation インスタンスで addTransformer() メソッドを呼び、トランスフォーマーを登録することでロード

タイムの変換を行うことができる。図 4.1 のようなエージェントクラスを用意すると、ClassFileTransformer の transform() メソッドがクラス定義の度に呼び出されるようになる。そこで、19 行目のように新しいバイトコードを返すことで、ロードタイムのバイトコード変換が行われる。transform() メソッドは、全てのクラス定義の度に呼び出されるので、15 行目で変換したいクラスかどうかを調べて変換している。変換しない場合は 21 行目のように null を返せばよい。この例では HelloWorld クラスを書き換える場合を示している。

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

図 4.1: ロードタイムの変換例

4.1.4　再定義による変換

premain メソッドで得られる Instrumentation インスタンスで 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 インスタンスをエージェントクラスのフィールドに保持しておくと、エージェントクラスを通じて書き換えを行うことができる。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。

簡単な動的変換を行うアプリケーションとして GUI を用いてバイトコード変換するものを図 4.2 のように実装することができる。

premain() メソッドでは、8, 9 行目で動的変換を行うための GUI を作成している。このとき、エージェントクラスを引数として渡すことで、AgentFrame クラスがエージェントクラスを通してバイトコード変換を行うことを可能にしている。13 行目のエージェントクラスのコンストラクタで、 Instrumentation インスタンスの参照がフィールドに代入されている。これでエージェントクラスは動的な変換をいつでも行うことができるようになる。

作成した GUI は 16-23 行目の weave() メソッドを呼ぶことで書き換えを行う。このメソッドでは、引数から新しいバイトコードを生成し、そこから 36-43 行目の処理に移る。ここでは、ClassDefinition インスタンスが生成され、この配列を引数として、Instrumentation インタフェースの 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　バイトコードの生成

java.lang.instrument パッケージには、既存のバイトコードを変換したり、生成したりするような API は含まれていない。よって、バイトコードの変換を行うために、変換先となるバイトコードを生成しなくてはならない。

javassist [2, 17] を用いると、変換する新しいバイトコードを生成することができる。javassist とは 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、クラス構造を抽象化した API があるので、プログラマはバイトコードの詳しい知識を必要としないといった特徴がある。

例えば、図 4.2 の 25-34 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の頭にコードを挿入するようになっている。

まず、5,6 行目で引数のクラス名から、クラスを表す CtClass インスタンスを生成している。そして、12 行目で引数のメソッド名からメソッドを表す CtMethod インスタンスを生成している。次に、14-18 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、21 行目でコードを挿入している。24 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  public class Agent {
2      private Instrumentation instrumentation;
3
4      public static void premain(String agentArgs, Instrumentation inst) {
5          Agent agent = new Agent(agentArgs, inst);
6
7          /* create redefine application GUI */
8          AgentFrame agentFrame = new AgentFrame(agent);
9          agentFrame.setVisible(true);
10     }
11
12     public Agent(String agentArgs, Instrumentation inst) {
13         this.instrumentation = inst;
14     }
15
16     public void weave(String className, String methodName, String code) {
17         try {
18             byte[] woven = getInstrumented(className, methodName, code);
19             redefineClass(instrumentation, className, woven);
20         } catch (Exception e) {
21             e.printStackTrace();
22         }
23     }
24
25     private byte[] getInstrumented(String className, String methodName,
26             String code) throws NotFoundException,
27             CannotCompileException, IOException {
28         /*
29          * create bytecode
30          */
31
32          return /* newByteCode */;
33
34     }
35
36     private void redefineClass(Instrumentation inst, String className,
37             byte[] byteCode) throws ClassNotFoundException,
38             UnmodifiableClassException {
39         ClassDefinition def =
40             new ClassDefinition(Class.forName(className), byteCode);
41         ClassDefinition[] defs = new ClassDefinition[] { def };
42         inst.redefineClasses(defs);
43     }
44 }
```

図 4.2: 再定義による変換例

```
1  private byte[] getInstrumented (String className, String methodName,
2          String code) throws NotFoundException,
3          CannotCompileException, IOException {
4
5      ClassPool classPool = ClassPool.getDefault();
6      CtClass ctClass = classPool.get(className);
7
8      ctClass.stopPruning(true);
9      if (ctClass.isFrozen())
10         ctClass.defrost();
11
12     CtMethod ctMethod = ctClass.getDeclaredMethod(methodName);
13
14     ClassPool newClassPool = new ClassPool();
15     newClassPool.appendSystemPath();
16     CtClass newClass = newClassPool.get(className);
17
18     CtMethod newMethod = newClass.getDeclaredMethod(methodName);
19
20     /* insert code */
21     newMethod.insertBefore(code);
22
23     /* set new method */
24     ctMethod.setBody(newMethod, null);
25
26     return ctClass.toBytecode();
27 }
```

図 4.3: javassist を用いたバイトコード作成例

4.2　本システムの設計

本システムは、Instrument API を使用するクラスであるエージェントクラスを Java RMI のリモートオブジェクト呼び出しで呼ぶことで、分散動的バイトコード変換を行っている。バイトコード生成は javassist を用いて行っている。以下、本システムの実装について述べる。

4.2.1　アスペクトの織り込み

本システムは、Java RMI を用いたリモートオブジェクト呼び出しでアスペクトの記述から作成したコードを挿入することで、アスペクトの織り込みを実現している。

織り込み命令が起こると、図 4.4 の上段のように、まずアドバイス本体を rmiregistry に bind する。織り込み先のプログラムは、この bind されたアドバイスをリモートオブジェクトで呼び出すことになる。次に、本システムは、実際に挿入するコードとなる、rmiregistry を取得するコードやアドバイス呼び出しのコードを作成する。このとき、図 3.5 の 32 行目のようにアドバイスに対して setBlock() が呼ばれていたら、ブロックも作成する。また、Activation Pointcut が指定されていれば、そのインスタンスから条件文を作成し、一緒に挿入することになる。挿入するコードが完成したら、スレッドを用いて並列にコード挿入作業が始まる。図 4.4 の上段右側のように、各ホストのエージェントクラスは rmiregistry に bind されている。その、各ホストのエージェントクラスを通すことで織り込みが行われる。織り込み命令がリモートのエージェントクラスに届くと、図 4.4 のようにエージェントクラスは redefineClasses() メソッドを呼び、アドバイス呼び出しコードを挿入する。そして、アドバイスが活性化されると (Activation Pointcut が指定されていなければ織り込まれた後すぐ)、リモートのアドバイスを呼び出すようになる。最後に、アドバイスが活性化された後で、挿入したコードの中で無駄な部分が取り除かれる。例えば、Activation Pointcut から生成された条件文は活性化された後では必要ないので、アドバイス呼び出しに必要なコードのみに置き換えられる。

4.2.2　活性化

Activation Pointcut を用いると、挿入するアドバイス呼び出しコードに if 文が追加される。コードが挿入されたクラスは、アドバイスを活性化してもよいかどうかをアスペクトのサーバーに対して尋ねることになる。そこで活性化の条件がそろっていなければ、アドバイスを呼び出すこ

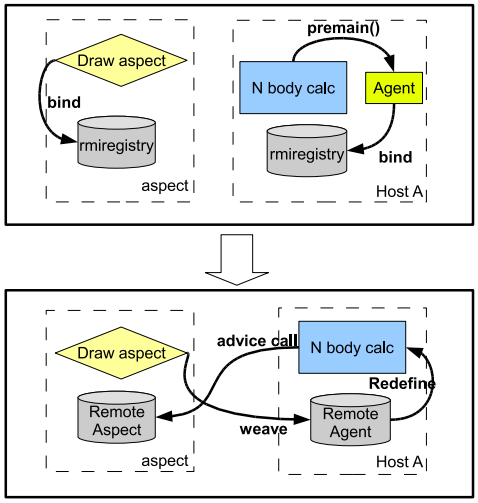

図 4.4: アスペクト織り込みの様子

となく次の呼び出しまで if 文は挿入されたままとなる。そして、アドバイスが活性化されると、if 文は取り除かれる。

```
1 Registry registry = LocateRegistry.getRegistry(/* host name */);
2 CallServer callServer = (CallServer) registry.lookup(“CallServer”);
3 if ( callServer.ready() ) {
4       Advice advice = (Advice) registry.lookup(“Advice”);
5       advice.advice(/* argument */);
6 }
```

図 4.5: call Activation Pointcut を用いた時に挿入されるコード

実際に、call Activation Pointcut を用いてアスペクトの織り込みを行うと 図 4.5 のようなコードを挿入することになる。1 行目でアスペクトのサーバの registry を取得し、2 行目で call Activation Pointcut の活性化条件を管理するリモートオブジェクトを取得している。そして 3 行目でアドバイスが活性化されているかを確認し、されていれば、4 行目でアドバイスのリモートオブジェクトを取得、5 行目で実行となっている。

アドバイスが活性化されると、1，4，5 行目のみ残しあとは取り除かれるので、無駄なコードが残り続けることはない。

以下、それぞれの Activation Pointcut の仕組みについて述べる。

call

call Activation Pointcut では、引数として指定した、アドバイスを活性化するときの基準とするメソッド呼び出しにもコード挿入を行うことで活性化の判断を行っている。以下、この基準とするメソッドを基準メソッドと呼ぶことにする。

call は、アドバイス呼び出しについている if 文 での活性化確認の呼び出しと、基準メソッドでの呼び出しの 2 つの呼び出しを用いて活性化の判断を行う。call の仕様は、

> 引数として クラス名・メソッド名 を指定することで、ポイントカットしている全てのホストで指定したメソッド呼び出しが起こった後でアドバイスを活性化させるようにする。

となっていた。よって、基本的には、基準メソッドでの呼び出しが起こったどうかをチェックしておき、全てのホストで基準メソッドが呼ばれた後、アドバイスを活性化すればいいように思える。しかし、この Activation Pointcut は同期に用いられるものであるため、単純に考えるわけにはいかない。アスペクトが織り込まれるタイミングは全てのホストでバラバラなため、もちろん基準メソッドへのコード挿入も同時には起こらない。基準メソッドは同期に用いられるメソッドを指定するのが一般的であり、それを基準にすれば同期してアドバイスを活性化できると考えられる。しかし、図 4.6 の左側のような織り込まれ方と実行がされた場合、“全てのホストで基準メソッドが呼ばれた後のアドバイスから活性化” というアルゴリズムではうまくいかないことが分かる。ホスト B の 2 回目のアドバイス呼び出しの時には、確かに全てのホストの基準メソッドの呼び出し後になっているが、同期に合わせた活性化はできていない。ホスト A での基準メソッドへのコード挿入が遅れてしまった上に、実行の順番に差ができてしまったことでこのような不具合が起こっている。

この原因となっているのは、基準メソッドの呼び出しの瞬間を参照していることである。基準メソッドから復帰するときには、同期が行われた後であることが期待できるので、図の右側のように基準メソッドからの復帰を参照するようにする。こうすることで、このような不具合は起こらないようになる。

その下で、以下のステップを踏んで活性化を行っている。

1. 全てのホストにアドバイスが準備されるのを待つ。
2. 基準メソッドが呼ばれたかどうかをチェックし始める。
3. アドバイスの条件文で基準メソッドが全てのホストで呼び終わっているかをチェックする。

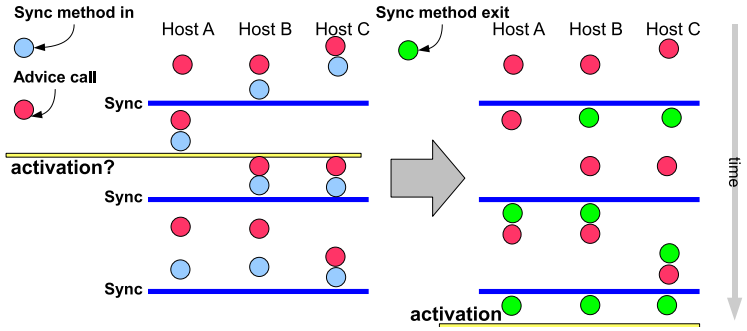

図 4.6: 同期した活性化がうまくいかない場合

4. 呼ばれていれば活性化。

5. 呼ばれていなければ、基準メソッドが呼ばれたかどうかのチェックをリセットする。

このようなアルゴリズムで活性化を行うと、基準メソッドによってアプリケーションの同期がとられている場合は、その同期に合わせてアドバイスを活性化することができるようになる。図 4.6 の右の例だと、ホスト A での 2 回目のアドバイス呼び出しの試みの時に、全てのホストで基準メソッドの呼び出しがチェックされていないため、活性化されない。そして、この例では最後の基準メソッドの復帰の後で活性化される。ホスト C の 2 回目の基準メソッドからの復帰が遅れてしまっていて、このときすでに他のホストでアドバイス呼び出しが試みられてしまっているが、それらのホストの基準メソッドのチェックがリセットされているので、活性化のタイミングは遅れるが、不具合になることはない。

field

field Activation Pointcut は、アドバイスを呼び出すかどうかの条件文で、実際に field の値を送信し、その値を使って活性化するかどうかの判断をしている。field Activation Pointcut を使った場合に挿入されるコードは図 4.7 のようになっている。

1 行目で registry を取得し、2 行目で field Activation Pointcut の活性化条件を管理するリモートオブジェクトを取得している。そして 3 行目で、指定したフィールド名のフィールドを引数として活性化条件を管理するリモートオブジェクトに渡している。そして、アドバイスが活性化され

```
1 Registry registry = LocateRegistry.getRegistry(/* host name */);
2 FieldServer fieldServer = (FieldServer) registry.lookup(‘‘FieldServer’’);
3 if ( fieldServer.ready(/* field name */) ) {
4       Advice advice = (Advice) registry.lookup(‘‘Advice’’);
5       advice.advice(/* argument */);
6 }
```

図 4.7: field Activation Pointcut を用いた時に挿入されるコード

ると 4，5 行目の実行になり、1，4，5 行目を残して他のコードは取り除かれる。

フィールドの値が送信されると、まず、全てのホストからフィールドの値が送信されているかチェックする。全てのホストから送信されていれば、その値が全て一致しているかを調べる。一致していなければアドバイスは活性化されない。もし一致していれば、その後のアドバイスは活性化される。このように、field Activation Pointcut を用いれば、フィールドの値で同期しているアプリケーションにアスペクトを織り込む時に、アドバイスを同期的に活性化させることができる。

activated

アドバイスには、活性化されたかどうかを返す isActivated() メソッドが実装されている。activated Activation Pointcut は、引数として Advice を渡すことで、そのアドバイスが活性化されているかを isActivated() メソッドで調べ、true が帰ってくれば活性化するようになっている。

woven

woven Activation Pointcut は、全てのホストで条件文が実行されたかどうかをチェックしている。そして、全てのホストで条件文が実行されていなければ、活性化はされず、全てのホストですでに実行されていたならば、活性化されるようになっている。

and, or

and や or を使うと、挿入される条件文が論理演算子で結合されるようになる。例えば、call と field を and で結びつけた場合は、図 4.8 のようなコードが挿入される。

```
1 Registry registry = LocateRegistry.getRegistry(/* host name */);
2 CallServer callServer = (CallServer) registry.lookup(‘‘CallServer’’);
3 FieldServer fieldServer = (FieldServer) registry.lookup(‘‘FieldServer’’);
4 if ( callServer.ready() && fieldServer.ready(/* field name */) ) {
5     Advice advice = (Advice) registry.lookup(‘‘Advice’’);
6     advice.advice(/* argument */);
7 }
```

図 4.8: and を用いた時に挿入されるコード

4.2.3　アドバイスのアンウィーブ

アドバイスのアンウィーブは、ポイントカットのメソッドを元のバイトコードに書き戻すことで行っている。

アンウィーブ命令が起こると、スレッドを用いて並列にエージェントクラスにバイトコードの書き戻し命令を送る。そのとき、Inactivation Poincut を用いていた場合は、いきなり書き戻しはせずに、まず非活性化のためのコードを挿入することからはじまる。

4.2.4　非活性化

Inactivation Pointcut を用いると、元のバイトコードに書き戻す前に、図 4.9 のようなコードが挿入される。このコードの挿入により、同期したアドバイスの非活性化が行われてからアンウィーブすることが可能となる。

```
1  Registry registry = LocateRegistry.getRegistry(/* host name */);
2  CallServer callServer = (CallServer) registry.lookup(‘‘CallServer’’);
3  UnWeaveServer unWeaveServer =
4                  (UnWeaveServer) registry.lookup(‘‘UnWeaveServer’’);
5
6  if ( callServer.ready() ) ) {
7      unWeaveServer.call();
8  }
9
10 if ( unWeaveServer.ready() ){
11     Advice advice = (Advice) registry.lookup(‘‘Advice’’);
12     advice.advice(/* argument */);
13 }
```

図 4.9: call Inactivation Pointcut を用いた時に挿入されるコード

1 行目で rmiregistry を取得し、2，3 行目で Inactivation Pointcut の非活性化条件を管理するオブジェクトと非活性化を行うオブジェクトの

参照を得る。そして、Activation Pointcut のときと同じように、6 行目で Inactivation Pointcut の条件が調べられる。非活性化の条件が整っていなければ、7 行目の実行はされず、10 行目の返り値も true となり通常のアドバイス実行をすることになる。アドバイスを非活性化してもよいならば、7 行目の実行に入り、このときにアドバイスが非活性化される。すると、10 行目での返り値が false となり、アドバイスが実行されなくなる。そして、アドバイスが非活性化された後で、これらのコードは全て取り除かれる。こうして、Inactivation Pointcut を用いることで同期したアンウィーブが可能となる。

第5章　実験

この章では、本システムの性能を測るための実験について述べる。

5.1　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、Instrument API の HotSwap を用いてアスペクトの織り込みを行っていた。そのため、アプリケーションを動かすときに -javaagent オプションで本システムの JAR ファイルを指定しなければならない。このことによるオーバーヘッドを計測する実験をした。以下、本システムを -javaagent オプションで指定することを “本システムを用いる” と表現する。実験環境は以下の通りである。

プログラム N 体問題アプリケーションを質点 5000 個で実行

コンパイラ JDK1.6.0

マシン InTrigger [3] 東京大学 hongo の hongo000-hongo069 の中から 50 台

実行環境 OS:Debian、CPU:Pentium M 1.8GHz、メモリ:1GB

N 体問題アプリケーションの 1 ステップの計算にかかる時間を、本システムを用いた場合と用いなかった場合でそれぞれ 600 ステップ計測した。結果は表 5.1 と図 5.1 の通りである。

表 5.1: N 体問題計算 1 ステップの実行時間 (ms)

-javaagent	平均	標準偏差
on	1929	40
off	1886	40

実験結果から、本システムを用いることにより約 2%のオーバーヘッドがあることが分かる。しかし、フックを用いた織り込みやデバッガを用いたものではオーバーヘッドが数十%にもなるため、それらと比べるとほとんどオーバーヘッドはないと言える。

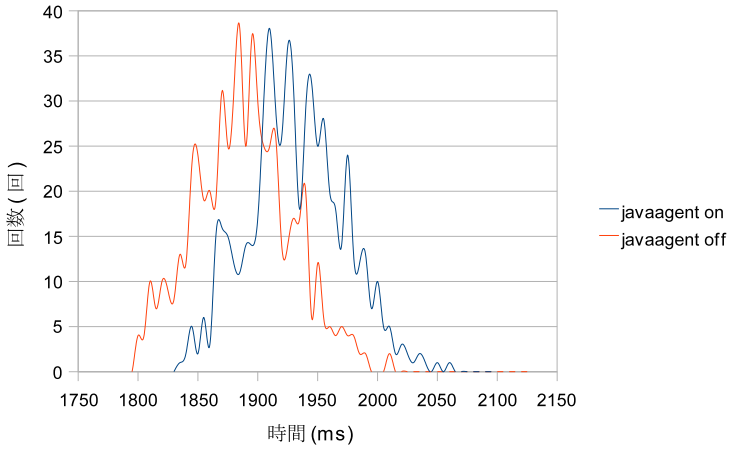

図 5.1: N 体問題計算 1 ステップの実行時間

次に、アスペクトを織り込んだ後のオーバーヘッドについて計測するために、N 体問題アプリケーションに描画機能を追加し、追加した後の 1 ステップの計算にかかる時間を計測した。N 体問題アプリケーションは、はじめから描画機能を追加してあるものと計算機能だけのものを用意し、以下の場合をそれぞれ 500 ステップ計測した。

- はじめから描画機能が追加されているアプリケーションを、本システムを用いずに実行した 1 ステップ
- はじめから描画機能が追加されているアプリケーションを、本システムを用いて実行した 1 ステップ
- 計算機能のみのアプリケーションを、本システムを用いて実行し、描画機能をアスペクトとして織り込んだ後の 1 ステップ

アスペクトを織り込むポイントカットは、N 体問題計算のメインループで実行されている calc() メソッドの実行を選択した。

実験結果は、表 5.2 の通りである。本システムを用いるか用いないかの差は、先ほどの実験と同じような結果となった。表 5.2 の 1,3 行目のように、本システムを用いている 2 つの場合ではほとんど差がないため、はじめから実装されていた描画機能と、本システムを用いて織り込んだ描画機能では、オーバーヘッドは変わらないということが分かった。このことか

表 5.2: アドバイスの実行時間 (ms)

描画方法	-javaagent	平均	標準偏差
実装済み	on	2021	50
実装済み	off	1976	48
織り込み	on	2021	48

ら、織り込み後のアスペクトは、はじめから実装されていた場合と同じ動きをしていることが確認できた。

5.2　活性化されるまでの時間

以下、アドバイスが活性化されるまでの時間を測る実験について述べる。実験環境は先ほどの実験と同じである。

5.2.1　Activation Pointcut

Activation Pointcut を用いたときの、アドバイスが活性化されるまでにかかる時間を計測した。この実験では、アスペクトのインスタンスを作成したところから、最初にアドバイスが呼ばれるまでの時間を測定している。

call, field Activation Pointcut について、それぞれ N 体問題アプリケーションに対して、合計 100 回アスペクトの織り込みを行った。実験結果は表 5.3 と図 5.2、 図 5.3 の通りである。

表 5.3: 活性化までにかかる時間 (ms)

advice	apc	MAX	MIN	AVERAGE
before	call	5284	2660	3873
	field	5198	2553	3873
after	call	7396	4721	6087
	field	7220	4602	5888

これらの Activation Pointcut は、図 5.4 のように、アドバイスが活性化されるまでに最低 1 回はアドバイスを活性化しないメソッド実行がある。これは、全てのホストにアドバイスが準備されていることを保証するために行われる。よって、織り込みを行うと、全てのホストにアドバイス

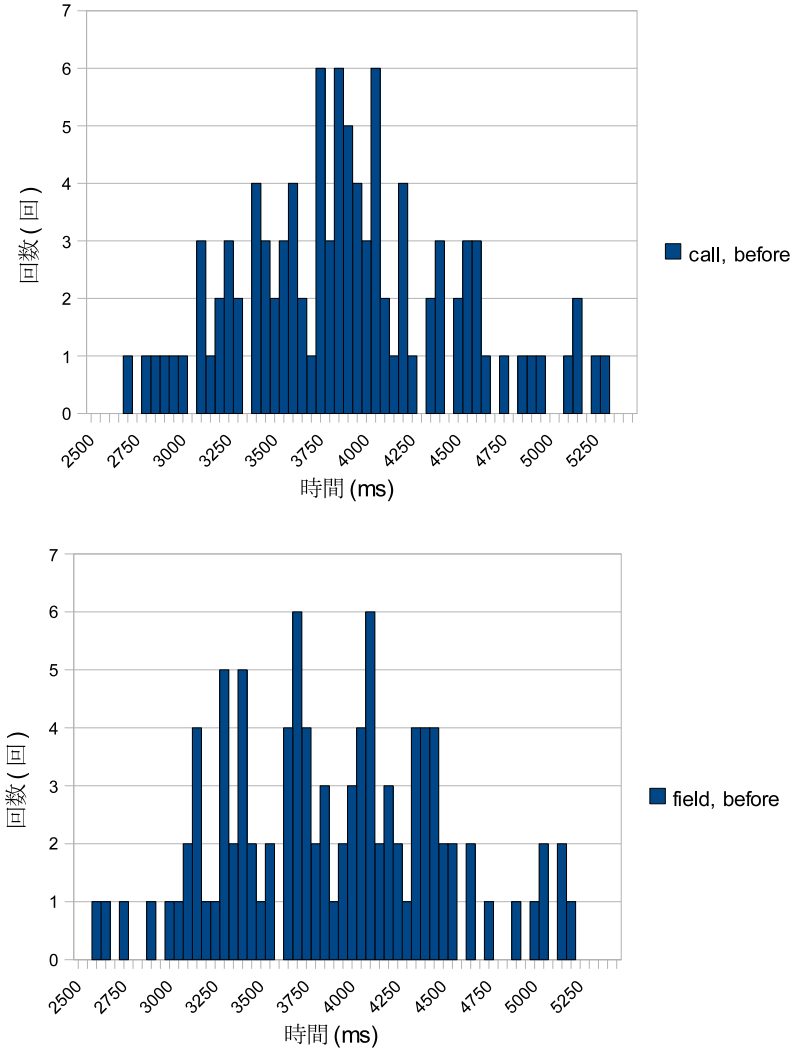

図 5.2: call, field で活性化までにかかる時間 (before アドバイス)

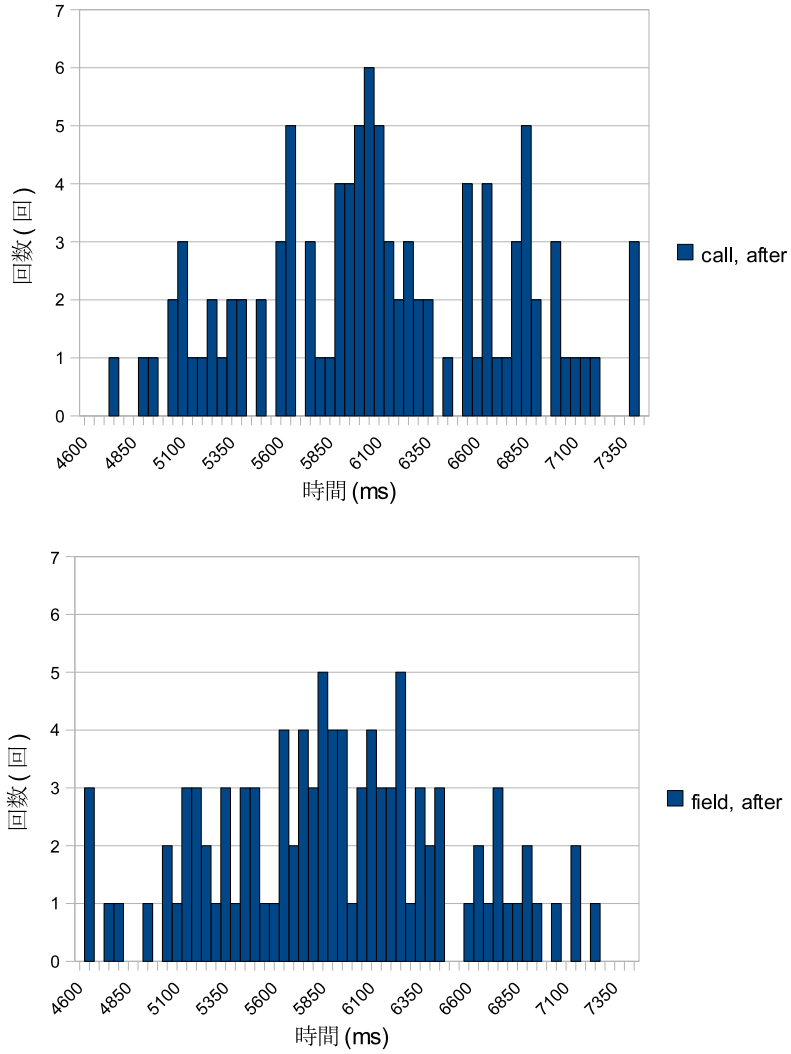

図 5.3: call, field で活性化までにかかる時間 (after アドバイス)

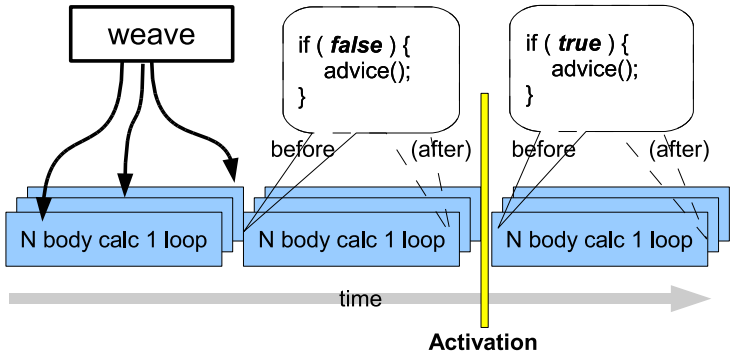

図 5.4: Activation Pointcut による活性化までの流れ

が準備され、次のメソッド実行で活性化の準備が行われ、その次の実行から活性化されるという流れになる。これを基に実験結果をみる。

before アドバイスでは、アドバイスを実行する場所がメソッドの最初なので、活性化の準備が行われた次の実行の頭で最初のアドバイスが実行される。よって、アドバイスが活性化されるまでにかかる時間は、アスペクトのインスタンスが生成されてから全てのホストにアドバイスの準備ができるまでの時間＋その次のメソッド実行が全てのホストで行われるまでの時間＋最初のアドバイス実行の時間である。after アドバイスは実行する場所がメソッドの最後なので、活性化の準備が行われた次の実行の最後で最初のアドバイスが実行される。よって、図 5.4 からも分かるように、同じタイミングでアドバイスが準備されても、before アドバイスよりも活性化までに時間がかかっている。実験結果で before、after で活性化までの時間が違うのはこれが原因である。

また、field Activation Pointcut と call Activation Pointcut を使ったときのアドバイスが活性化されるまでの時間を比較すると、before アドバイスのときはほとんど同じだったが after アドバイスでは call のほうが遅くなってしまっている。これは、図 5.4 のように、活性化までにかかるステップ数が多くなってしまう after アドバイスでは、call で指定した基準とするメソッド呼び出しも多くなってしまうためコストが高くなるからだと考えられる。

5.2.2　比較実験

比較対象として、織り込み対象の情報を利用しない方法でアスペクトの織り込みを行った。この方法では、同期を実現するためにバリア同期をしてからアドバイスを活性化するようになっている。この同期には、デッドロックの対策としてタイムアウトの時間を設定しておき、タイムアウトの時間を過ぎると、一度スレッドを開放し、次の実行で同期を試みるようになっている。この実験では、タイムアウトは、3000、1000、100 ms に設定してアスペクトの織り込みを行っている。実験は、タイムアウトが 3000 の場合を before、after アドバイスをそれぞれ織り込み、その他は before アドバイスのみ織り込みを行った。それぞれ 100 回行ったときの実験結果が、表 5.4 と図 5.5、図 5.6 である。

表 5.4: バリア同期を用いたときに活性化までにかかる時間 (ms)

timeout	advice	MAX	MIN	AVERAGE	AVE(not timeout)
3000	before	7862	601	2311	1884
	after	8823	2742	4245	3952
1000	before	4478	626	2029	1878
100	before	11085	808	4040	-

結果から分かるように、この方法では、アドバイスが活性化されるまでにかかる時間が本システムよりも短い場合が多かった。これは、バリア同期に成功した時点で活性化の条件を満たすので、アドバイスが準備できたらすぐに実行されているからである。これに対し、本システムの Activation Pointcut では、活性化の準備のために全てのホストで 1 回はアドバイスが活性化されないメソッド呼び出しがあった。よって、バリア同期を用いた場合でタイムアウトせずにアドバイスを活性化できたものは Activation Pointcut を用いたときよりも約 1 ステップ分早いことが予想でき、実際、実験結果をみると約 1 ステップ分の差があることが分かる。また、バリア同期では、タイムアウトの時間により、アドバイスが活性化されるまでが遅れてしまっているが分かる。これは、織り込みがずれてしまう場合に、アドバイス呼び出しが挿入されていないホストが存在するため、バリア同期に失敗し、タイムアウトの時間までアスペクトが織り込まれているホストでスレッドが無駄にストップしてしまうからである。このように、バリア同期を用いた活性化では、織り込みがアプリケーションの同期と同期の間に行われた場合にはすぐに活性化され、織り込みがずれてしまった場合は、タイムアウトの時間に応じて活性化が遅れることが分かる。しかし、タイムアウトが 100 ms の時だけは例外であり、活性化が遅

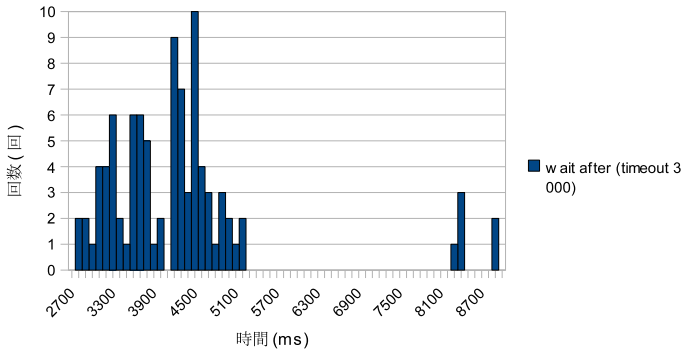

図 5.5: バリア同期を用いた時に活性化までにかかる時間 (after アドバイス)

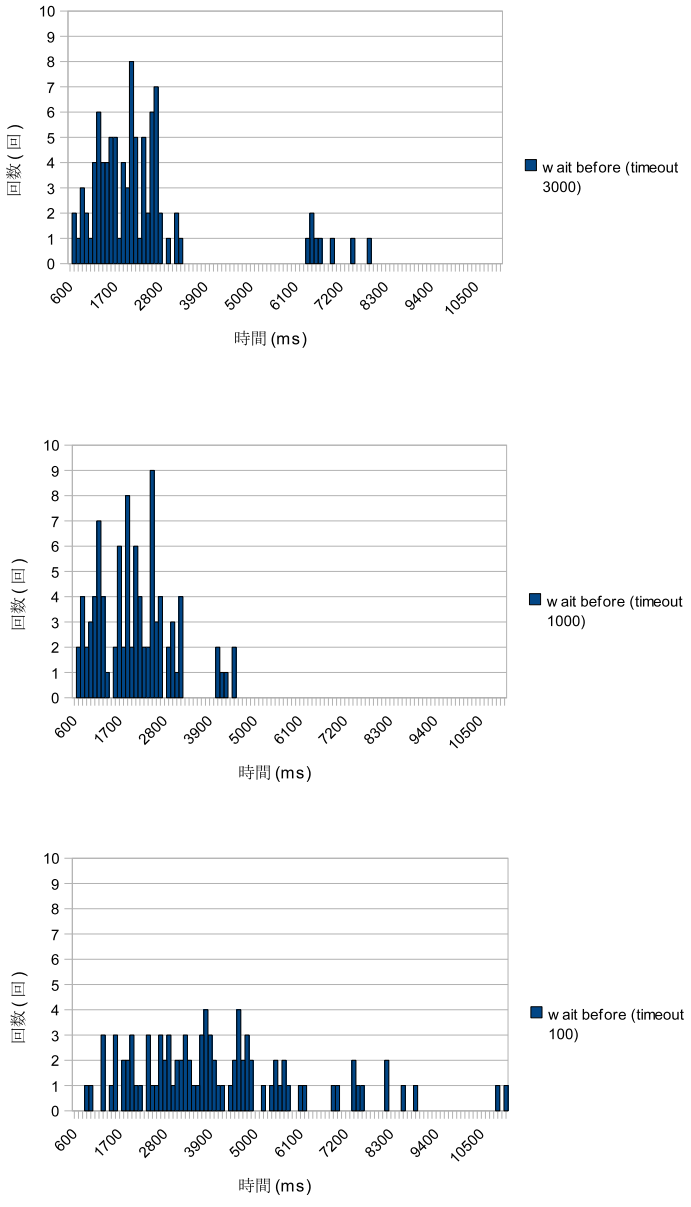

図 5.6: バリア同期を用いた時に活性化までにかかる時間 (before アドバイス)

れてしまうことが多かった。これは、タイムアウトまでが短すぎて同期が終了する前にスレッドを開放することが多かったからだと言える。このことから、タイムアウトを短くしすぎると同期が不可能になることが予想でき、実際に 50 ms で実験を行ったところ、いつまでたってもアドバイスが活性化されなかった。

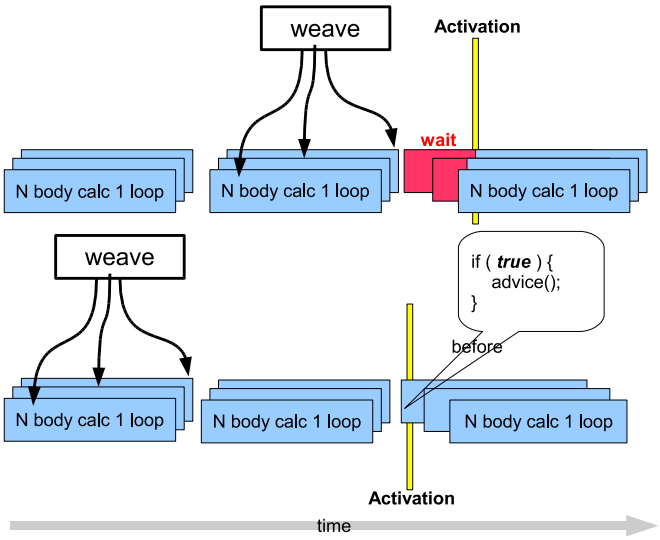

図 5.7: Activation Pointcut を用いた場合とバリア同期を用いた場合の活性化のタイミングの違い (上段：バリア同期、下段：APC)

N 体問題の場合は、アプリケーションの同期をまたいでアスペクトが織り込まれてしまうことがまれであるため、本システムの Activation Pointcut よりもバリア同期を用いたほうがいいように思えるかもしれない。しかし、本システムの Activation Pointcut とバリア同期を用いた場合の一番の違いは、同期のためにアプリケーションを止めているかどうかということだ。本システムは、アプリケーションがアドバイスを実行しようとする段階で活性化すべきかどうかを判断していたが、バリア同期では、アドバイス実行しようとすると、いったんスレッドを停止させ、全てのホストでアドバイスを実行しようとしてスレッドが停止するのを待ち、スレッドを開放するという方法で同期したアドバイスの活性化を行っている。このように、バリア同期を用いると、最初にアドバイスを実行しようとしたホストのスレッドは、最後のホストがアドバイスを実行しようとするまで停止することになってしまう。さらに、図 5.7 のように、Activation Pointcut では、一番早くアドバイスを実行しようとしたホストで活性化されるのに

対して、バリア同期の場合では一番遅くアドバイスを実行しようとしたホストで活性化されるという違いもある。また、タイムアウトの時間は 1 ステップの実行にかかる時間に依存し、図 5.6 を見ても分かるように、タイムアウトの時間を少なく設定してしまうと同期に失敗する可能性が増え、多く設定してしまうとアスペクトの織り込みがずれてしまったときに無駄にスレッドを停止させるコストが高くなってしまう。しかも、1 ステップにかかる時間を事前に調べておくことはあまり考えられないため、タイムアウトする時間を決めることは難しいと言える。この点、本システムでは、活性化までに 1 回アドバイスが実行されないメソッド呼び出しがあるため活性化は 1 ステップ分遅れてしまっているが、織り込み先のスレッドを停止させることもなく、織り込み先の実行にかかる時間による設定などもない。

以上のように、本システムで用いる Activation Pointcut では、アドバイスが活性化されるまでに、アドバイスが準備されているかを確認するための呼び出しが余分に必要となる分、活性化までに時間が余分にかかってしまうが、織り込み先のスレッドを停止させたり、織り込みがずれることによる特別な遅れが発生したりしない点で、単純に同期をするバリア同期よりも優れていると言える。

また、今回の実験ではアスペクトの織り込みがホストによってずれてしまうケースが少なかったということが、バリア同期のほうが有利に見える理由である。ネットワークなどの問題でアスペクトの織り込みがずれてしまう場合が多い時、本システムを用いたほうが、アドバイスが活性化されるまでのスピードでも、アプリケーションに与える影響の少なさでも勝ると考えられる。よって、今回のような単一クラスタ内での実験だけでなく、クラスタをまたがるような、ネットワークスピードの差が生じやすい環境で実験をするべきである。

第6章　まとめと今後の課題

6.1　まとめ

本研究では、分散アプリケーションが同期している場合に、その同期に合わせてアドバイスを活性化することができる分散 DAOP システムを提案した。

アプリケーションが同期している場合、アドバイスも同期して活性化されることが望まれることがあるが、既存の分散 DAOP システムでは、ネットワークなどの問題で全てのホストに対して一斉にアスペクトを織り込むことは不可能である。これを解決するためには、アドバイスボディーでアドバイスが “活性化” されるタイミングを調整する必要があった。しかし、活性化するべきタイミングをアドバイス内に記述することで、可読性が低下してしまうという問題がある。また、活性化するべきタイミング調べることは困難であり、タイミングを調整するためのコンテクストが利用できない場合など、アドバイスの実装で解決するには限界があった。

本システムでは、既存の分散 DAOP システムでは調整することが困難であった “活性化のタイミング” を特別なポイントカットである Activation Pointcut を用いて記述することができる。よって、開発者はアドバイスボディーに活性化のタイミングを考慮する記述をする必要がなくなる。この Activation Pointcut を用いることで、メソッド呼び出しやフィールドの値を同期の基準にしてアドバイスを活性化できる。また、アドバイス間の依存関係にも対応している。

アドバイス呼び出しのコストを抑えるため、本システムの実装には、Java Instrument API の HotSwap 機能を用いている。実験により、挿入したアドバイス実行にかかる特別なオーバーヘッドはほとんどないことが確認できた。

6.2　今後の課題

現在のシステムにはまだ解決すべき問題が残っている。以下で、その問題を確認する。

6.2.1　ネットワークの問題

本システムは、アドバイスボディーの実行や Activation Pointcut の処理など単一ホストで行っている。今回の実験では 50 台のマシンで行っていたが、台数が増加すればアクセス集中により問題が発生することが予想できる。また、台数が増加すれば、それだけネットワークに関するオーバーヘッドも増加することも予想できる。よって、アドバイスの処理や Activation Pointcut の処理を分散させてオーバーヘッドを減らすことを考えることは重要な課題である。

また、本システムを用いた場合、バリア同期よりもアドバイスの活性化に時間がかかってしまっていた。本システムは、アスペクトの織り込みがずれてしまう場合に性能を発揮できるため、ネットワークによるずれが発生しやすい環境での実験が必要である。よって、今回の実験は東京大学の InTrigger hongo のみで行っていたが、クラスタをまたがるような、ネットワークスピードの差が生じやすい場合での実験も行うべきである。

6.2.2　同期できるアプリケーションについて

この論文では、N 体問題アプリケーションに描画機能を追加することを同期しているアプリケーションへの織り込みの例として取り上げていた。この例では同期した織り込みをすることができたが、他のアプリケーションで、現在の Activation Pointcut だけで対応可能かを調べなくてはならない。そのために、さまざまなアプリケーションで、同期した織り込みが望まれるものをケーススタディとして探し、同期した織り込みには何が必要なのかを考えなくてはならない。

6.2.3　アドバイス間の依存関係への対応

現在のシステムでは、アドバイス間の依存関係として、活性化される順番を調整することが可能になっていた。しかし、アドバイスの依存関係はそれだけでは不十分であると考えられる。例えば、順番だけでなく、2 つのアドバイスが同時に活性化されるべきであったり、2 つのアドバイスが全てのホストに準備された後で活性化されるべきであったり、様々なパターンが考えられる。よって、アドバイス間の依存関係があるようなケーススタディを探し、対応するための Activation Pointcut としてどのようなものが必要なのかを考えなくてはならない。

6.2.4　ポイントカット

本システムは、ポイントカットとして execution ポイントカットのみ選択可能であった。しかし、execution ポイントカットのみで対応できない場合もあるかもしれない。よって、execution ポイントカット以外のポイントカットに対応させることも今後の課題である。また、1 つのポイントカットには 1 つしかアドバイスすることができなかったため、複数のアドバイス呼び出しを 1 つのポイントカットに対して挿入することを可能にすることも今後の課題である。

6.2.5　アドバイス

本システムでは、アドバイスとして、before、after が使えた。代表的なものとして around アドバイスに対応できていないので、使用できるようにすることも今後の課題である。

参考文献

[1] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual machine support for dynamic join points. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 83–92, New York, NY, USA, 2004. ACM.

[2] Shigeru Chiba. Load-time structural reflection in java. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 313–336, London, UK, 2000. Springer-Verlag.

[3] InTrigger. Intrigger platform. `https://www.logos.ic.i.u-tokyo.ac.jp/intrigger/registration/`.

[4] Nico Janssens, Eddy Truyen, Frans Sanen, and Wouter Joosen. Adding dynamic reconfiguration support to jboss aop. In *MAI '07: Proceedings of the 1st workshop on Middleware-application interaction*, pages 1–8, New York, NY, USA, 2007. ACM.

[5] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming, LNCS2027*, pages 327–353, London, UK, 2001. Springer-Verlag.

[6] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée. Explicitly distributed aop using awed. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 51–62, New York, NY, USA, 2006. ACM.

[7] Muga Nishizawa, Shigeru Chiba, and Michiaki Tatsubori. Remote pointcut: a language construct for distributed aop. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented*

software development, pages 7–15, New York, NY, USA, 2004. ACM.

[8] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gerard Florin. Jac: A flexible solution for aspect-oriented programming in java. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 1–24, London, UK, 2001. Springer-Verlag.

[9] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, Gérard Florin, Fabrice Legond-Aubry, and Laurent Martelli. Jac: an aspect-based distributed dynamic framework. *Softw. Pract. Exper.*, 34(12):1119–1148, 2004.

[10] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-in-time aspects: efficient dynamic weaving for java. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 100–109, New York, NY, USA, 2003. ACM.

[11] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, New York, NY, USA, 2002. ACM.

[12] AspectJ Project. The aspectj project. `http://www.eclipse.org/aspectj/`.

[13] Yoshiki Sato, Shigeru Chiba, and Michiaki Tatsubori. A selective, just-in-time aspect weaver. In *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering*, pages 189–208, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[14] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29, New York, NY, USA, 2003. ACM.

[15] Eddy Truyen, Nico Janssens, Frans Sanen, and Wouter Joosen. Support for distributed adaptations in aspect-oriented middleware. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development(to appear)*, 2008.

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

[17] 千葉 滋. Javassist home page. http://www.csg.is.titech.ac.jp/~chiba/javassist/index.html.

[18] 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, 11 2005.