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

阿部洋丈[†] 一杉裕志[††] 加藤和彦[†††]

一般にオブジェクトは，メソッドコード，属性値，計算の状態（いわゆるスレッド）の3つの要素より構成されている．モバイルエージェントシステムのプラットフォームとしてJava言語システムを利用する場合，クラスの動的ロード機能とオブジェクト・シリアリゼーション機能によってメソッドコードと属性値のコンピュータサイト間移動は容易に達成されるが，スレッドを移動することは容易には行えない．そのため，これまでに提案されたJava上のモバイルエージェントシステムの多くは，メソッドコードと属性値の移動のみをサポートし，スレッドの移動はサポートしていない．本論文では，Java言語で記述されたソースコードに対するソースコード変換技術を用いて，スレッドのモビリティ機能を有するモバイルエージェントシステムの実現法を提案する．この方法では，ソースコード上で標準的に利用可能な情報のみを用いて，スレッドを抽出・復元する．このような変換を行うと一般に実行時オーバヘッドを生じるが，提案変換技術は，そのオーバヘッドを極力低減化する最適化技術を含んでいる．ソースコード変換器作成プラットフォームであるEPPを使った実現を述べると共に，提案方式の有効性を検証するために行った実験結果を示す．

An Implementation Scheme of Mobile Threads with a Source Code Translation Technique in Java

HIROTAKE ABE,[†] YUUJI ICHISUGI[††] and KAZUHIKO KATO[†††]

In general, an object consists of program code, attributes, and calculation states (so-called threads). When we use Java as a platform of mobile agent system, the transportation of program code and attributes is easily achieved by using the dynamic class loading mechanism and object serialization that are provided as Java's standard functionalities. However, it is not easy to transport threads from a host to another, because a standard way to access the contents of threads is not provided. Thus most Java-based mobile agent systems only support the mobility of program code and attributes. In this paper we propose a scheme to implement mobile agent system that allows to transport threads using a source code translation technique. The emitted code are standard Java code, so the scheme is portable. The scheme also includes several optimization techniques to reduce the overheads incurred by extra instructions inserted by the translation. We also describe an implemented system of the proposed scheme. The system is implemented with EPP, a platform for implementing the source code translator. We present an experimental result using the system to verify the effectiveness of the presented scheme.

1. はじめに

インターネットに代表される広域ネットワーク環境や，携帯端末等を用いたモバイルネットワーク環境が一般的となった現在，高度なネットワークアプリケーションを構築する基本技術としてモバイルエージェント技術が注目を集めており，世界各地で活発な研究開発が進められている[6),15),20)]．モバイルエージェントを実現するための基本アプローチとしては，全く新たなプログラミング言語システムとして設計したアプローチ[2),16)]，ネットワーク環境上での利用を前提に設計されたJava言語システムを利用したアプローチ[8),17)]，そして複数のプログラミング言語システムに対応可能なように設計したアプローチ[5),7)]が知られている[19)]．この中でJava言語システムを利用したアプローチは，Java言語がマルチプラットフォーム環境上での利用を前提に設計されていることに加え，ク

† 筑波大学大学院博士課程工学研究科
Doctoral Program in Engineering, University of Tsukuba
†† 電子技術総合研究所
Electrotechnical Laboratory
††† 筑波大学電子・情報工学系
Institute of Information Sciences and Electronics, University of Tsukuba

ラスの動的ロード機能，オブジェクト・シリアリゼーション，および遠隔メソッド起動等の，モバイルエージェントを実現するための基本機能を標準的に提供しているため，最も多くの研究プロジェクトが採用しているアプローチである．

Java言語システムにおいてスレッドの移動をサポートすることが容易でない最大の理由は，計算の状態を移動させるためには，送り出し側が実行中のJava仮想計算機の状態を抽出してデータ化し，サイト間で転送し，受け取ったサイトが復元する必要があるが，このための標準的な基本機能が提供されていない点にある．一般に仮想機械は，プログラムカウンタや内部レジスタに加えて，計算状態を格納したスタックデータを含み，その状態抽出は複雑である．しかもJava言語システムの場合，オペレーティングシステムやCPUアーキテクチャ等の実行時プラットフォームとは独立に設計するという設計ポリシーを有するため，この設計ポリシーを保ちつつ，かつ，一般的に使われているjust-in-timeコンパイラ等の性能向上技術との共存を図りながら，Java仮想計算機の状態を一般的に抽出・復元する機能を提供することは容易ではない．

本論文では，Java言語で記述されたソースコードに対するソースコード変換技術を用いて，スレッドのモビリティ機能を有するモバイルエージェントシステムの実現法を提案する．この方法では，ソースコード上で標準的に利用可能な情報のみを用いて，スレッドを抽出・復元する．ごく最近，本研究と同様に，ソースコード変換技術を用いてJava言語システム上にモバイルエージェントを実現する方法が提案されている[4),14)]．本研究で提案する方法はそれらと比べて，以下のような特徴を有している．

- モバイルエージェントを実現するためのソースコード変換を施すと，モバイルエージェントの移動が起きる際にオーバヘッドを要するのみならず，プログラム全般に実行時オーバヘッドが増加する．提案方式は，このオーバヘッドを大幅に軽減するための最適化機構を含む．この機構は，実行中に移動が発生する可能性のあるメソッドごとに，オーバヘッド増加の原因であるスレッドの状態復元を伴うコードと，状態復元を伴わないコードの2種類をソースコード変換により生成し，前者のコードの呼び出し回数を最低限とするようなコード生成を自動的に行うことにより，オーバヘッドを軽減する．
- ソースコード変換技術は，オブジェクトおよびスレッドのモビリティを実現するための有用な技術であるのみならず，他の言語上の機能拡張を実現する上でも有用な技術である．例えば，パラメータ化されたクラス等の実現にソースコード変換を用いるのは常套的である[10),12)]．複数のソースコード変換技術を，ユーザが組み合わせて利用することを可能にするため，本論文で述べる実装では，複数種類のソースコード変換を可能にするプリプロセッサ構築フレームワークであるEPP[18)]を用いた．
- 提案変換方式は標準的なJavaコードを出力し，Java仮想機械に対する拡張は全く要さない．また実装に用いたEPP自体も標準的なJava仮想機械上で動作する．このため，本論文で提案する実現方式および実装はポータビリティが高いものとなっている．

以下，本論文は次のように構成されている．2章では，モバイルエージェントのプログラミング法を論じ，提案方式の位置づけを行う．3章では，スレッドのモビリティを実現するソースコード変換法を提案する．4章では提案した変換法の実現について述べ，5章で実験結果を示す．最後に6章においてまとめと今後の課題について述べる．

2. モバイルエージェントのプログラミング

本章では，モバイルエージェントの方式分類と，それがプログラミングに与える影響について論じる．

モバイルエージェントの移動の形態は，強いマイグレーションと弱いマイグレーションの2つに分類される[1)]．強いマイグレーションは，エージェントの移動時にプログラムコード，属性値，スレッドの状態の3つを移動先のホストに転送する．それに対し，弱いマイグレーションでは，移動時に転送されるのはプログラムコードと属性値のみで，スレッドの状態は転送されない．

Javaでは，プログラムコードを転送する方法，および属性値を転送する方法は標準的に提供されているが，スレッドの状態を転送する方法は標準的には提供されていない．そのために，Aglets[9)]やVoyager[11)]などのJava上で実装されたモバイルエージェントシステムの多くは弱いマイグレーションを採用している．弱いマイグレーションを用いてモバイルエージェントの移動を表現するには，スレッドの実行状態をプログラマが自分の責任において保存・復元を行う必要がある．そのためには，モバイルエージェント本来の仕事の他に，自分のスレッドの状態を保存・復元するためのプログラム記述を行う必要がある．

```
public void run() {
  migrate(hostA);
  job(hostA);
  migrate(hostB);
  job(hostB);
  migrate(hostC);
  job(hostC);
}
```

図1 強いマイグレーションの例1
Fig. 1 Example of strong migration (1).

```
int state = 0;
public void run() {
  switch (state) {
  case 0:
    state = 1;
    migrateWeakly(hostA);
  case 1:
    job(hostA);
    state = 2;
    migrateWeakly(hostB);
  case 2:
    job(hostB);
    state = 3;
    migrateWeakly(hostC);
  case 3:
    job(hostC);
  }
}
```

図2 弱いマイグレーションの例1
Fig. 2 Example of weak migration (1).

```
public void run() {
  for (int i = 0; i < hosts.length; i++) {
    for(int j = 0;j < hosts[i].length;j++){
      migrate(hosts[i][j]);
      job(hosts[i][j]);
    }
  }
}
```

図3 強いマイグレーションの例2
Fig. 3 Example of strong migration (2).

```
int i = 0, j = 0;
boolean running = false;
public void run() {
  if (! running) {
    running = true;
    migrateWeakly(hosts[i][j]);
  } else {
    job(hosts[i][j]);
    if (++j < hosts[i].length) {
      migrateWeakly(hosts[i][j]);
    } else {
      j = 0;
      if (++i < hosts.length) {
        migrateWeakly(hosts[i][j]);
      }
    }
  }
}
```

図4 弱いマイグレーションの例2
Fig. 4 Example of weak migration (2).

例として，決められたホストを巡回して，それぞれのホスト上で仕事を行うエージェントを，強いマイグレーションおよび弱いマイグレーションを用いてプログラムする場合を取り上げる．強いマイグレーションを用いた場合，**図1**に示すように自然に記述することができる．一方，弱いマイグレーションを用いて記述するには，**図2**に示すように，スレッドの実行状態を保存するための属性値を新たにエージェントに追加する必要がある．

この例では，巡回すべきホストが予め分かっているので，繰り返しなどの制御構造を使う必要が無く，弱いマイグレーションでも比較的簡単に記述することが可能である．次に，これよりも複雑な例として，巡回すべきホストが二次元配列に格納されている場合を考える．強いマイグレーションによる記述例が**図3**，弱いマイグレーションによる記述例が**図4**である．今度の例では，強いマイグレーションでの記述例に比べて，弱い記述例のプログラムの方がより一層複雑になっている．その原因は，プログラマが管理すべきスレッドの状態が増えていることと，複数の文が複合しているために，実行を再開すべき文に到達することが難しいということが挙げられる．この程度の複雑さになると，手動でプログラムを変換することは困難になる．また，プログラムの可読性も低下し，プログラム中に誤りが含まれる可能性も高くなってしまう．

このように，モバイルエージェント・システムが強いマイグレーションをサポートするか，弱いそれをサポートするかはモバイルエージェントのプログラミングに大きな影響を与える．プログラムの可読性や保守性の向上のためには，強いマイグレーションのサポートが望ましいと筆者らは考えている．以下3章以降では，Java言語上で強いマイグレーションを効率的にサポートする方式とその実現，および実現を用いた実験結果について述べる．

3. 変換方式

本章では，Java言語上のソースコード変換技術を用いて，強いマイグレーションをサポートする方式を提案する．ごく最近提案された類似の変換方式[4),14)]と比べて，実行時オーバヘッドを極力減らす工夫が凝

らされている点が大きな特徴となっている．

3.1 実行状態の保存と復帰

スレッドの移動を行うためには，実行状態（スタックの状態，メソッド内のローカル変数の値，プログラムカウンタの位置）を取り出し，移動先のホストに送った後，移動先でそれを復元させて実行を再開する処理が必要となる．

Java 言語ではスレッドの実行状態を取り出したり変更する手段は提供されていない．そこで，移動させたいプログラムのソースコードを変換して，メソッドの実行状態を保存するコードと復元するコードを埋め込むことにより，スレッドの移動を実現可能にする．実行状態の保存には文献 4)，14) と同様に例外処理を利用することで，普段の実行のオーバヘッドを小さく抑えることができる．まずこの方法について説明する．

スレッドが他のホストに移動しようとして `migrate(host)` というメソッドを呼び出すと，このメソッドは例外 SaveContext を発生させる．すると，まず一番上☆のスタックフレーム上のメソッドがその例外を受け取り，例外ハンドラの中でローカル変数の値などの自分の実行状態をエージェントを表すオブジェクト内に保存する．その後，再び例外 SaveContext を発生させる．この例外は，一段下のスタックフレームを実行しているメソッドで受け取られ，そこの状態も保存される．同様にして，スタックの上から下に向かって，すべてのフレームの状態が保存される．

メソッド中でのプログラムカウンタの位置は，具体的には以下のように保存される．SaveContext という例外はメソッド呼び出し式の実行中でしか発生し得ない．そこで，メソッドのプログラム中の何番目のメソッド呼び出し式を実行中かを表す整数を「メソッド内のプログラムカウンタ」とする．例えば，図 5 の `m2()` を実行中に例外が発生したなら，`foo` 内のプログラムカウンタは 2 とする．

図 5 のメソッドは図 6 のように，全てのメソッド呼び出し式を try-catch で囲むように変換することによって，実行状態を保存可能にできる．このメソッドは，他のメソッドの呼び出し中に例外 SaveContext が発生したら，catch 節で，その時のプログラムカウンタの値を外部に保存した後，例外 SaveContext を再び発生させる．なお，一般的な言語処理系では例外処理構文は，例外が起きない限りはオーバヘッドがほとんどない実装になっているため，この変換による実行速度の低下は小さいと期待できる☆☆．

```
void foo(){
  m1();
  m2();
  m3();
}
```

図 5 変換前のメソッド
Fig. 5 Method before translation.

```
void foo() throws SaveContext {
  try {
     m1();
  } catch(SaveContext e) {
    /* save context, pc = 1 */
    ...
    throw e;
  }
  try {
     m2();
  } catch(SaveContext e) {
    /* save context, pc = 2 */
    ...
    throw e;
  }
  try {
     m3();
  } catch(SaveContext e) {
    /* save context, pc = 3 */
    ...
    throw e;
  }
}
```

図 6 実行状態の保存
Fig. 6 Saving the execution states.

```
void foo() throws SaveContext {
  int pc;
  if (<状態復帰中>){
    pc = <保存されている pc>;
  } else {
    pc = 0;
  }
  switch (pc){
  case 0:
  case 1: try{ m1(); }catch ...;
  case 2: try{ m2(); }catch ...;
  case 3: try{ m3(); }catch ...;
  }
}
```

図 7 実行状態の復帰
Fig. 7 Restoring the execution states.

図 6 のメソッドの途中から実行を再開可能にするためには，メソッドをさらに図 7 のように変換すれば

☆ スタックは下から上に向かってのびるものとする．

☆☆ ただし，最適化コンパイラによる最適化を妨げることによって，例外処理構文がない場合より実行速度が低下する場合がある．

良い．このメソッドは呼び出されるとまず，実行状態を復帰中なのか，通常の実行なのかを外部にあるフラグを参照して判断する．復帰中でないなら pc の値を 0 にし，メソッドの先頭から普通に実行する．復帰中の場合は，pc の値を，実行が中断された時の値に設定し，その後の switch 文によってジャンプする．例えば，メソッド m2() の呼び出し中に実行が中断されていた場合は，pc の値は 2 であり，ジャンプ後には再び m2() が呼び出される．呼び出された m2() もまた自分自身の実行状態を復帰し，自分が呼び出していたメソッドを再び呼び出す．同様にして，スタックの下から上まで次々に実行状態が復帰され，最終的にメソッド migrate が呼び出されたところで，「状態復帰中」のフラグはクリアされ，実行が再開される．

図5は，ローカル変数のないメソッドだが，ローカル変数がある場合は，例外 SaveContext を受け取る catch 節中で各ローカル変数の値を保存し，状態復帰時にはメソッドの先頭で値をローカル変数に戻すように変換すれば良い．

pc やローカル変数の値は，現在のスタックフレームを表すオブジェクト内にコピーされ，そのオブジェクトをグローバルな場所にあるスタック型のデータ構造中に保存し，呼び出しスタックの状態を保存する．スタックフレームを表すオブジェクトのクラス定義は，ソースコード変換時にそれぞれのメソッドごとに自動的に生成される．

なお4章で述べる実装では変換処理が簡単になるように，変数名の衝突に対処しながらローカル変数の宣言をすべてメソッドの先頭に移動し，保存・復元の変換処理を行っている．また次のように，1つの式の中に複数のメソッド呼び出しがある場合は，式の評価途中の実行状態を保存できるようにする必要がある．

```
y = f(x) + g(x);
```

上記の様な場合は，以下の様に一時変数に値を代入する形にあらかじめ変換しておく．

```
int temp = f(x);
y = temp + g(x);
```

3.2 ネストした制御構造の状態の復帰

Java 言語では block の内側にジャンプする方法はないため，メソッド内のプログラムカウンタの位置を復帰するためには工夫が必要である☆．提案方式では，block の中の文を switch 文のトップレベルにコピーすることで分岐可能にする．**図8** のメソッドの実行状態

```
void foo(){
  while (e1){
    while (e2){
      m1();
      m2();
    }
    while (e3){
      m3();
      m4();
    }
  }
  m5();
}
```

図8 ネストした制御構造を持つメソッド
Fig. 8 Method with nested structure.

を復帰する場合を例として用いてこの変換法を説明する．このメソッドは**図9**のように変換される．図9のメソッドは複数の switch 文によって，ネストした block の一番深いところから次々に実行を再開していく．最も深く制御構造がネストしている，図8中の m1() の呼び出しで実行が中断されていた場合（pc の値が1）を考える．まず1番目の switch 文で深さ3の block での実行の再開，すなわち m1(); m2(); の呼び出しを実行する．次の switch 文では，深さ2の block の実行の再開，すなわち while (e2)... と while (e3)... のループを実行する．最後の switch 文では，メソッドのトップレベルの block の while (e1)... と m5(); を実行する．

この変換方法の大きな利点として，オリジナルのメソッドが持つループの内側の構造が，変換後もそのまま残っている点がある．このため，最適化コンパイラによるループの最適化をあまり妨げないと期待できる．しかし，この変換方法を用いると，変換後のプログラムは，変換前のプログラムの長さを m，プログラム中のブロックのネストの深さを n とすると，$O(mn)$ の長さになってしまう．その結果，キャッシュへの悪影響による性能低下を起こすことも考えられる．

この他にも，プログラムが長くならないような変換方法も考えられるが，次のように考えてこの方法を採用した．

- たとえプログラムが長くなっても，オーバヘッドが少ない方がモバイルエージェントには有利である．
- ブロックのネストが深くない場合は影響はあまり受けない．大きな影響を受けるくらいにネストが深くなることはあまりない．

文献14)でも，出力されるコードの形態は異なるが，ここで述べた方法と同等の方法が用いられている．

☆ C, C++ 言語ならば goto 文か switch 文で block の内側に飛び込めるため，復帰は簡単に行える．

```
void foo() throws SaveContext {
  // pc の値とローカル変数の値の復帰
  ...;
  // 深さ3の block の実行再開
  switch (pc){
  case 1: try {m1();} catch ...
  case 2: try {m2();} catch ...
     break;
  case 3: try {m3();} catch ...
  case 4: try {m4();} catch ...
     break;
  }
  // 深さ2の block の実行再開
  switch (pc){
  case 1:
  case 2: while (e2){
            try {m1();} catch ...
            try {m2();} catch ...
          }
  case 3:
  case 4: while (e3){
            try {m3();} catch ...
            try {m4();} catch ...
          }
          break;
  }
  // 深さ1（トップレベル）の block の実行再開
  switch (pc){
  case 0:
  case 1:
  case 2:
  case 3:
  case 4: while (e1){
            while (e2){
              try {m1();} catch ...
              try {m2();} catch ...
            }
            while (e3){
              try {m3();} catch ...
              try {m4();} catch ...
            }
          }
  case 5: try {m5();} catch ...
    break;
  }
}
```

図 9 ネストした制御構造の実行位置の復帰
Fig. 9 Translated code of nested structure.

3.3 大域脱出に対する対処法

前節で述べたような，ネストしたブロック内の文をコピーする方法を用いると，break や continue などの大域脱出のための文が正しく動作しなくなる場合がある．それは，文を展開したことにより，break や continue が，脱出すべきループの外側に出てしまうことが原因である．そのような場合は，break や continue 文を適切な位置へのジャンプするような文に置き換える．具体的には，pc の値を変更して switch から break する文に置き換える．たとえば，図 10 のようなプログラムは図 11 のように変換することで正しく動作させることができる．

```
void foo() {
L1:
  for (...) {
    for (...) {
      bar(); // pc == 1
      if (...) break L1;
    }
  }
}
```

図 10 ネストしたループからの脱出：変換前
Fig. 10 Breaking out of nested loop: before translation.

```
void restore_foo() {
  ...;

  LEVEL3:
  switch (pc) {
  case 1:
    if (...) {
      pc = 2; break LEVEL3;
    }
  }

  LEVEL2:
  switch (pc) {
  case 1:
    for (...) {
      bar();
      if (...) {
        pc = 2; break LEVEL3;
      }
    }
  }

  LEVEL1:
  switch (pc) {
  case 1:
  L1:
    for (...) {
      for (...) {
        bar();
        if (...)
          break L1;
      }
    }
  case 2:
  }
}
```

図 11 ネストしたループからの脱出：変換後
Fig. 11 Breaking out of nested loop: after translation.

3.4 メソッド呼び出しのオーバヘッド削減

前節までに説明した方法では，マイグレーションを起こさない通常の実行時であっても，メソッドの先頭

```
void restore_foo() throws SaveContext {
  // 一段上のフレームの復帰実行メソッドを呼ぶ
  switch (pc){
  case 1: try{restore_m1();} catch ...
    break;
  case 2: try{restore_m2();} catch ...
    break;
  case 3: try{restore_m3();} catch ...
    break;
  }
  // 一段上のメソッドの実行が終了したら,
  //自分自身の実行を再開する.
  switch (pc){
  case 1: try {m2();} catch ...
  case 2: try {m3();} catch ...
  case 3: break;
  }
}
```

図 12 復帰実行メソッド
Fig. 12 Restoring execution states.

で「状態を復帰すべきかどうか」の判断を下さなければならず，メソッド呼び出しのたびにオーバヘッドが生じる．この問題を解決するため，単に実行だけをするメソッドと，状態を復帰してから実行を再開するメソッドの2つのバージョンのメソッドを作り，これら2つを状況に応じて静的に呼び出し分ける方法を提案する．2種類のバージョンのプログラムを用意して実行時間を短縮するという手法は並列処理用言語の分野では今までにもいくつか提案されてきている[3),21)]．我々は，そのような手法を用いることでメソッド呼び出しのオーバヘッドを軽減することができることを発見し，その手法を取り入れることにした．

2つのバージョンのメソッドを，以下「実行メソッド」と「復帰実行メソッド」と呼ぶ．図5のメソッドを変換してできる実行メソッドは図6のようになり，復帰実行メソッドは**図12**のようになる．なお復帰実行メソッドは，もとのメソッド名に `restore_` を追加したメソッド名になる．

実行メソッドは，変換前のメソッドと同じ動作をするが，実行中にある例外（SaveContextとする）が発生した時は，実行を中断してそのメソッドの実行状態を外部に用意された領域に保存する．復帰実行メソッドは，保存された実行状態を復帰し，中断されていた場所から実行を再開するメソッドである．復帰実行メソッドは，状態の復帰を行った後は，実行メソッドと全く同じ動作をする．

具体的に状態の保存・復帰がどのように進むかを説明する．最初は，実行メソッドだけを使ってプログラムの実行が進む．スレッドが他のホストに移動しようとして `migrate(host)` というメソッドを呼び出すと，このメソッドは例外SaveContextを発生させ，これによって，スレッドの実行状態が保存される．スレッドの移動先では，まず，一番下のスタックフレームに対応する復帰実行メソッドが呼び出される．復帰実行メソッドは，自分の実行状態を復帰させた後，自分より一段上のスタックフレームに相当するメソッドの復帰実行メソッドを呼び出す．このようにしてスタックが一番上まで復帰すると最後には，メソッド `restore_migrate` が呼び出されるが，これは何もしないで実行を終える．`restore_migrate` を呼び出した一番上の復帰実行メソッドは，そこから中断されていた実行を再開する．この復帰実行メソッドの実行が終了すると，一段下の復帰実行メソッドが実行を再開するという具合に，再開された実行が進む．実行再開後は，実行メソッドの呼び出しのみが使われ，復帰実行メソッドが呼び出されることはない．

復帰実行メソッドは，実行メソッドに比べると，メソッドの先頭で復帰処理を行うため，呼び出し時に余分なオーバヘッドがある．しかし復帰実行メソッドの呼び出しは，スレッド移動が起きた直後にスタックの深さの回数だけ起きるにすぎない．従ってこのオーバヘッドは，スレッド移動が起きない時の普段の実行速度には全く影響を与えないと考えてよい．

実行時に2つのバージョンのメソッドのどちらが多く使われるかは，アプリケーションの性質に依存する．例えばメソッド内でのループが少なく，頻繁に他のメソッドを呼び出すアプリケーションでは，復帰実行メソッドよりも実行メソッドによる実行時間が占める割合が大きい．逆に，特定のメソッド内でのループが実行の大部分を占める数値計算処理の場合は，復帰実行メソッドが復帰後もループを実行し続けるため，復帰実行メソッドの実行が占める割合が大きくなる．しかし，本論文で提案する変換方法では，これまでの節で述べたように，どちらのバージョンも変換前のコードに比べて実行効率の低下ができるだけ小さくなるように考慮されている．従って，再帰呼び出しが多いアプリケーションでもループが多いアプリケーションでも小さいオーバヘッドで実行できる．

4. 実　　装

提案方式の有効性を検証するために，提案方式を用いてモバイルエージェントシステムを実現した．

4.1 ソースコード変換器の実装

ソースコード変換器の実装には，プリプロセッサ構築フレームワークであるEPP[18)]を用いた．EPPは，system mixinという一種の継承機構を用いることで

その動作を差分的に拡張可能な Java 言語のプリプロセッサであり，Java 言語仕様に準拠したパーザと型チェック機構を有する．EPP プラグインと呼ぶ拡張モジュールを追加することで，プリプロセッサとしての動作を変更・拡張することができる．EPP を利用したソースコード変換の実装には以下の利点がある．

第一に，フレームワークを利用することで，変換アルゴリズムの実装が容易になる．

第二に，EPP プラグインは composability が高く，他のプラグインと組み合わせて利用することが可能である．例えば，モバイルエージェントを実現するプラグインとパラメタ付きクラスの機能を提供するプラグインを同時に追加するだけで，パラメタ付きクラスをサポートするモバイルエージェント言語の処理系を実現できる．

具体的な変換は，大まかに次のような手順になっている—まず，EPP 本体によって型チェックまで完了した状態の抽象構文木を受け取り，その中から変換すべき対象のクラス定義を探す．そして，そのクラス定義中の mobile 宣言されたメソッド定義を元に，そのメソッドのスタックフレームを表す内部クラスの定義，および実行メソッドと実行復帰メソッドの定義を生成する．

4.2 モバイルエージェントシステムの仕様

本実装では，モバイルエージェントが継承すべきスーパークラスとして，MigThread というクラスが提供されている．MigThread は抽象クラスであり，プログラマは `run()` というメソッドを実装する必要がある．エージェントの本体部の実行は，このメソッドが呼び出されることにより開始される．

MigThread を継承したクラスの中ではマイグレーションのためのプリミティブ `migrate()` を使用することができる．`migrate()` を呼び出すメソッドは，そのメソッド宣言に mobile という修飾子をつける必要がある．また，別の mobile 宣言されたメソッドを呼び出す場合も同様である．mobile 修飾子は，ソースコード変換器が処理を行う際に，変換すべきメソッドを見つけるための目印である．

本実装ではエージェントの転送には Java RMI を用いている．そのため，エージェントの属性値や保存されたスレッドの状態などはオブジェクトシリアリゼーションによって転送されている．このため，エージェントのマイグレーションによる環境の変化などはオブジェクトシリアリゼーションの仕様に依存している．

図 14 に我々のシステムでのモバイルエージェントの記述例を示す．この例は，前節で述べた EPP を用いる利点である，他のプラグインと組み合わせて使用できるということの例にもなっている．プログラム先頭の#で始まる 2 行は EPP への命令で，ソースコード変換プラグインとして Generic および MigThread というモジュールを用いて処理を行うことを要請している．

```
#epp Generic
#epp MigThread

class TestGeneric extends MigThread {
  mobile public void run(){
    TStack<String> s
      = new TStack<String>();

    migrate(anotherHost);
    ...;
    s.push("str");
    ...;
    String str = s.pop(); // キャスト不要
    ...;
  }
}
```

図 14 本実現でのプログラム記述例
Fig. 14 Example code using our system.

5. 実　　験

4 章で述べた実現を用いて行った実験結果について述べる．実験環境として以下に示す環境を使用した．

- Sun Ultra60
  - CPU：UltraSPARC-II 360MHz
  - Memory：640MB
- OS：Solaris 2.6
- Java：JDK 1.2.1_03 （jit あり，HotSpot なし）
- Network：100Base-TX

5.1 ソースコード変換によるオーバヘッド

まず，ソースコード変換を行ったことにより生じる実行時間の増加を計測するための実験を行った．この実験では，マイグレーションを全く行わないプログラムを処理系に与え，出力されたプログラムを実行し，そのプログラムの実行が終了するまでの時間を計測した．実験には，フィボナッチ数を計算するプログラム（`fib`），クイックソートを行なうプログラム（`qsort`），N クイーン問題を解くプログラム（nqueen）の三種のプログラムを用いた．また，比較のために，実行メソッドと実行復帰メソッドの 2 種類のメソッドを作らずに，1 種類のメソッドのみを用いて実行した場合の計測も行った．それらの計測結果を**表 1** に示す．マイグレーションを行っていないので，この結果は実行メソッドだけを用いて実行を行った場合の実行時間であ

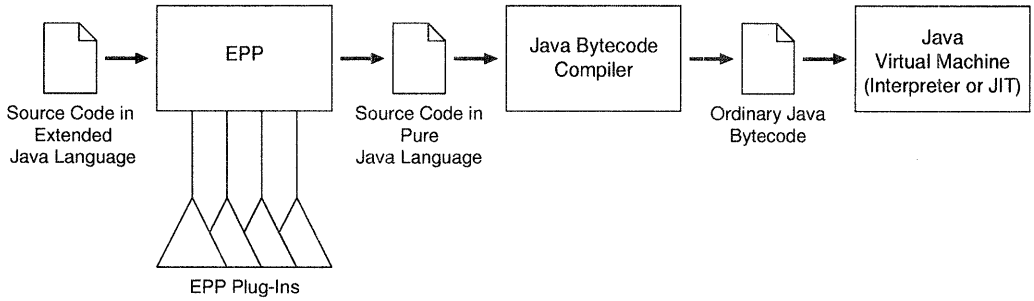


図 13 ソースコード変換のワークフロー
Fig. 13 Workflow of source code translation.

表 1 ソースコード変換による恒常的なオーバヘッド（単位：msec.，括弧内は増加率）
Table 1 Constant overheads by translation (msec.).

	変換無し	1 バージョン方式で変換	提案方式で変換
fib(39)	14740	19050 (129.2%)	15722 (106.7%)
qsort(10000000)	19438	19540 (100.5%)	19594 (100.8%)
nqueen(13)	13703	13523 (98.7%)	13810 (100.8%)

る．この実験結果の中では fib におけるオーバヘッドが最も大きくなっている．これは，fib のメソッド本体が小さいにもかかわらず変換により 2 つの try-catch および 1 つの一時変数が追加されるためである．実際のアプリケーションでは，これよりも小さいオーバヘッドで実行が進むと期待できる．

文献 14) では，同様のプログラムの実行時間は fib の場合で約 2.8 倍に増加するとの報告があり，本論文で提案した変換方式の実行時オーバヘッドは十分に小さいものと考えられる．この性能向上は，マイグレーションを行わない場合には常に実行メソッドが使用されるという性質に因る所が大きい．

実際には，マイグレーションを行った場合には実行復帰メソッドも使用されるので，この結果よりも実行時間は長くなる．しかし，

- 実行復帰メソッドが使用されるのはスタックフレームを復元する必要がある場合に限られる，
- 実行復帰メソッドは，実行が進むにつれてオーバヘッドが減少し，最終的には実行メソッドと同じになる，

という性質を考慮すると，マイグレーションの間隔を十分に取れば，モバイルエージェントは表 1 のような少ないオーバヘッドでプログラムを実行することが可能である．

5.2 モバイルエージェントの実行効率

実行メソッドだけでなく，実行復帰メソッドのオーバヘッドも含めた，総合的なオーバヘッドの評価を行うために，前章で説明したモーバイルエージェントシステムを用いてモーバイルエージェントを作成し，それを実際に実行して実行時間を計測した．実験に用いたモバイルエージェントのプログラムを**図 15** に示す．このプログラムは，フィボナッチ数を求めるプログラム中に，メソッド呼び出しが一定回数起こるたびにマイグレーションを行い，2 つのホストを往復しながら計算を進めるというプログラムである．

```
mobile public int mobileFib(int n) {
  if ((--count) == 0) {
    count = cycle;
    migrate(anotherHost);
  }
  if (n < 2)
    return 1;
  else
    return mobileFib(n - 1)
           + mobileFib(n - 2);
}
```

図 15 実験に用いたモーバイルエージェントのプログラム（一部）
Fig. 15 Program code for examination (a part).

このプログラムを用いて，マイグレーションを行う回数を変化させながら mobileFib(38) の計算を実行させた場合の所要時間を計測した．マイグレーションを行う回数を変化させるのは，我々の提案方式ではマイグレーションが頻繁に行われる程，実行時のオーバヘッドが増加することが考えられるためである．この実験の目的は，マイグレーションの頻繁さと実際のモバイルエージェントの実行効率との関係を調べることである．

所要時間は，プログラムの実行時間（Execute）とモバイルエージェントの移動（Transfer）にかかる時

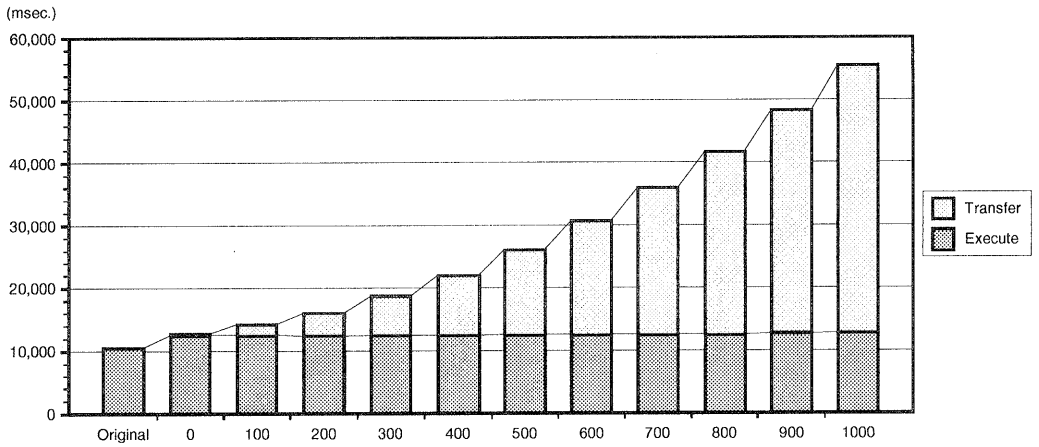


図 16 マイグレーション回数による所要時間の比較
Fig. 16 The processing times for execution and migration.

表 2 モーバイルエージェントの所要時間の内訳
Table 2 Itemization of processing time.

	実行	移動
変換元	10586	–
0	12521.9	315.3
100	12505.7	1688.5
200	12371.1	3664.2
300	12430.6	6290.0
400	12462.3	9586.6
500	12523.6	13534.4
600	12520.6	18122.8
700	12569.6	23400.9
800	12582.6	29066.5
900	12728.2	35640.4
1000	12803.5	42712.9

間に分けられる．プログラムの実行時間は，エージェントの本体部の実行にかかる時間の他に，移動に伴うスレッドの状態の保存・復元のための時間も含まれている．モバイルエージェントの移動にかかる時間は，RMI によるリモートホストの呼び出し，オブジェクトシリアリゼーションによる表現形式の変換，変換されたデータの送受信の時間などの時間の総和である．この実験の結果を表したのが**表 2** で，変化の様子を表したのが**図 16** である．この表および図中の移動回数とは，エージェントが計算を終えるまでに行ったマイグレーションの回数である．

計測の結果を見ると，マイグレーションの回数が多くなるにつれて，モーバイルエージェントが仕事を終えるまでの所要時間が長くなっているが，増加するのはほとんど移動のコストであることが読み取れる．マイグレーションを繰り返し行うことでプログラムの実行にも影響があるが，移動のコストの増加に比べると非常に少ない．この実験の結果から，実際にモーバイルエージェントに我々の変換方法を適用した場合には，変換によるオーバヘッドよりもむしろ移動のコストの増大が顕著になり，プログラム本体の実行効率の低下をほとんど気にせずにモバイルエージェントを実行させることができると考えられる．

6. おわりに

本論文では，ソースコード変換を用いてスレッドのモビリティを実現する方法について述べ，それを効率的に実現する方法について提案した．また，本論文ではその方法を実装したシステムを開発し，その性能を実験的に評価し，提案方式の有用性を示した．提案変換方式が出力するコードは標準的な Java コードであり，変換システム自体もすべて標準的な Java コードのみで実装されているため，ポータビリティも高い．

しかし，本論文で述べている方法だけでは，完全に Java 言語の仕様を満たしながらスレッドのモビリティを得ることはできない．例外処理のための try-catch は，文献 14) で述べられているような方法で対処する必要がある．他に，スレッド間の同期のためのロックがマイグレーションによって解放されてしまうという問題や，ローカルファイルをオープンしたまま移動した場合などの対処については本論文では未検討である．

今後は，以下のような課題に取り組みたいと考えている．第一に，static compiler での速度計測，マイクロベンチマークなどによる実行時オーバヘッドの原因の詳しい調査とそれに対する改善を予定している．第二に，本提案方式を用いた本格的なモバイルエージェントシステムの開発，および，広域分散環境でのアプリケーションの作成を予定している．アプリケーションの一例として，文献 7) で提案されているモーバイ

ル Web サーチロボットを実現することを予定している．第三に，提案方式をチェックポインティグ [13] や永続オブジェクトの実現等に応用することを検討している．第四に，バイトコードの変換技術と組み合わせて，実用的に有用性の高いモーバイルエージェントシステムを実現する方法を検討している．

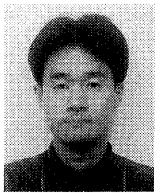
参考文献

1) Baumann, J., Hohl, F., Rothermel, K. and Straßer, M.: Mole - Concepts of a Mobile Agent System, *WWW Journal, Special issue on Applications and Techniques of Web Agents*, Vol. 1, No. 3, pp. 123–137 (1998).
2) Cardelli, L.: A Language with Distributed Scope, *Computing Systems*, Vol. 8, No. 1, pp. 27–59 (1995).
3) Frigo, M., Leiserson, C. E. and Randall, K. H.: The Implementation of the Cilk-5 Multithreaded Language, *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 212–223 (1998).
4) Fünfrocken, S.: Transparent Migration of Java-Based Mobile Agents, *Second International Workshop on Mobile Agents*, LNCS, Vol. 1477, Springer, pp. 26–37 (1998).
5) Gray, R. S., Kotz, D., Cybenko, G. and Rus, D.: D'Agents: Security in a multiple-language, mobile-agent system, *Mobile Agents and Security* (Vigna, G.(ed.)), Lecture Notes in Computer Science, Vol. 1419, Springer-Verlag, pp. 154–187 (1998).
6) Karnik, N. M. and Tripathi, A. R.: Design Issues in Mobile-Agent Programming Systems, *IEEE Concurrency*, Vol. 6, No. 3, pp. 52–61 (1998).
7) Kato, K., Someya, Y., Matsubara, K., Toumura, K. and Abe, H.: An Approach to Mobile Software Robots for the WWW, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 11, No. 4, pp. 526–548 (1999).
8) Kiniry, J. and Zimmerman, D.: A Hands-on Look at Java Mobile Agents, *IEEE Internet Computing*, Vol. 1, No. 4, pp. 21–30 (1997).
9) Lange, D. B. and Oshima, M.: *Programming and Deploying Java Mobile Agents with Aglets*, Addison Wesley (1998).
10) Myers, A. C., Bank, J. A. and Liskov, B.: Parameterized Types for Java, *Proc. 24th ACM Symp. on Principles of Programming Languages*, pp. 132–145 (1997).
11) ObjectSpace, Inc.: ObjectSpace Voyager. `http://www.objectspace.com/products/voyager/`.
12) Odersky, M. and Wadler, P.: Pizza into Java: Translating Theory into Practice, *Proc. 24th ACM Symp. on Principles of Programming Languages*, pp. 146–159 (1997).
13) Ramkumar, B. and Strumpen, V.: Portable Checkpointing for Heterogeneous Architectures, *27th International Symposium on Fault-Tolerant Computing*, IEEE Computer Society, pp. 58–67 (1997).
14) Sekiguchi, T., Masuhara, H. and Yonezawa, A.: A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation, *Coordination Languages and Models*, Lecture Notes in Computer Science, Vol. 1594, pp. 211–226 (1999).
15) Thorn, T.: Programming languages for mobile code, *ACM Computing Surveys*, Vol. 29, No. 3, pp. 213–239 (1997).
16) White, J.E.: Telescript Technology: The Foundation for the Electronic Marketplace, Technical Report, General Magic, Inc. (1994).
17) Wong, D., Paciorek, N. and Moore, D.: Java-based mobile agents, *Comm. of the ACM*, Vol. 42, No. 3, pp. 92–102 (1999).
18) 一杉裕志：高いモジュラリティと拡張性を持つ構文解析器, 情報処理学会論文誌：プログラミング, Vol. 39, No. SIG 1 (PRO 1), pp. 61–69 (1998).
19) 加藤和彦：モーバイルエージェントの実現技術, 日本ソフトウェア科学会主催チュートリアル「モーバイルエージェント」, 日本ソフトウェア科学会, chapter 2, pp. 9–45 (1999).
20) 佐藤一郎：モーバイルエージェントシステムの比較と研究動向, 日本ソフトウェア科学会主催チュートリアル「モーバイルエージェント」, 日本ソフトウェア科学会, chapter 2, pp. 79–109 (1999).
21) 八杉昌宏, 瀧和男：並列処理のためのオブジェクト指向言語 OPA の設計と実装, 情報処理学会研究報告 96-PRO-8 (SWoPP '96), Vol. 96, No. 82, pp. 157–162 (1996).

(平成 11 年 7 月 16 日受付)
(平成 11 年 12 月 29 日採録)

阿部 洋丈

1976 年生．1999 年筑波大学第三学群情報学類卒業．同年筑波大学博士課程工学研究科入学．分散協調システム，自律エージェントに興味を持つ．



一杉　裕志（正会員）

1965年生．1990年東京工業大学大学院情報科学専攻修士課程修了．1993年東京大学大学院情報科学専攻博士課程修了．博士（理学）．同年電子技術総合研究所入所．オブジェクト指向言語，並列・分散言語，拡張可能システムに興味を持つ．日本ソフトウェア科学会，ACM各会員．

加藤　和彦（正会員）

1962年生．1985年筑波大学第三学群情報学類卒業，1987年工学修士（筑波大学大学院工学研究科），1992年博士（理学）（東京大学大学院理学系研究科）．1989年東京大学理学部情報科学科助手，1993年筑波大学電子・情報工学系講師，1996年同助教授，現在に至る．オペレーティングシステム，プログラミング言語システム，データベースシステム，分散システム，モバイルオブジェクト計算に興味を持つ．情報処理学会，電子情報通信学会，日本ソフトウェア科学会，ACM，IEEE各会員．