通常の実行効率をほとんど損わないスレッドマイグレーションが可能なC++

多賀 奈由太† 関口 龍郎† 米澤 明憲†

我々は，C++ 言語を拡張して C++ の例外機構を利用した，通常の実行効率をほとんど損わないマイグレーション機能を実現した．通常の実行効率とは，スレッドマイグレーションの処理以外の部分の実行効率のことである．従来の多くのモーバイル言語システムでは，通常の実行効率を犠牲にしてスレッドマイグレーション機能を実装している．我々の方式はソースコード変換に基づいており多くのコンパイラとプラットフォームに適用することができる．対象となる言語は，マーシャリングができるようにポインタや union の取り扱いなどが制限されている．我々のマイグレーション方式は透明である．つまり，場所に依存する操作以外の部分のプログラムの意味はスレッドマイグレーションの前後で変化しない．我々は複数のアプリケーションについてベンチマークを実行し，オーバヘッドを分析し，実行効率がほとんど損われないことを確かめた．

An Extension of C++ That Supports Thread Migration with Little Loss of Normal Execution Efficiency

NAYUTA TAGA,† TATSUROU SEKIGUCHI† and AKINORI YONEZAWA†

We present a mobile C++ language system in which normal execution efficiency is hardly degraded, where normal execution efficiency means the execution efficiency of non-migration efficiency. In existing mobile programming languages, the normal execution efficiency is usually degraded due to the implementation of thread migration. Our implementation is based on source-code-level transformation, exploiting the C++ exception handling mechanism. Thus our thread migration mechanism can be applied to many platforms and it can be combined with many compilers. Compared to the standard C++, use of pointers and unions is restricted in our language so that data on the heap can be migrated to a remote host. Our migration scheme is transparent, namely the semantics of a single thread except location-dependent operations does not change with thread migration. We measured execution performance for several application programs, analyzed their overheads, and verified that the normal execution efficiency was hardly degraded.

1. はじめに

モーバイル言語システム[1]は異機種分散システムのソフトウェア実行環境の有望な基盤として注目されているが，そのようなシステムではマイグレーションによって分散するホスト間でプログラムの実行状態を移動させることができる．

プログラムが移動する際には，実行状態を保存し移動先ホストへ転送する必要がある．プログラムが移動先のホストで移動前と全く同じ実行状態でプログラムの続きを実行再開できる場合，そのような移動のことを『透明なマイグレーション』[2]という．もし，実行状態の保存と復元の処理をプログラマが明示的に記述する必要があるならば，そのような移動は透明なマイグレーションではない．既存の多くのモーバイル言語システム[3)~5)]ではマイグレーションは透明ではない．移動を行うプログラムを簡単に記述するという点と，移動のセマンティクスを簡単に理解するという点において透明なマイグレーションのほうが透明でないマイグレーションより望ましい[2),6),7)]．透明なマイグレーションを実現するためには，プログラムの実行状態（変数の値，関数のコールスタック，プログラムカウンタの値）の移動が必要である．

透明なスレッドマイグレーションが可能なモーバイル言語システムの中には，マシンの能力を有効に活用するため，または過去の豊富なソフトウェア資産を活用するために C，C++ 言語で記述されたプログラム

† 東京大学大学院理学系研究科 情報科学専攻
Department of Information Science, Graduate School of Science, University of Tokyo

を移動できるシステムがいくつか存在する[8)~11)]．そのようなシステムでのスレッドの移動方式は大きく2種類に分けられる．

ひとつは，スタックイメージをコピーすることによりマイグレーションを実現するシステムである[8)]．これらのシステムでは通常の実行効率の低下がなく，非常に高速な実行が可能である．しかし，実行状態を移動させるためには，スタックのレイアウトや関数のリターンアドレスに関する知識が必要になり，特定の実行環境やプロセッサに依存し，システムのポータビリティが失われてしまう．

もう1つはソースコード変換に基づくものである[10)]．ソースコード変換によりマイグレーションに必要な実行状態の保存や実行の再開を行うコードを挿入する．また，スタックを含む実行状態を保存するという点では，チェックポインティング[12)]はマイグレーションと本質的に同じ処理を行う．ソースコード変換によりポータブルなチェックポインティングを実現しているシステムとしてPorch[11)]が存在する．これらのソースコード変換に基づくシステムでは，状態を保存または回復するためのコードを挿入するために，マイグレーションに関係のない通常の場合の実行効率が低下する．

我々はソースコード変換技術を用いてC++言語のサブセットをマイグレーション可能にした．我々の手法における通常の実行効率へのオーバヘッドは多くの場合15%以内である．主要な技術は，以下の3つである．

- 例外機構を利用したローカル変数と実行ポイントの保存
- fast，slowバージョンの関数の生成[13)]による実行再開のコードの分離
- アドレスを取得しないローカル変数のマーシャリング

我々は透明なマイグレーションを実現するため，次のようにC++を拡張，制限した．

- 移動可能な関数とクラスを表すmigratoryキーワードを導入した．
- 移動を実行するgo関数を導入した．
- ポインタの不正な使用は禁止される．
- union型の使用は禁止される．

ソースコード変換を元にしているシステムとしてJava言語を元にしたものがあり[6),7)]，それらは我々の変換アルゴリズムと深い関わりがある．しかしJava言語上での変換とC++言語上の変換とでは，個々の変換のオーバヘッドへ寄与する割合が変化する．あるプログラムをそれらの手法でソースコード変換した場合，インタープリタ実行の時は，通常の実行効率に対するオーバヘッドは20%程度であったが，それをJITコンパイルした時にはオーバヘッドは70%に増大した．

プログラムを遠隔ホストに移動させる手法にプロセスマイグレーション[14)]と呼ばれるものがある．多くのプロセスマイグレーションは同じアーキテクチャのホスト間でのマイグレーションを行う．メモリイメージをそのままコピーし，移動前と同じプログラムカウンタで実行を再開する．プロセスマイグレーションで移動できるリソースはファイル記述子のみに制限されることが多い．モーバイル言語システムではウィンドウなどの資源を移動させる必要がある．また，多くのモーバイル言語システムはインターネット上での利用を前提としているため異機種間でマイグレーションすることが重要になる．

これ以降のこの論文の構成は次の通りである．2章ではマイグレーションを制御するためにC++に導入した言語拡張と制限について述べる．3章では我々の移動方式について大まかに説明する．4章では，関数の直後へgotoするために実行ポイントを明らかにする変換について述べ，5章では，関数の実行を中断し実行状態を保存するためにtry-catchを挿入する変換について説明する．6章では，実行状態を復元して実行再開のために必要な変換について述べる．7章では実行状態をマーシャリングするために必要な変換について説明する．8章では現在の実装でのソースコード変換器の作成方法と移動方法について述べる．9章では実行効率とコード量の変化についての実験結果を述べる．10章ではこの論文のまとめを行う．

2. マイグレーションのための言語拡張と制限

この章では，マイグレーションを記述するためにC++言語に新たに追加したキーワードmigratoryと関数goについて説明を行った後，マイグレーションを可能にするために導入された制限について述べる．

2.1 マイグレーションの実行

マイグレーションはgo関数を実行することによって行われる．

```
printf("Let's go !");
go("harp.is.s.u-tokyo.ac.jp");
printf("Arrive !");
```

goの引数は移動先ホスト名である．goが実行されると実行状態とプログラムが移動先ホストへ転送され，そこでgo以降の文の実行が再開される．上の例では，移動元ホストで"Let's go !"と表示され，移動先のホ

ストで "Arrive !" と表示されることになる．

我々のマイグレーション方式は透明である．関数の実行途中に go が実行されると，ローカル変数の値，関数中のどの位置まで実行したかという情報（実行ポイント），ヒープの状態を転送し，移動先ホストでは go の直後の命令から実行を再開する．

2.2 マイグレーション可能関数

マイグレーション可能関数とは，その関数内でマイグレーション可能関数または go を実行する可能性がある関数である．マイグレーションはマイグレーション可能関数の中でのみ行うことができる．マイグレーション可能関数には，宣言時に migratory キーワードを付加しなければならない．マイグレーション可能関数はマイグレーション可能関数内からのみ呼び出すことができ，マイグレーション可能でない関数からは呼び出すことはできない．main 関数はマイグレーション可能関数として宣言しなければならない．したがって，main 関数では任意のマイグレーション可能関数を実行することができる．

2.3 マイグレーション可能クラス

マイグレーション可能クラスとは，そのクラスのオブジェクトがマイグレーション時に転送されるようなクラスである．整数や実数などの組み込みデータ型の値や，配列，ポインタとポインタから参照されるデータは，マイグレーション時にシステムが移動先ホストへ転送するが，クラスのオブジェクトを転送するためには，以下のようにマイグレーション可能クラスとして宣言しなければならない．

```
class migratory AClass { ... };
```

マイグレーション可能関数とマイグレーション可能クラスだけがソースコード変換の対象となる．

2.4 制　　限

この節では我々の現在の実装におけるプログラマが守らなければならない制限について述べる．

まず，ポインタは正しい型のポインタ変数に保持しなければならない．ヒープ上のオブジェクトへのポインタが指定とは異なる型の変数に格納されていると，その型のオブジェクトとしてマーシャリングが行われるので正しく移動できない．また，ポインタ以外の型へ格納されたポインタや，正しい値を持っていないポインタを作成してはならない．ただし，仮想関数を持っている基本クラスへのポインタが派生クラスのオブジェクトを参照している場合は問題ない．

union 型は使用できない．これは，union 型の変数にどの型の値が格納されているか正確に知るのが難しいためである．

これらの制限は，マーシャリング，アンマーシャリング関数にユーザによるカスタマイズを可能にする仕組みを設けたり，union 型に現在の型を表すタグを設けることによって，解決できると思われる．

3. 我々の移動方式の概要

透明なマイグレーションを実現するためにプログラムが明示的に実行状態を取り扱えるようにプログラムを変換する．この章では我々の移動方式の概要について説明する．次章以降において具体的な変換アルゴリズムについて説明を行う．

マイグレーション可能関数をソースコード変換することによって 2 種類の関数（fast，slow バージョン）を生成する．fast バージョンは通常の実行で用いられる．slow バージョンはマイグレーションの後に実行を再開する時のみ用いられる．fast バージョンには実行の中断と実行状態の保存を行うコードが挿入される．slow バージョンにはそれに加えて実行状態の復元と実行の再開を行うためのコードが挿入される．

fast，slow バージョンの関数を生成するのは，fast バージョンの関数へ実行再開のためのコードを挿入しないことにより，実行再開のためのコードのオーバヘッドが，通常の実行に影響を与えないようにするためである．

3.1 関数の実行中断と実行状態の保存のための変換

関数の実行状態は，関数の中のどこから実行を再開するかという情報（実行ポイント）と，その関数のスタックフレーム中の有効なローカル変数（引数と初期化は行われているがまだ解体されていない変数）から構成されている．

マイグレーション可能関数を次のように変換して fast バージョンの関数を生成する．

(1) 式の途中での実行の中断が可能になるように式の分解を行う．
(2) 関数内のマイグレーション可能関数の呼び出しは fast バージョンを呼び出すように変更する．
(3) 関数内のマイグレーション可能関数を呼んでいる場所の周囲に，特別な例外 wcxx::LetsGo を受け取る try-catch 文を配置する．そして catch ブロックの中に関数の状態をマーシャリングするコードを配置する．

プログラムは，我々のシステムのランタイムライブラリが main 関数の fast バージョンを実行することにより開始される．fast バージョンの関数は fast バージョンの関数またはマイグレーションを行わない関数しか呼び出さない．

プログラムがマイグレーション可能関数内で go を実行すると，wcxx::LetsGo 例外が発生する．マイグレーション可能関数はその例外を catch し，関数の実行状態をマーシャリングして，再びその例外を throw する．そして，それを繰り返し，最終的に main 関数を呼び出したランタイムライブラリがその例外を catch し，ヒープなどの状態をマーシャリングして go の引数で指定されたホストへコードとマーシャリングした実行状態を送る．

3.2 実行状態の復元と実行の再開の概要

マイグレーション可能関数を次のように変換して slow バージョンの関数を生成する．

(1) fast バージョンの関数と同じ変換をする．
(2) 関数の途中からの実行の再開が可能になるようにラベルとジャンプ命令を挿入する．
(3) 関数の先頭に実行状態をアンマーシャリングするコードを挿入する．

移動先のホストでプログラムが実行再開すると，ランタイムライブラリが main 関数の slow バージョンを呼び出す．

関数の slow バージョンは，まず実行状態をアンマーシャリングし，アンマーシャリングした実行ポイントに従って実行途中だった関数の slow バージョンの関数を呼び出し，それから実行を中断した場所の直後へ goto する．このようにしてマイグレーション時に呼び出し中だった全ての関数の slow バージョンが呼び出され，関数のコールスタックが再構築される．

slow バージョンの関数は，実行途中だった fast バージョンの関数に対応する slow バージョンの関数を実行再開直後に呼び出す時以外は全て fast バージョンの関数を呼び出す．そのため，実行を再開した後の処理は fast バージョンの関数とほぼ同じになる．

4. 実行ポイントを明らかにする変換

この節で述べる変換は次の 2 つの目的を持っている．

第一に，slow バージョンの関数では，関数の途中から実行を再開できなければならないので，関数中の移動する可能性のある地点，すなわちマイグレーション可能関数の直後へジャンプできる必要がある．C++ 言語には goto 文があり，それを使えば関数中のどの点へもジャンプできるが，ローカル変数の初期化をバイパスするようなジャンプは禁止されている．また式の途中へは goto できないという問題を解決する必要もある．

第二に，fast，slow 両バージョンにおいてマイグレーションが発生したときにそれが自分の呼び出したどの関数の中で発生したのかを正確に求める必要がある．

我々の方式では，変数宣言と初期化を分離することによってローカル変数の初期化をバイパスするようなジャンプを可能にし，マイグレーション可能関数を呼び出す式を分解することによって式の途中へのジャンプとマイグレーションを発生させた関数の特定を可能にしている．

4.1 マイグレーション可能関数を呼び出す式の分解

前節で述べた理由により，マイグレーション可能関数の中に出現する全てのマイグレーション可能関数の呼び出しの直後へ goto できるように式を分解する必要がある．例えば，次のような式を考える（hoge と muha はマイグレーション可能関数）．

```
int n = hoge() + muha();
```

このような式は，以下のように一時変数を用いて分解する．

```
int tmp1 = hoge();
label1:
int tmp2 = muha();
label2:
int n = tmp1 + tmp2;
```

この変換の結果，hoge と muha の呼び出し直後から実行を再開できるようになる．

4.2 変数宣言と初期化の分離

C++ ではローカル変数の初期化をしている場所をバイパスするような goto は禁止されている．例えば，次のような関数を考える．

```
void migratory hoge() {
  int n = 1; // ローカル変数 n の初期化
  muha();   // マイグレーション可能関数
  ... use n ...
```

このような関数の muha(); の直後から実行再開をするために，以下のように goto を挿入すると，goto label; が n の初期化を飛び越えているので C++ コンパイラはエラーを出す．

```
void hoge() {
  if (resuming) goto label;
  int n = 1; // ローカル変数 n の初期化
  muha();   // マイグレーション可能関数
label: ... use n ...
```

そこで，我々の方法では変数宣言と初期化を分離し，変数宣言を関数の先頭にまとめることによってこの問題を回避している．上の例では以下のように分離する．

```
void hoge() {
  int n; // ローカル変数 n の宣言
  if (resuming) goto label;
```

```
    goto label;
    n = 1; // ローカル変数 n の初期化
    muha();
  label: ... use n ...
```

この変数宣言と初期化の分離は 4.1 節で導入された一時変数にも同様に適用する必要がある．

4.3 クラスの変数宣言と初期化の分離

整数や実数やポインタなどの単純な型の変数ならば 4.2 節のように変数宣言と初期化を簡単に分離できる．しかし変数の型がコンストラクタを持つクラスの場合は，宣言すると同時に初期化も行われてしまうため，単純に 4.2 節の方法を適用するわけにはいかない．そのため，そのクラスのオブジェクトを格納するバッファをスタック上に確保しそのバッファに対してコンストラクタを呼び出すという手順を踏む必要がある．例えば，次のようなコードを考える（クラス AClass はコンストラクタを持つとする）．

```
void hoge() {
    AClass a;
    muha();
    ... use a ...
```

ここでは，変数 a の宣言と同時にコンストラクタが呼ばれている．そこでまずクラス AClass と同じサイズのバッファを用意し、初期化をしている場所では、そのバッファに対して初期化を行う．

```
void hoge() {
    char buf[sizeof(AClass)];
#define a (*(AClass *)buf) // 変数宣言
    goto label;
    new (&a) AClass(); // ローカル変数 a の初期化
    muha();
  label: ... use a ...
    a.~AClass(); // ローカル変数 a の解体
}
```

new (&a) AClass() は，new の配置構文で，&a の位置のメモリに AClass を構築する．この変換によりデストラクタが自動的に呼び出されなくなってしまうので，スコープから出るときには明示的にデストラクタを呼ぶように変換する．

この変換は，C++ コンパイラが暗黙に生成する処理を明示的に記述しているにすぎない．

4.4 マイグレーション可能関数の返値がクラスの場合の式の分解

マイグレーション可能関数の返値がクラスの場合は式の分解に特別な扱いを必要とする．例えば，

```
AClass migratory hoge(void) {
    ...
    return ...;
}
```

のようなマイグレーション可能関数があるとすると，

```
    a.muha(hoge()); // (1)
```

という文は，4.1 節の変換方式により一時変数を用いて単純に分解すると，

```
    AClass tmp(hoge()); // (2)
  label1:
    a.muha(tmp);
```

となる．しかし（2）の文では AClass のコピーコンストラクタを呼び出すが，(1) の文ではコピーコンストラクタは呼び出しておらず，プログラムの意味が変わってしまう．そこで我々の方法では，返り値が生成されるメモリをスタック上にとりそのアドレスを渡すことでこれを防いでいる．上のプログラムの断片は次のように変換される．

```
    char buf[sizeof(AClass)];
#define tmp (*(AClass *)buf)
    hoge(&tmp);
  label1:
    a.muha(tmp);
```

tmp の初期化を以下のように関数 hoge の中ですることによってコピーコンストラクタが呼び出されるのを避けることができる．

```
void hoge(AClass *return_value) {
    ...
    new (return_value) AClass( ... );
}
```

この変換の結果，AClass *return_value という引数が増えているために，関数に引数を 1 つ渡す分のオーバヘッドが生じる．

5. 実行中の関数の中断と実行状態の保存

fast，slow バージョンの関数では，移動が起こる可能性のある場所で実行状態をマーシャリングするコードを挿入する必要がある．移動が起こる可能性のある場所とは，マイグレーション可能関数を呼び出している場所である．マイグレーション可能関数の周囲に，移動の時に発生する wcxx::LetsGo 例外を捕まえるための try-catch を挿入する．そして，catch ブロックの中では関数の実行状態をマーシャリングし，その後 wcxx::LetsGo 例外を再 throw する．

この変換による通常の実行へのオーバヘッドは try-catch の実装に依存し，try 文を素通りする時のオーバヘッドに等しい．Solaris 2.6 上で egcs-2.91.66 によっ

てコンパイルした場合は，UltraSPARC では無条件ジャンプ 1 つと delay slot 1 つ，Pentium II では無条件ジャンプ 1 つのオーバヘッドが生じる．しかし，Microsoft Windows（Win32）では，Win32 の構造化例外処理の実装のために非常に大きなオーバヘッドを必要としており，Visual C++ では (try-catch の数) + 12 命令程度，Cygnus の cygwin b20.1 の egcs-2.91.66 では (try-catch の数) × 16 命令程度のオーバヘッドが生じる．

5.1 マイグレーション可能関数を呼び出す式の分解

マイグレーション可能関数の中に出現する全てのマイグレーション可能関数の呼び出しの周囲に try-catch 文を挿入できるようにする必要があるので，4.1 節のように式を分解しておく．

5.2 wcxx::LetsGo 例外の捕捉とマーシャリング

移動の時に発生する wcxx::LetsGo 例外を捕まえるために，5.1 節で分解された式の周囲に，try-catch を挿入する．そして catch ブロックの中では，関数の実行状態をマーシャリングし wcxx::LetsGo 例外を再 throw する．また，slow バージョンの関数は実行再開処理中にのみ呼ばれる関数なので，マイグレーション可能関数を呼んでいる場所は全て fast バージョンを呼ぶようにする．例えば，

```
... do something 1...
AClass a;
int i = 0;
int n = hoge() + muha(&i);
... do something 2...
```

というコードは，fast バージョンでは

```
... do something 1...
AClass a;
int i = 0;
try {
  int tmp1 = hoge_fast();
} catch (wcxx::LetsGo) {
  wcxx::marshalObjectAt(&a);
  wcxx::marshalObjectAt(&i);
  wcxx::marshalState(0);
  throw;
}
try {
  int tmp2 = muha_fast(&i);
} catch (wcxx::LetsGo) {
  wcxx::marshalObjectAt(&a);
  wcxx::marshalObjectAt(&i);
  wcxx::marshalObjectAt(&tmp1);
  wcxx::marshalState(1);
  throw;
}
int n = tmp1 + tmp2;
... do something 2...
```

のように変換される．ここで，wcxx::marshalObjectAt はローカル変数のアドレスを与えるとマーシャリングするランタイム関数であり，型ごとにオーバロードされている．wcxx::marshalState は引数に現在の実行ポイントを受け取り，それをマーシャリングするランタイム関数である．

しかし，このような変換には 2 つの問題がある．1 つは，一時変数の宣言が try ブロックの中に入ってしまい一時変数 tmp1 と tmp2 が try ブロックを出ると無効になってしまうこと．もう 1 つは，ローカル変数 a は catch ブロックの最後の throw が実行されると解体されるが，プログラム自体は移動するだけであって終了するわけではないのでこの時点で解体されるとプログラムの意味が変わってしまうという問題である．

これらの問題は，slow バージョンの関数の場合と同様に 4.2〜4.4 節で説明したように変数宣言と初期化を分離することで解決できる．よって，ここまでの変換での fast バージョンと slow バージョンの違いは，ジャンプのためのラベルが付いているかどうかという点のみとなる．上記の例は以下のようになる．

```
  char buf[sizeof(AClass)];
#define a (*(AClass *)buf) // 変数宣言
  int tmp1; int tmp2; int i; int n;
  ... do something 1...
  new (&a) AClass(); // ローカル変数 a の初期化
  i = 0;
  try {
    tmp1 = hoge_fast();
  } catch (wcxx::LetsGo) {
    wcxx::marshalObjectAt(&a);
    wcxx::marshalObjectAt(&i);
    wcxx::marshalState(0);
    throw;
  }
 label0: // slow バージョンのみ
  try {
    tmp2 = muha_fast(&i);
  } catch (wcxx::LetsGo) {
    wcxx::marshalObjectAt(&a);
    wcxx::marshalObjectAt(&i);
    wcxx::marshalObjectAt(&tmp1);
```

```
    wcxx::marshalState(1);
    throw;
  }
label1: // slow バージョンのみ
  n = tmp1 + tmp2;
  ... do something 2...
```

6. 関数実行を再開可能にする変換

関数実行を再開するためには，関数の途中のどの実行ポイントから再開するか決定して，関数の実行状態を復元する必要がある．この復元のための変換はslowバージョンの関数にのみ必要である．以下の関数を例にとって説明する．(ただしmuhaはマイグレーション可能関数である)

```
void migratory hoge(int n) {
  while (0 < n) {
    n -= muha(n);
  }
}
```

slowバージョンの関数は，まず，実行ポイントをアンマーシャリングする．次に，その実行ポイントに従って，ローカル変数と引数をアンマーシャリングし，移動する時に実行が中断してしまった関数のslowバージョンを実行する．最後に，中断した関数の直後へgotoする．上の例では，以下のように変換される．

```
void migratory hoge_slow() {
  int n; // 引数
  int tmp1;
  switch (wcxx::unmarshalState()) {
    case 0:
      wcxx::unmarshal(&n); // (1)
      try {
        tmp1 = muha_slow();
      } catch (wcxx::LetsGo) {
        wcxx::marshalObjectAt(&n);
        wcxx::marshalState(0);
        throw;
      }
      goto label0;
  }
  while (0 < n) {
    try {
      tmp1 = muha_fast(n);
    } catch (wcxx::LetsGo) {
      wcxx::marshalObjectAt(&n);
      wcxx::marshalState(0);
      throw;
    }
    label0:
      n -= tmp1;
  }
}
```

ただし実行ポイント0はmuhaを呼び出した直後を指している．この関数にはマイグレーション可能関数呼び出しが1つしかないので実行を再開するポイントも1つしかない．このように変換することによって，実行を再開する時以外はfastバージョンと同じコードを実行できるようになる．

slowバージョンの関数は引数を受け取る必要はない．なぜなら，関数の実行状態をマーシャリングした時に引数の情報までマーシャリングしているからである．この例では(1)で引数をアンマーシャリングしている．

7. マーシャリングのための変換

この章ではマーシャリングするために必要な，マイグレーション可能クラスの変換，ヒープへの配列確保の変換，ポインタの変換について述べる．

ヒープ上のデータ構造を別のコンピュータに転送できるために我々のシステムでは型が特定できるようにC++のデータ構造を制限している．マーシャリングによりオブジェクトは異機種間で交換可能なバイト列へ変換される．

ヒープ上のデータ構造のマーシャリング方法は，我々のスレッドマイグレーションの方法とは独立であり，別のマーシャリング方法を考えても良い．

7.1 マイグレーション可能クラス

マイグレーション可能クラスは次のように変換する．まず，全てのクラスにユニークな識別子(id)が付けられる．これは，ヒープ上のオブジェクトをアンマーシャリングする時にどのクラスのオブジェクトかを区別するためである．さらに，クラスにアンマーシャリング用のコンストラクタとマーシャリング用の関数を付け加える．

例えば，

```
class migratory MClass : public MBase{
  public:
    double d;
};
```

というクラスの場合は，以下のようなコードへと変換される．

```
class MClass : public MBase{
```

```
  public:
    double d;
    MClass(wcxx::Unmarshalling u)
      : MBase(u),
        d(wcxx::unmarshal(&d)) { }
    static const wcxx::idv_t id[];
};
const wcxx::idv_t MClass::id[] = { 100 };
void wcxx::marshal(MClass const *objptr) {
    wcxx::marshal((MBase *)objptr);
    wcxx::marshal(&objptr->d);
}
```

wcxx::marshal 関数は各型ごとにオーバロードされているマーシャリング関数である．引数に与えられたオブジェクトのアドレスをランタイムに記憶させ，そのオブジェクトの内部にポインタ変数を含んでいるのならばポインタが参照している先を辿るという処理をする．ランタイムに記憶させたアドレスは，ポインタ変数をマーシャリングする際に使用する．ヒープ上にあるオブジェクトをマーシャリングする必要のある時は，ランタイムライブラリがそのオブジェクトの型に対応する wcxx::marshal 関数を呼び出す．

wcxx::unmarshal 関数は各型ごとにオーバロードされているアンマーシャリング関数である．引数にオブジェクトのアドレスを受取り，そのオブジェクトのアドレスをランタイムに記憶させ，オブジェクト自身をアンマーシャリングする．ランタイムに記憶させたオブジェクトのアドレスは，ポインタ変数をアンマーシャリングする時に使用する．

アンマーシャリング時には，ランタイムライブラリが wcxx::Unmarshalling オブジェクトを引数にしてコンストラクタを呼び出しアンマーシャリングされる．コンストラクタは，メンバフィールドをアンマーシャリングする前に親クラスをアンマーシャリングする．

仮想関数が1つでも存在するクラスの場合は，静的にオブジェクトの型が分からなくなるので id 取得用の仮想関数も追加する．

7.2 ヒープ

スタック上の変数をマーシャリングする場合は，5.2節のように変換することによって，型ごとにオーバロードされた wcxx::marshalObjectAt ランタイム関数を適切に呼び出すことができるが，ヒープ上のデータの場合は型を特定しなければならない．我々は以下のようにしている．

スタック上に存在するポインタ変数からポインタを辿り，マーシャリングする．この時，スタック上のポインタの型からそのポインタから参照されているヒープ上のオブジェクトの型が特定できる．しかし，仮想関数を持つクラスへのポインタは，派生クラスのオブジェクトを参照している可能性がある．その場合は，オブジェクトの実際の型を特定するために id 取得用の仮想関数を実行する．

しかし，ヒープ上の配列の要素数はポインタを辿るだけでは知ることができない．そこで，配列をヒープに割り当てる場合は型とサイズを記録するようにする．具体的には new ではなく wcxx::new_array 関数を使うように変更する．

例えば，

```
int *i = new int[3];
int *j = new int;
```

は，

```
int *i = wcxx::new_array(3, id_int);
int *j = new int; // 配列ではないので変換しない
```

と変換される．id_int はランタイムライブラリが定義している int 型の id である．

ポインタを辿ってヒープ上のオブジェクトをマーシャリングする際に，配列の要素へのポインタやオブジェクトのメンバ変数へのポインタなどに注意する必要がある．例えば，オブジェクトのメンバ変数へのポインタがある場合，そのメンバ変数を1つのオブジェクトとしてマーシャリングしてしまうと，もはやそのオブジェクトをオブジェクトとしてアンマーシャリングすることができなくなってしまう．

そこで我々の方法では，まずポインタを全て辿り全てのオブジェクトの位置とサイズを記録する．その後マーシャリングを行うことで，配列の要素へのポインタやオブジェクトのメンバ変数へのポインタなどがある場合でも適切に対処できる．

7.3 ポインタ

異機種混在環境ではある型の値がそれぞれのホスト上では違う表現形式を利用している場合がある．我々は移動時にはそれぞれの型に標準的な形式を用意し，その形式に変換して転送している．整数や実数などの組み込みデータ型の値の場合は一般的にバイト順序と型のサイズを適切に変換すれば良いが，ポインタ型の場合はポインタのアドレスを再配置できるように適切に変換する必要がある．

C++ 言語では，クラスや配列は int などの組み込みデータ型の集まりでできており，マーシャリング処理の最小単位はそのような組み込みデータ型となっている．マーシャリング処理では整数とポインタと構造体と配列の組合わせとしてヒープ上のオブジェクトを

解釈する．

7.4 ローカル変数のマーシャリングの最適化

ここまでの変換では，ローカル変数がポインタで参照されている可能性を考慮して，ローカル変数をマーシャリングするときにローカル変数のアドレスをwcxx::marshalObjectAt ランタイム関数に与えることでマーシャリングしていた．しかし，一般に int などの単純な変数のアドレスを取ることは通常の実行に大きなオーバヘッドをかける（9 章参照）．なぜなら，その変数をレジスタだけに割り付けることができず，スタックにストアしたりロードしたりしなければならなくなるからである．

我々の方式の場合，ローカル変数のアドレスを取っている箇所は catch ブロックの中であり，しかも catch ブロックは throw で終わっているので，賢いコンパイラならば catch ブロック以外の場所ではローカル変数をレジスタ上に割り付けることが可能なはずである．しかし我々が評価に使用した egcs では，catch の後に書かれている部分では，スタックへ変数の値をストアするコードとスタックから値をロードするコードが挿入される．例えば，

```
int i = 0;
hoge();
i++;
func();
```

（hoge はマイグレーション可能関数）というプログラムを変換すると，

```
int i;
i = 0;
try {
  hoge_fast();
} catch (wcxx::LetsGo) {
  wcxx::marshalObjectAt(&i);
  wcxx::marshalState(0);
  throw;
}
i++; // (1)
func();
```

このようになるが，本来必要ないにも関わらず egcs では（1）の直前に i をスタックからレジスタへロードし，(1) の直後に i をレジスタからスタックへストアするというコードが生成される．

プログラム中でアドレスが一度も取られていない変数については，その変数がどこか他のポインタから参照されている可能性はないので，マーシャリング時にもアドレスを取らないように変換する．具体的には上の例では，wcxx::marshalObjectAt(&i) ではなくwcxx::marshalNeverPointed(i) に変換する．この変換の結果，他に原因がない限り変数 i がスタックに割り当てられることはなくなる．

8. 実　　装

ソースコード変換器のプロトタイプは，Edison Design Group[15] の C++ front end（以下 EDG）に約 4000 行の C コードを追加して作成した．さらにマーシャリングのためのライブラリを約 2000 行の C++ コードで作成し，移動してきたプログラムを受け入れるエージェントサーバを Perl で数 10 行で作成した．ただし，我々のソースコード変換方式は特定の変換器やサーバの実装に依存するものではない．

EDG は，C や C++ のソースを読み込んで abstract syntax tree を作成し，back end に渡すものである．back end には通常 C や C++ コンパイラが想定されているが，EDG には C や C++ のソースを出力する back end が付属しているので我々はそれを利用した．C++ front end の部分を改造して migratory キーワードを認識するようにし，C++ のソースを出力する back end を改造し，式の分解，try-catch の挿入，fast，slow バージョンの関数の生成，移動可能クラスのマーシャリング，アンマーシャリングをするコードの生成などを行った．

移動させたいアプリケーションプログラムは，そのソースを EDG によるソースコード変換処理後，egcs-2.91.66 によりコンパイルを行い，マーシャリングするためのランタイムライブラリとリンクして実行バイナリを作成する．

移動は次のように行われる．まず，移動先のホストでエージェントサーバを動かす．アプリケーションプログラムの実行中に移動が起こると，ランタイムライブラリはマーシャリングした実行状態と，ソースコード変換前のオリジナルのソースを他のホスト上のエージェントサーバへ転送する．実行状態とソースを受け取ったエージェントサーバはそれらをディスクに保存し，ソースをソースコード変換し，コンパイルして実行バイナリを作成し，それを特別なオプションを付けて実行する．特別なオプションにより，ランタイムライブラリは実行直後にディスクに保存された実行状態を読み取り，その後 main 関数の slow バージョンを呼び出すことで実行を再開する．

我々はプログラムを移動させるためにソースを転送する方法を用いたが，適切なバイトコード[16] を転送することも考えられる．

表 1 実行時間
Table 1 Elapsed time.

	オリジナル	我々の方式	Porch
qsort(4000)	36.4s	37.0s (+2%)	47.1s (+29%)
fib(40)	40.0s	36.1s (−10%)	66.7s (+67%)
multimat	14.0s	15.2s (+9%)	34.5s (+146%)
bintree	3.4s	3.9s (+15%)	4.9s (+44%)

表 2 実行時間 (Arachne)
Table 2 Elapsed time (Arachne).

	Arachne
fib(30) (構造体メンバへのアクセス +ヒープへの状態変数の確保)	5762s (+14305%)
fib(30) (構造体メンバへのアクセス)	597s (+1392%)

9. 評 価

我々の変換方式によるオーバヘッドを評価するためにいくつか簡単なアプリケーションでベンチマークテストを実行した．**表 1** はマイグレーションをしない通常の場合の実行時間を表している．比較のため Porch[11] と Arachne[10] (**表 2**) の実行時間も載せる．我々の方法では 15% 以内のオーバヘッドとなっている．このオーバヘッドの最大の要因は，try 節の挿入による．

測定に使用したアプリケーションは次のようなものである．qsort は 4000 個のランダムな整数を保持する配列の quick sort を 4000 回行うもの．fib はフィボナッチ関数を計算するもの．multimat は 200×200 の整数行列の掛け算を 10 回行うもの．bintree は 2 分木にランダムな整数を 100000 個挿入するもの．測定は全て UltraSPARC 168MHz 上で egcs-2.91.66 でコンパイルしたもので行い，6 回実行した時の平均実行時間を測定した．

Arachne 方式はスタックフレームに相当する構造体をヒープ上に作り，スタックへのアクセスはその構造体へのアクセスに変換されている．オーバヘッドの大部分は関数呼び出しのたびに構造体をヒープ上に作るコストである．それを省いたバージョンでも，構造体アクセスに大きなオーバヘッドがかかっていることが分かる．

Porch はソースコード変換によりポータブルなチェックポインティングを実現しているシステムである．チェックポインティングはスタックを含む実行状態をストレージへ保存する必要があり，マイグレーションと本質的に同じ処理を行う．Porch の方式と我々の方式との違いは次の 3 個所からなる．それは，(1) 実行再開のための分岐 (2) 人工のプログラムカウンタ (3) 各関数へのマイグレーションの通知方法である．

表 3 Porch のオーバヘッドの分析
Table 3 An analysis of Porch's overhead.

	qsort	fib
オリジナル	36.4s	40.0s
Porch で変換後	47.1s (+29%)	66.7s (+67%)
変数のアドレス取得	8.1s (+22%)	15.1s (+38%)
再開のための分岐	1.8s (+5%)	10.0s (+25%)
人工のプログラムカウンタ	0.8s (+2%)	1.7s (+4%)

表 3 では，qsort と fib について Porch のオーバヘッドを分類している．主なオーバヘッドは「変数のアドレス取得」である．

Porch では 7.4 節の最適化が全く行われていないため「変数のアドレス取得」のオーバヘッドが生じるが，我々の方法ではその最適化を行うことによりオーバヘッドを削減している．

Porch では我々の方法で言うところの slow バージョンの関数のみが用意されているので，実行再開のための switch のオーバヘッドが常に存在するが，我々は fast バージョンの関数を用意することによって「再開のための分岐」のコストを fast バージョンの関数から削減した．

Porch では，実行の再開位置を記録するために，チェックポインティングが起こる可能性のある関数 (我々の方法ではマイグレーション可能関数に当る) の呼び出しの直前にその実行ポイントを人工のプログラムカウンタに保存している．また，実際にチェックポインティングが行われたかどうかを判定するために例外ではなくグローバル変数を使用している．チェックポインティングが起こる可能性のある関数の呼び出しから return した直後に if 文を挿入してチェックポインティングが行われたかどうかを判断している．我々の方法では，try-catch ブロックを挿入することによってこれらのオーバヘッドと人工のプログラムカウンタを削減した．

表 4 は我々の方法で変換後の実行バイナリのサイズである．ただしランタイムライブラリは含んでいない．我々の方式ではおよそ 5〜9 倍であるが Porch よりもサイズは小さい．一般に，変換結果のサイズは，関数内のローカル変数の数とマイグレーション可能関数を呼び出している回数の積に比例する．

表 5 では関数の fast バージョンと slow バージョンの実行時間の差を示している．この測定で使用したアプリケーションは，それぞれ 400 回繰り返す 3 重ループの中で空の移動可能関数を 6 回実行するものである．

表4 実行バイナリのサイズの増加（バイト）
Table 4 Glowth in binary size (bytes).

	オリジナル	変換後 (倍率)	Porch (倍率)
qsort	2080	12252 (5.9)	22472 (10.8)
fib	1476	7548 (5.1)	20108 (13.6)
multimat	2160	12444 (5.8)	24760 (11.5)
bintree	2100	18776 (8.9)	21608 (10.3)

表5 slow バージョンのオーバヘッド
Table 5 Overhead of slow version.

オリジナル	18.569s
我々の方式（fast）	30.820s (+66%)
我々の方式（slow）	31.685s (+71%)
我々の方式（fast'）	21.429s (+15%)

ただしこの表では，slow バージョンの冒頭の switch の部分は実行を再開してから一度しか実行されないので実行時間には含めていない．つまり，この表における slow バージョンのオーバヘッドとは，goto とラベルの挿入によってプログラムの実行フローが変わったために最適化ができなかったことを原因として生じるオーバヘッドである．例えば，

```
void migratory func()
{
  int i = 0;
  hoge();
  i++;
}
```

（hoge はマイグレーション可能関数）このような関数は以下のように変換されるが，

```
void f_slow()
{
  int i;
  switch (wcxx::unmarshalState())
  {
   case 0:
    i = wcxx::unmarshalNeverPointed_int();
    try {
      hoge_slow();
    } catch (wcxx::LetsGo) {
      wcxx::marshalNeverPointed(i);
      wcxx::marshalState(0);
      throw;
    }
    goto label0;
  }
  i = 0;
  try {
    hoge_fast();
  } catch (wcxx::LetsGo) {
    wcxx::marshalNeverPointed(i);
    wcxx::marshalState(0);
    throw;
  }
  label0:
  i++; // (1)
}
```

(1) の所へたどり着く実行フローが 2 種類存在してしまう．1 つは switch の case 0 の最後から goto label0 でたどり着く実行フローである．もう 1 つは switch をそのまま通りすぎて最初から順番にプログラムを実行していく実行フローである．この 2 種類の実行フローがあるためにレジスター割り当てなどの最適化が妨げられてオーバヘッドが生じてしまう．この測定結果では fast バージョンの関数よりも実行時間に対して 5% 余分にオーバヘッドが生じている．

try 節のコンパイル方法については最適化の余地がある．それによりソースコード変換のためのオーバヘッドをさらに削除することができる．SPARC の egcs では try ブロックの後に短いコードが挿入されるため try ブロックを抜けた後に無条件分岐命令が必ず挿入されている．

egcs で以下のようなプログラムをコンパイルすると，

```
try { ... (A) ... }
catch ( ... ) { ... (B) ... }
try { ... (C) ... }
catch ( ... ) { ... (D) ... }
... (E) ...
return;
```

以下のようなアセンブリコードになる．

```
... (A) ...
jump to block C
call throw
... (C) ...
jump to block E
call throw
... (E) ...
return
... (B) ...
... (D) ...
```

例外処理の方法を工夫すれば，call throw と，その直前の jump を除去するような最適化は可能である．それを手作業で行ったプログラムの実行時間を測定した結果が 表 5 の fast' である．fast' は fast から 51% の

オーバヘッドが削減できる．ただし fast' への変換をしたのは 表5 のみであり，他の表での我々の方式の測定結果は fast バージョンを使用しているので，fast' への変換をすることによりさらにオーバヘッドを削減できる可能性がある．

10. まとめ

我々は，ソースコード変換技術を用いて，マイグレーションをしない通常の場合の実行効率をほとんど損うことなしに C++ 言語でマイグレーションを行う方式を考案しその実装を行った．我々の方式は実行効率を重視するモーバイル言語システムに有効であると考えられる．

変換の際に重要なのは，例外機構を利用したスタックなどの実行状態の保存，fast，slow バージョンの関数の生成による実行再開のコードの分離，アドレスを取得しないローカル変数のマーシャリングである．ベンチマークによれば，特にローカル変数のアドレスを取得することなしにマーシャリングを行うことは，コンパイラの最適化技術を有効に活用させるために有用であった．

我々の方法では，マイグレーションをしない場合の通常の実行効率が多くの場合 15% ほどしか損われない．我々の方法はコンパイラの try-catch の実装方式や最適化技術に深く依存しており，try-catch を効率的に実装できない Microsoft Windows（Win32）のような環境では実行効率が損われてしまう可能性があるが，そのような時にどう効率的に実装するかは今後の課題である．

参考文献

1) Cugola, G., Ghezzi, C., Picco, G.P. and Vigna, G.: Analyzing Mobile Code Languages, *Mobile Object System: Towards the Programmable Internet*, Lecture Notes in Computer Science, Vol. 1222, pp. 93–109 (1996).
2) Gray, R.S.: Agent Tcl: A Transportable Agent System, *CIKM Workshop on Intelligent Information Agents, Fourth International Conference of Information and Knowledge Management* (1995).
3) Cardelli, L.: A Language with Distributed Scope (1994). `http://www.research.digital.com/src/publications/cartoons/src-rr-122.html`.
4) Oshima, M., Karjoth, G. and Ono, K.: Aglets Specification 1.1 Draft (1998). `http://www.trl.ibm.co.jp/aglets/spec11.html`.
5) ObjectSpace, Inc.: ObjectSpace Voyager. `http://www.objectspace.com/products/vgrOverview.htm`.
6) Fünfrocken, S.: Transparent Migration of Java-based Mobile Agents, *Second International Workshop on Mobile Agents*, Lecture Notes in Computer Science, Vol. 1477, pp. 26–37 (1998). `http://www.informatik.tu-darmstadt.de/VS/Mitarbeiter/Fuenfrocken/`.
7) 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). `http://www.yl.is.s.u-tokyo.ac.jp/amo/`.
8) Kato, K., Toumura, K., Matsubara, K., Aikawa, S., Yoshida, J., Kono, K., Taura, K. and Sekiguchi, T.: Protected and Secure Mobile Object Computing in Planet, *2nd ECOOP Workshop on Mobile Object Systems*, pp. 319–326 (1996). `http://www.osss.is.tsukuba.ac.jp/~planet/papers.html`.
9) Steensgaard, B. and Ju, E.: Object and Native Code Thread Mobility Among Heterogeneous Computers, *Fifteenth Symposium on Operating Systems Principles* (1995).
10) Dimitrov, B. and Rego, V.: Arachne: A Portable Threads System Supporting Migrant Threads on Heterogeneous Network Farms, *IEEE Parallel and Distributed Systems*, Vol. 9, No. 5, pp. 459–469 (1998). `http://www.cs.purdue.edu/~bdd/Purdue/resume.html`.
11) Strumpen, V.: Compiler Technology for Portable Checkpoints (1999). `http://theory.lcs.mit.edu/~porch`.
12) Deconinck, G., Vounckx, J., Cuyvers, R. and Lauwereins, R.: Survey of Checkpointing and Rollback Techniques, Technical Report O3.1.8 and O3.1.12, ESAT-ACCA Laboratory, Katholieke Universiteit Leuven, Belgium (1993).
13) 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).
14) Smith J.M.: A Survey of Process Migration Mechanisms, *Operation System Review*, Vol.22, No. 3, pp. 28–40 (1988).
15) Edison Design Group: EDG C++ Front End. `http://www.edg.com/cpp.html`.
16) Lucco, S., Sharp, O. and Wahbe, R.: Om-

niware: A Universal Substrate for Web Programming. http://www.w3.org/Conferences/WWW4/Papers/165/.

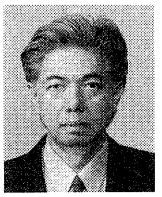
(平成11年 7 月16日受付)
(平成11年12月29日採録)

多賀奈由太

1975年生．1998年東京大学理学部情報科学科卒業．現在東京大学大学院理学系研究科情報科学専攻修士課程に在学中．主にモーバイル言語システムの研究に従事．

関口 龍郎（学生会員）

1970年生．1999年より日本学術振興会研究員．1999年東京大学大学院より理学博士取得．主にモーバイル言語システムの研究に従事．



米澤 明憲（正会員）

1947年生．1977年 Ph. D. in Computer Science（MIT）．1989年より東京大学理学部情報科学科教授．超並列・分散ソフトウェアアーキテクチャなどに興味を持つ．共著書「モデルと表現」等（岩波書店），編著書「ABCL」（MIT Press）等がある．1992–1996年ドイツ国立情報処理研究所（GMD）科学顧問，ACM Transaction on Programming Languages and Systems 副編集長，IEEE Parallel & Distributed Technology および Computer 編集委員などを歴任，元日本ソフトウェア科学会理事長，ACM Fellow．