スレッド移送のためのバイトコード上の変換技法

浅野　貴史　　脇田　建　　佐々　政孝
東京工業大学大学院　数理・計算科学専攻*
{asano,wakita,sassa}@is.titech.ac.jp

概 要

本研究は、バイトコード上のコード変換技術を利用してJavaにスレッド移送機能を提供する。これによって、多くのモバイルエージェントシステムが提供できていないエージェントの移動の透明性をプログラマに与えることができる。本稿では、バイトコード上の変換の利点やその利点をいかしたコード変換技術や解析方法を紹介する。また、応用例としてスレッド移送を実現しない既存のモバイルエージェントシステムへスレッド移送機能を提供するメカニズムを紹介する。

1　はじめに

近年、ネットワークを介して計算 (処理) をすることが一般化するとともに、その計算モデルも多様化している。そのひとつに、ユーザーの「代理人」となるエージェントがネットワークを介してタスクを処理する、**ネットワークエージェント**というモデルがある。ネットワークエージェントは、マルチエージェントやAIやインターフェースエージェントなど様々な要素技術が組み合わさって成り立つ。

ネットワークエージェントの重要な性質のひとつにエージェントの**移動性**がある。エージェントが移動性をもつ利点は、自律的処理や通信の効率化や遠隔資源の獲得などである [Che97]。エージェントの移動性に比較的特化したシステムが **モバイルエージェントシステム**である [Lan98, Gla98, 佐藤 98]。エージェントの移動性の重要度を証明するように、すくなくとも 70 を越えるシステムが産業界や学術分野から提案されている [Hoh99]。この中の多くは Java 言語をベースとしている。これは、Java が広く普及していることに加えて、モバイルエージェントを実装する際に必要となる技術 (Code Mobility, Portability, Safety, Security など) の多くを標準で備えているからである。

既存の Java ベースのモバイルエージェントシステムの多くは、移送対象が非活性なオブジェクトに制限されており、**弱い移送**をサポートすると言われる。これに対して、スレッドの状態およびそのスレッドが扱うオブジェクトをまとめて移動する移送方式は**強い移送**と呼ばれる [Cug97]。弱い移送システムでは、移送時にプログラムコードとデータだけを移動し、そのデータを計算した経過や次にやるべき処理を忘れてしまう。プログラマは、移動後の処理や移送してほしいデータを明示することを要求される。このため、エージェントに移動性を持たせるために、アプリケーションに大幅な変更が強いられる。一方、強い移送システムでは、プログラムコードやデータに加えて、スレッド及びその実行状態を移送することができるため、このような問題が発生しない。

本研究で提案する **Jamith** (**Ja**va with

*Dept. of Mathematical and Computing Sciences, Graduate School of Tokyo Institute of Technology

Migratory **Th**reads) システムは、スレッド移送（強い移送）を実現することによって、エージェントの移動の透明性を提供する。Jamith では、スレッド移送を実現するために**バイトコード上の変換技法**を採用した。バイトコード上でコード変換する利点は、(1) 任意の Java 言語のプログラムおよび、そのクラスファイルやライブラリを扱えること、(2) ロード時のコード変換が可能であること、(3) スレッド移送のためのコード変換に伴う実行性能の劣化を最小限に押さえていること、である。

(1) の特長を活かし、Jamith を既存のモバイルエージェントシステムに組み込むことにより、弱い移送のみに制限されていたシステムに強い移送機能を与えることができる。このことは、AgentSpace[佐藤 98] に組み込むことで実現可能であることを確認した。

(2) の特長は、実行時情報を利用した効率のよいコードへの変換やスレッドの移送先でのコード変換を可能にする。

(3) の成果として、HotSpot[Sun] で複数のアプリケーションで実行速度を計測したところ、最も性能を劣化させた場合でも、オーバヘッドは 43%に押さえられた。これは、既存の研究でのオーバヘッドが 177%であるのに比べると、飛躍的な向上である。また、変換後に付加されるコード量も既存の研究に比べて 40%ほど減らすことができた。

論文の構成は、以下の通りである。2 章では Jamith におけるスレッド移送へのアプローチを述べ、それを他の研究と比較する。3 章で Jamith の実装を述べ、4 章では、応用例として、既存のモバイルエージェントシステムへ Jamith を組み込んで強い移送のモバイルエージェントシステムを構築できることを示す。5 章で Jamith によって変換されたコードの性能評価を行う。6 章で関連研究、7 章でまとめと今後の課題について述べる。

2 スレッド移送へのアプローチ

スレッド移送とは、スレッドをその実行状態を維持したまま、他のホストに移動させることである。スレッド移送を実現するには、スレッドの実行状態 (**スレッド状態**) であるプログラムカウンタ (PC) やスタック、およびスタックから指されているオブジェクトを移動先に送信し、移動先でスレッド状態を再構成する必要がある。

モバイルエージェントシステムがスレッド移送機能を実現する際には、次の要件を満たすことが重要である。

可搬性　モバイルエージェントシステムが広く利用できるためには、システムがさまざまなプラットフォームで実行できること（可搬性）が重要である。Java における可搬性は、システムの実装が Java に準拠することを意味する。可搬性の利点には、JIT コンパイラを用いた高速な実行、RMI をはじめとする Java のソフトウェア資産の利用がある。

性能　スレッド移送を実現する機構によって、アプリケーションの実行性能が、通常の Java システム上での実行性能に比較して、著しく劣化しないことが望ましい。

通信量　移送に要する通信量が少ないことが望ましい。スレッド移送の通信量は、移送する実行状態の量とコード量からなる。スレッド移送システム同士の比較においては、実行状態の量は、ほぼ同じであると考えられるために、ここではコード量だけを比較対象とする。

バイトコードへの対応　Java では、さまざまなパッケージがバイトコードにコンパイルされたクラスやライブラリとして配布、またはネットワークからダウンロードされることが一般的である。それらを利用したアプリケーションに対応でき

ることが望ましい。また、バイトコードを扱うことができれば、他の言語からのバイトコードへのトランスレータ（コンパイラ）を利用して、その言語にスレッド移送機能を与えることも可能である。

これらの要件をもとに、既存のスレッド移送システムの実現方法と Jamith の採用するバイトコード上のコード変換手法を比較する。Java では、スレッド状態を直接的にアクセスするためのバイトコード命令を用意していないので、スレッド移送システムは、それぞれなんらかの工夫を凝らしてスレッド移送を実現している。

VM の変更　JavaVM の変更やネイティブメソッドの追加によりスレッド状態へのアクセスを可能にしたシステムである [Pei97, Gra98, Shu]。しかし、可搬性を損なっており、エージェントが特定の JavaVM でしか動作しないことや任意の JIT を利用できないために性能面に欠点がある。一方、アプリケーションコードに変更を必要としないために、コード量はコンパクトであり、またバイトコードへの対応もできる。

エージェント言語　Java 言語の上でスレッド移送機能を持った言語を構築する [Ohs97]。システムは、Java で実現されるため可搬性は維持されるが、新しい言語システムの上でエージェントが動くため、通常の Java プログラムと比べると性能が劣る。システムは対象言語が Java でないので、コード量やバイトコードへの対応は比較できない。

ソースコード変換　ソースコードを変換することでスレッド状態の保存、復元を実現する [Fün98, Sek99, Hoh98]。ソースコード上の変換であるために、可搬性は維持される。しかし、変換の際に言語的な制約に縛られるために、変換後のコードの性能劣化やコード量の肥大化が問題になる。また、ソースコードのないクラスやライブラリを扱うことはできない。

バイトコード変換（Jamith）　ソースコード変換の方式をバイトコード上の変換を用いて実現する。エージェントは、バイトコードレベルで変換されるために可搬性を維持できる。性能面では、ソースコード変換よりも効率的なコード変換が可能な上、変換によって付加されるコード量を小さくおさめることが可能である。また、バイトコードを実行時（ロード時) に変換できるために、実行時情報を利用した効率のよいコードへの変換やネットワーク状況に応じた移送メカニズムが実現でき、システム全体の性能向上を図ることが可能である（3.4 節参照)。

上記に述べた各アプローチの比較を表 1 にまとめた。スレッド移送システムに求められる要件をすべて満たすためには、バイトコード上でコード変換することが最も適している。

表 1: スレッド移送へのアプローチの比較

	可搬性	性能	通信量	バイトコードへの対応
VM の変更	×	×	○	○
エージェント言語	○	×	-	-
ソースコード変換	○	△	×	×
バイトコード変換 (**Jamith**)	○	○	△ (○)	○

3　Jamith システム

Jamith は、バイトコード上の変換ツールである。クラスファイルを入力としてもらい、それをパーズし、コード変換するための解析を行った後、**スレッド移送を実現するコード**へと変換を行う。スレッド移送を実現するコードとは、スレッド移送時にスレッド状態を保存し、移送後にスレッド状態の復元と実行の再開を行うコードである。スレッド状態の送信は、Jamith が利用す

る実行時システムが行う。つまり、実行時システムが実現する通信手段によって Jamith によって保存されたスレッド状態が送信される。

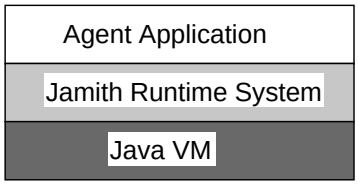

図 1: ソフトウェア階層

図 1 に Jamith が想定するソフトウェア階層を示す。Agent application は、Jamith によって変換されたアプリケーションを指す。その下位に位置するのが Jamith が利用する実行時システムである。Jamith は、実行時システムに依存しないように設計し、実行時システムとのインターフェースを定めることで、実行時システムの変更、交換に対応できる。

以下で、Jamith におけるコード変換手法やコード解析方法などの実装について述べる。

3.1 Jamith における変換

Jamith で採用しているコードの変換方式は、[Fün98, Sek99] で提案されたソースコード変換技術と同様である。それは、スレッド移送を起こす**移送命令** (例、go 命令) をもとに行われる。つまり、移送命令の直前で、スレッド状態の保存を行い、移送後にスレッド状態の復元と移送命令の直後からの実行の再開を行うコードに変換する。まず変換方式を概説し、その後に Jamith におけるバイトコード上の変換の特徴について述べる。

3.1.1 変換方式

前述のように、Java ではバイトコードレベルからスレッドの実行状態にアクセスできない。このため、PC やスタックを保存や復元する直接的な手段がない。したがって、PC やスタックの保存と復元はそれぞれ以下のアプローチを取る。

PC については、各メソッドごとにメソッド中のプログラムポイントの代用となる変数 (PC 変数）を用意する。そして、移送が起きるプログラムポイントごとにこの変数の値を設定し、移送後の状態復元時に、その変数の値によってプログラム再開ポイントを決める。

スタックは、フレーム単位でフレーム内の実行状態であるローカル変数とオペランドスタックをフレーム単位に用意するオブジェクトに保存していく。スレッド状態の復元は、フレームごとに保存先のオブジェクトから移送前と同じフレーム状態を復元し、移送前と同じメソッド呼び出しを再現することで移送前と同じスタックを作り出す。

以下、Jamith においてバイトコード上で行う変換例を、Java-like な疑似コードで説明する。

3.1.2 スレッド状態の保存

スレッド状態の保存は、移送が起こったときのみ行われるのが理想である。これを実現するために、[Fün98, Sek99] で提案される手法は、Java の例外処理 (try-catch 構文）を利用している。一般に例外処理機構は、例外が発生しない時の実行速度を劣化させないように実装されていることが期待できる。したがって、スレッド状態の保存は移送が起こる時のみ行われ、通常の移送が起こらないときの実行速度にはほとんど影響を及ぼさない。本研究でも、性能面からこの手法を採用することにした。図 2 にスレッド状態の保存のためのコード変換例を疑似コードで示す。左が変換前、右が変換後のコードである。

スレッド状態の保存のためのコード変換が必要になるのは、移送命令 (例、go) の地点と副作用として移送が起こる可能性のあるメソッド (**移送メソッド**) (例 may_go) 呼び出し地点である。以

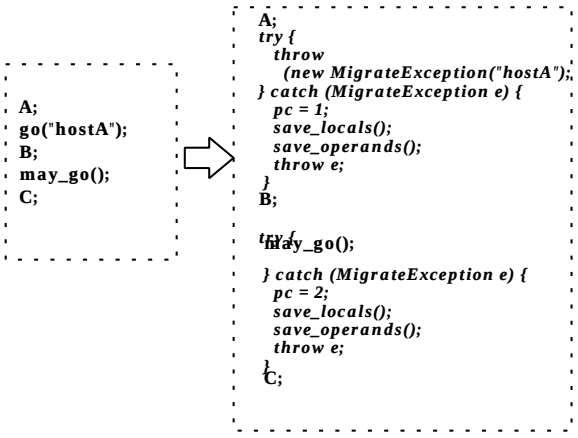

図 2: スレッド状態の保存のための変換

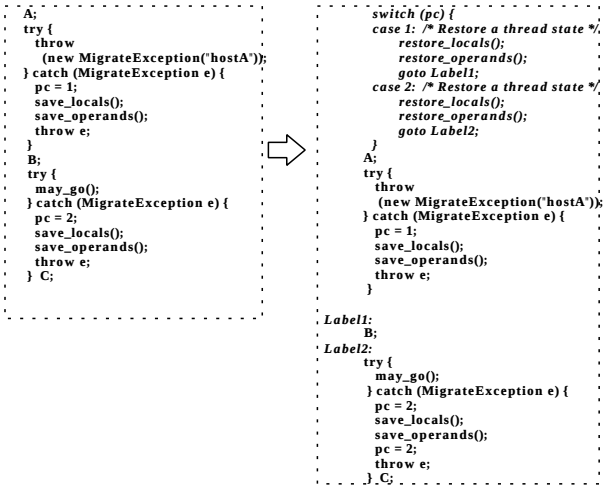

図 3: スレッド状態の復元のための変換

下では、この 2 点を**移送ポイント**と呼ぶことにする。

変換後には、移送命令が、**移送例外 (MigrateException)** を発生させる命令に置き換えられる。そして、移送ポイントを try-catch 構文で囲むことにより、移送例外を捉える。そして、移送例外の例外ハンドラで、save_locals() や save_operands() のようにそれぞれの移送ポイントでのフレームの状態（ローカル変数とオペランドスタック）と PC 変数が保存される。そして、保存が終了したら、再び移送例外をスローするという処理を再帰させることでスタックの各フレームの状態がスタックのトップから順に保存され、スタック全体の状態が保存される。

3.1.3 スレッド状態の復元

スレッド状態の復元は、各フレームごとに保存された PC 変数の値をもとに行われる。つまり、PC 変数の値によって、対応するフレームの状態を復元する (例では、restore_locals() や restore_operands())。その後、移送後に実行を再開するべきプログラムポイントにジャンプする。図 3 に変換例を示す。

復元後の実行再開ポイントは、移送命令では移送命令直後であり、移送メソッドではメソッド呼び出しの直前である。移送命令の地点に到達することはスタック全体の復元が完了したことを意味するので、移送命令の次の命令から再開する。一方、移送メソッドの場合は、復元されていないフレームを作るためにメソッド呼びだし直前から実行を再開する。

また、上の変換例で示した変換に加え、変換されるメソッドの先頭にはメソッド呼び出しがスレッド状態の復元によるものなのか、通常の呼び出しなのかを判断する if 文が挿入される。

3.1.4 Jamith における変換の特徴

Jamith は、バイトコードの高い表現力ミングの表現力の高さを利用してソースコード変換よりも効率的なコードへの変換を実現する。それは、次の 2 つのバイトコードの性質によって実現している。

- **非構造的な制御フローの表現**
  疑似コードの変換例では、goto によって任意の場所へのジャンプを記述している。しかし、Java には goto 文がない。したがって、

ソースコード変換では、switch 文によるトップレベルのジャンプを駆使して goto と同等の機能を実現しなくてはならない [Sek99]。しかし、これによって、複合文の展開や変数宣言文の移動などによりコード量が肥大化したり、変換後のコードの実行性能の劣化が問題となる。バイトコードには、goto があるのでこのような問題がない。

- **挿入コードの少量化**
  疑似コードからわかるように、移送ポイントごとにスレッド状態の保存や復元のためのコードが挿入が求められる。しかし、複数のこれらのコード間には、共有できるコードが多くある。Jamith は、例外ハンドラを複数の try-catch 構文で共有できること、共通の作業をサブルーチン化できること、といったバイトコードの性質を利用することで複数の共通コードをまとめて一つにすることにより、大幅にコード量を削減している。ソースコード変換においては、このような記述はできない。このコードの少量化は、メソッド中に多くの移送ポイントがある場合には効果がある。

3.2 Jamith における解析

Jamith は、クラスファイルに埋め込まれている豊富な情報を利用してコード変換に必要な解析を行う。それは、移送ポイントの検出と移送時のスタックの型解析である。

- 移送ポイントの検出
  コード変換を行うには、変換するべき移送ポイントを見つけ出さなければいけない。移送ポイントとは、移送命令と移送メソッドの呼び出しであった。移送命令は、バイトコードストリームを検査することで見つけられる。移送メソッドに関しては、コールグラフを作成して、移送命令を含むノードから逆に根に向かって通るパス上のノードがすべて移送メソッドである。こうすることで必要最小限の移送ポイントを見つけることができる。また、この解析には、Class Hierarchy Analysis[Dea95] とコールグラフを形成するすべてのノード（メソッド）の検査が必要となる。

- 移送時のスタックの型解析
  移送時にスタック（フレーム）の状態を保存するコードを挿入するには、各フレームで利用されるのローカル変数とオペランドスタックの型を知らなくてはならない。Jamith では、Java のベリファイアが行うのと同様に、バイトコード命令ごとのスタックの型付け規則 [Lin97] に従って移送時、つまり移送ポイントにおけるスタックの型を求める。この解析を容易にしたのは、型付きバイトコードであるという Java 特有の性質である。

3.3 位置依存のデータの扱い

移送するスレッドの状態の中には、あるホストでのみ意味をもつ位置依存のデータやプログラマの意志により移送したくないデータがある。つまり、スタックから到達するすべてのオブジェクトを移送することが理想であるとは言えない。したがって、Jamith は、他の多くのモバイルシステム同様にプログラマにある程度の migration-aware なプログラミングを可能にする API を提供する。この API を利用して、例えば、IO 関連のオブジェクトに関しては、移送前に保存されないようにそのオブジェクトへの参照を無効にし、移送後に再びオブジェクトを生成するなどの処理をユーザが明示的に行う。スレッド移送システムでは、生きているオブジェクトのすべてが移送対象となりうるので、プログラマは常にアプリケーション中の移送対象としないオブジェクトを意識してプログラミングする必要がある。

また、Jamith ではオペランドスタックも移送対象であるために、オペランドスタックに位置依存のデータがある場合にはシステム側で対処しなければならない。現在 Jamith は、ライブラリ中の位置依存のオブジェクトに関しては、ユビキタスなオブジェクトであると判断してシステム側で自動的に移送先でバインディングを行うようなコード変換を行っている。

将来的には、位置依存のデータは、移送先でバインディングすることや遠隔参照させるなどの指定をユーザが記述できるような API を実行時システムと連係して提供することを考えている。

3.4 Jamith での実行時情報の利用

バイトコード上の変換は、実行時 (ロード時) に変換することができる。これによって、ロードされるクラスだけを変換すること、コードを移送先で変換すること、さらに実行時情報を利用してコード変換することが可能になる。

Jamith では、この性質を利用して、ホストごとに異なったコード変換を行うことを想定する。例えば、エージェントが初めて生成されるホストならば、スレッド状態の復元コードは必要ない。したがってコード変換は、スレッド状態の保存に関するコードのみでよく、復元コードによって付加されるオーバヘッドをなくすことが可能である。同様に、移送先のコード変換時にも、状態の保存時に保存された PC 変数の値をもとに、変換するべきポイントを限定できる。例えば、もう実行されない移送ポイントのコード変換は不要である。

さらに、ネットワークの状況に応じて様々なコード移送メカニズムの選択を可能にする (図 4)。Jamith におけるコード変換では、移送ポイントの数や保存するデータの数に応じた量のコードが付加される。この付加されたコード量は、ネットワークの状況によってはボトルネックになる可能性がある。その対策として、コード変換を移送先でも行えば、コード量の小さい変換前のコード（normal code) を移送コードとして用いることが可能である。一方、十分にネットワーク容量が確保できる環境ならあらかじめ変換されたコード (transformed code) を送れることも出来る。また、Jamith では、コードの解析と変換を完全に切り分けて設計してあるために解析情報を再利用して移送先でコード変換だけを行うことも可能である。このような移送メカニズムの切り替えが自動的あるいは、ユーザの要求に応じて行えるシステム構築をすることでシステム全体の性能を高めることが可能であると考える。

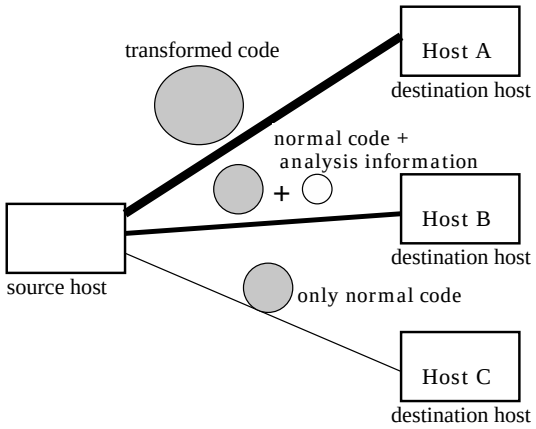

図 4: ネットワーク状況に応じた移送メカニズム

4 応用例：Jamith のモバイルエージェントシステムへの組み込み

Jamith は、スレッド移送を提供するプラグインだと考えることもできる。つまり、Jamith は、スレッド移送をサポートしていない既存のモバイルエージェントシステムに、簡単に組み込まむことが可能であり、そのシステムにスレッド移送機能を提供することを可能にする。Jamith は、実行時システムに依存しないよう設計されている。

つまり、Jamith が利用する実行時システムを既存のモバイルエージェントシステムとすることも可能である。

既存のシステム上で動くエージェント（コード）を Jamith によってコード変換させ、スレッド移送機能を持ったエージェントに書き換えるのである。したがって、既存のエージェントシステムには全く変更が必要ない。このメカニズムのもとに、AgentSpace[佐藤 98] においてスレッド移送機能をもった AgentSpace を実現した。この利点は、エージェントシステムの持つエージェント情報管理やエージェント間通信などの様々な機能を受け継ぎつつスレッド移送機能を加えることが出来ることである。

対象となるエージェントシステムは、移送命令 (aglets では dispatch, voyager では moveTo など）と移送後命令 (aglets では onArrival など) をもつシステムである。エージェント技術の標準化団体 FIPA[FIP] の mobility に関する仕様 (図 5) でいえば移送命令が 1. request であり、移送後命令が 4. execute にあたるため、多くのエージェントシステムが対象となることが期待できる。この二つの命令に Jamith を用いてフックを

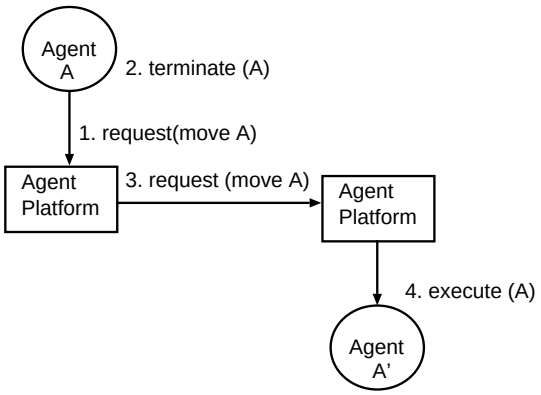

図 5: FIPA – Simple Mobility Protocol

かけて、移送命令の前にスレッド保存、移送後命令の直後にスレッド復元を行うようにコード変換する。コード変換は、静的に行うことも可能であるし、Jamith をモバイルエージェント化して、スレッド移送機能を持たせたいエージェントのいるホストに Jamith を移動させて、変換させることも可能であろう。

Jamith は、ユーザから任意の移送命令と移送後命令を指定できるように設計されているので、多くの既存のモバイルエージェントシステムから容易に利用できる。

5 性能評価

Jamith によって変換されたコードの性能を評価する。評価項目は、変換後の実行速度とコード量である。比較のために、変換前のコードとソースコード変換によるシステム JavaGO[Sek99] を用いる。JavaGO は、移送が失敗したときの処理のためのコード変換も行うが、ここでは公平な比較のためにその処理を削除している。 実験環境は、Pentium(R) II 450MHz、Memory 128MB, WindowsNT、JDK1.2、HotSpot1.0 である。

ベンチマークは、クイックソート (QSort)、fibonacci(Fib)、竹内関数 (Taku)、Matrix-vector Multiply(Matrix) を用いた。それぞれ移送ポイントは、3ヶ所、3ヶ所、5ヶ所、1ヶ所ある。Matrix は、2 重ループの内側に移送ポイントがある例であり、それ以外の 3 つは、再帰呼び出しを用いた例である。

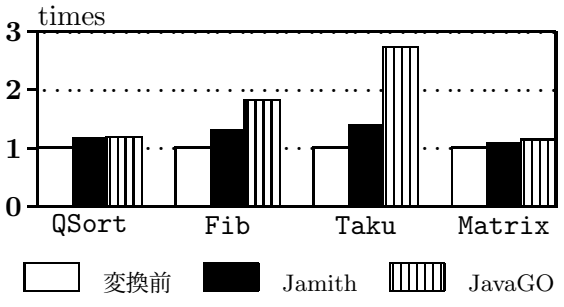

図 6: 実行速度の比較

実行性能を示す図 6 のグラフは、変換前のコードの実行時間を 1 とした場合の Jamith、JavaGO によって変換されたコードの実行時間の増加率である。Jamith によって変換されるコードが、変換前のコードに付加するオーバーヘッドは、(1) 移送ポイントの変換におけるオーバーヘッド、(2) 変換されるメソッドの先頭にある if 文 (復元時かどうかのチェック) のオーバヘッド、(3) 変換されたメソッドの呼び出しにかかるオーバヘッド、である。(1) は、例外処理機構のためにほとんどオーバヘッドがない。それは、Matrix がほとんどオーバヘッドなしで実行されている点でわかる。(2) と (3) は、メソッドの呼び出し回数に比例して大きくなる。それを示すように、QSort、Fib、Taku とメソッド呼び出しの回数が多くなるベンチマークほどオーバヘッドが大きくなっている。

JavaGO と比較した場合、3.1.4 節で述べたソースコード変換の欠点が現れるベンチマークでは、Jamith は、JavaGO よりもよい性能が得られている。

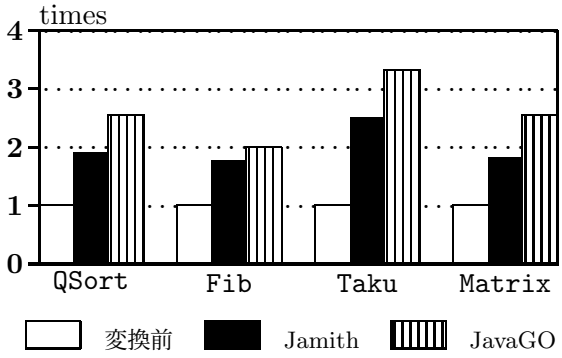

図 7: コード量の比較

コード量の比較を示す図 7 のグラフも変換前のコード量 (クラスファイル) を 1 とした場合のそれぞれの増加率である。Jamith においてコード量は、移送するスレッドの状態量と移送ポイントの数に応じて増加する。ここでのベンチマークでは、変換前と比べ平均 2 倍増加している。JavaGO が平均 2.6 倍であるのに比べると、大幅にコード量を減らすことができた。これは、Jamith が行ったコード量の少量化の効果とともに、3.1.4 節で述べたソースコード変換の場合に必要になる複合文の展開が要因である。

6 関連研究

[Fün98, Sek99] などのスレッド移送のためのソースコード変換技術をもとに、ほとんど通常の実行性能を損なわないコード変換技術が提案されている [阿部 99, 多賀 99]。その手法とは、スレッド状態の復元のためのコードは移送後に一度だけ実行されることに注目して、スレッド状態の保存と復元のためコードの両方を挿入したメソッドの他に、スレッド状態の保存のためのコードだけを挿入したメソッドを用意し、通常はそちらのメソッドが実行されるようにする。スレッド移送後のスレッド状態の復元時だけは、復元のコードを含んだメソッドを呼ぶようにする。こうすることで、通常はスレッド状態の保存のために挿入されるコードによるオーバヘッド、つまりほぼ例外処理が挿入されることによるオーバヘッドだけに抑えられるために、実行効率を損なうことなくスレッド移送を実現できる。しかし、この手法の欠点はメソッドを二つ用意することによってコード量の増大し、通信速度を劣化させてしまうことである。

Jamith では、上記の手法を欠点を補いつつ取り入れることが可能である。それは、Jamith では移送先でコード変換することができるために、通信速度の劣化を引き起こすことなく、実行効率を向上させられる利点だけを得ることができるからである。

7 まとめ

本稿では、スレッド移送を実現するためのバイトコード上の変換技術を提案した。さらに、既存のスレッド移送のアプローチと比較してバイト

コード上の変換の利点を述べ、それを有効利用した高速化や既存のモバイルエージェントシステムへの組み込み方法を紹介した。変換されたコードの質については、ソースコード変換の手法と幾つかのベンチマークで比較し、性能向上とコード量の削減が実現できることを確認した。

今後の課題として、マルチスレッド環境での利用と JIT による性能の違いへの対応について述べる。

マルチスレッド環境で Jamith を有効に利用するためには、複数スレッドの同時移送と同期制御情報の移送が必要になる。前者に関しては、移送命令を呼ぶ以外のスレッドの実行状態を保存するために、Checkpointing 技法のようなコード変換が必要になる。これは、あらゆるプログラム地点が移送ポイントになりうることを意味し、最悪の場合はほとんどのメソッドにたいしてコード変換が必要になる。しかし、このような状況においてこそ、Jamith が実現した実行速度の劣化とコード量の増加を最小限で抑えた利点が発揮されると考える。また、後者に関しては、スレッドの制御情報を管理する実行時システムを用意することと共にその実行時システムにスレッドの制御情報を受け渡すようなコード変換を行うことによって対処することを考えている。移送時には、実行時システムが管理する同期制御情報も同時に移動させることで同期制御の移送を実現する。

もうひとつの課題とした JIT による性能の違いによる問題とは、Jamith によるコード変換によるオーバヘッドが JIT などの実行形態によって大きく異なることである。ここに、竹内関数による HotSpot 1.0、Symantec 3.10.107、IBM JIT3.5、Microsoft SDK3.1J という JIT とインタプリタ実行によるオーバヘッドの割合の違いを図 8 のグラフに表す。グラフは、それぞれの JIT やインタプリタ実行での実行時間の変換前のコードに比べての増加率を示す。

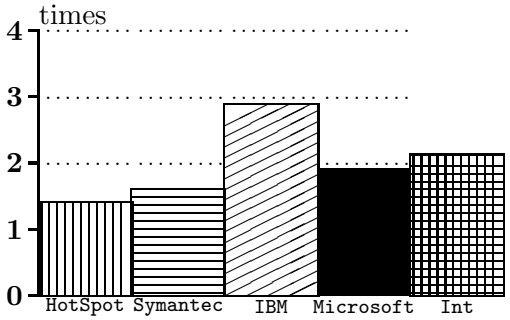

図 8: JIT による性能の違い

グラフを見てわかるように JIT によって性能が大きく異なっている。Jamith が挿入するコードが、それぞれの JIT でどのようにコンパイルされるかの詳細はわからないが、例えば、IBM JIT やインタプリタ実行においては例外処理を挿入することによって著しく性能を劣化させることがわかった。モバイルエージェントは、様々なホストを利用するが、あるホストが利用する JIT によってアプリケーション全体の性能を著しく損なうのは望ましくない。本研究の調査で、IBM JIT やインタプリタ実行において例外処理を用いるコード変換ではなく、[Hoh98] で行われているような下位のフレームを保存するために return 文を用いるコード変換手法の方がよい性能を出すことがわかった。つまり、利用する JIT によってコード変換手法を切り替えることによってより実行性能を向上させることが可能である。Jamith では、実行時情報を利用した移送先でのコード変換が出来るので、利用する JIT の種類という実行時の情報を利用してコード変換手法を切り替えることが可能であると考える。

参考文献

[Che97] Chess, D., C. Harrison, and A. Kershenbaum: Mobile agents: Are they a good idea?, in *Mobile Object Systems: Towards the Programmable Internet*, Vol. 1222 of *LNCS*, pp. 25–45, 1997.

[Cug97] Cugola, G., C. Ghezzi, G. P. Picco, and G. Vigna: Analyzing Mobile Code Languages, in *Mobile Objects Systems: Towards the Programmable Internet*, Vol. 1222 of *LNCS*, pp. 94–109, 1997.

[Dea95] Dean, J., D. Grove, and C. Chambers: Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis, in *ECOOP'95—Object-Oriented Programming, 9th European Conference*, Vol. 952 of *LNCS*, pp. 77–101, 1995.

[FIP] FIPA, : FIPA WWW pages: available at http://www.fipa.org.

[Fün98] Fünfrocken, S.: Transparent Migration of Java-Based Mobile Agents, in *Proceedings of the Second International Workshop on Mobile Agents '98*, Vol. 1477 of *LNCS*, pp. 26–37, 1998.

[Gla98] Glass, G.: ObjectSpace Voyager — The Agent ORB for Java, in *Proceedings of the Second International Conference on Worldwide Computing and Its Applications' 98*, Vol. 1368 of *LNCS*, pp. 38–55, 1998.

[Gra98] Gray, R. S., D. Kotz, G. Cybenko, and D. Rus: D'Agents: Security in a multiple-language mobile-agent system, in *Mobile Agents and Security*, Vol. 1419 of *LNCS*, pp. 154–187, 1998.

[Hoh98] Hohlfeld, M. and B. Yee: How to Migrate Agents, University of California, SAN DIEGO, Computer Science and Engineering, 1998.

[Hoh99] Hohl, F.: Mobile Agents List, 1999, available at http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole/mal/mal.html.

[Lan98] Lange, D. B. and M. Oshima: *Programming and Deploying Java Mobile Agents with Aglets*, Addison-Wesley, 1998.

[Lin97] Lindholm, T. and F. Yellin: *The Java Virtual Machine Specification*, Addison-Wesley, 1997.

[Ohs97] Ohsuga, A., Y. Nagai, Y. Irie, M. Hattori, and S. Honiden: PLANGENT: An Approach to Making Mobile Agents Intelligent, *IEEE Internet Computing*, Vol. 1, No. 4, pp. 50–57, 1997.

[Pei97] Peine, H. and T. Stolpmann: The Architecture of the Ara Platform for Mobile Agents, in *Proceedings of the First International Workshop on Mobile Agents '97*, Vol. 1219 of *LNCS*, pp. 50–61, 1997.

[Sek99] Sekiguchi, T., H. Masuhara, and A. Yonezawa: A simple extension of java language for controllable transparent migration and its portable implementation, *Coordination '99*, 1999.

[Shu] Shudo, K.: MOBA—Mobile agent facilities for Java language environment: available at http://www.shudo.net/moba.

[Sun] Sun Microsystems, : *Java HotSpot Perfomance Engine*: available at http://java.sun.com.

[阿部 99] 阿部 洋丈, 一杉 裕志, 加藤和彦：ソースコード変換技術を用いた Java 言語におけるスレッドのモビリティの実現法, 並列/分散/協調処理に関するサマー・ワークショップ (SWOPP '99) 会議録, PRO-2 (3), 1999.

[佐藤 98] 佐藤 一郎：AgentSpace: モバイルエージェントシステム, 1998, available at http://www.is.ocha.ac.jp/~ichiro.

[多賀 99] 多賀 奈由太, 関口 龍郎, 田浦 健次郎, 米澤 明憲：通常の実行効率をほとんど損なわないスレッドマイグレー ションが可能な C++, 並列/分散/協調処理に関するサマー・ワークショップ (SWOPP '99) 会議録, PRO-2 (4), 1999.