実行移送機能のあるトランスレータの性能改善

小林　正史†　　梅村　恭司‡
豊橋技術科学大学情報工学系†‡

1 はじめに

分散処理の基礎技術としてエージェントがある。エージェントとは、コンピュータ間の通信において、あるコンピュータから他のコンピュータに対して、実行を依頼する手続きそのものを転送して処理する方式で、中間に立つソフトウェア機能を指している。この手続きは、送り手のコンピュータで処理が行われている途中で送ることができ、受け手側のコンピュータでその処理の続きを行うことができる。つまり、異なる機種の間においてプロセスの移送を可能とする。我々がこれから述べるエージェントとは、更に異機種間で実行コンテクストを移動できる機能をもち、この機能をもつ言語をエージェント言語と定義する。

エージェントを実現する方法のひとつとしてトランスレータを用いる方法があり、先行研究の段階で実現までの基本的な指針は既に示されている。しかし、これまでの方式で生成された翻訳コードでは実行速度が大きく低下する。そこで本研究では、トランスレータの基本的な方針はこれまでのものを踏襲しつつ、実行速度の低下が少なくなるような翻訳コードの生成手法を提案する。

2 改善前のトランスレータ

2.1 原始プログラム言語

下川[1]のトランスレータは、実行移送機能を想定した簡易エージェント言語から、特定の規約に準じたＣコードを生成する。このエージェント言語の主な仕様は以下の通りである。

・基本的な文法は PASCAL に基づく
・データは整数型のみ
・再帰呼び出し可能
・手続き定義のネストはなし
・制御文は if 文、while 文、複合文
・手続きの引数は値渡し
・実行移送文 migrate をもつ

migrate 文は、引数として与えられたマシン名のコンピュータシステムに移動してプログラムを再開する命令である。我々のトランスレータもこの点は同じである。

2.2 実行コンテクストのテキスト化

実行コンテクストとは、ある時点でのプログラムの状態すべてのことであり、これを使うことによってプログラムを途中から実行することができる。機種に依存せずに移送するために、プログラムの実行コンテクストは機種独立なテキスト形式へ変換する。実行コンテクストは、call/return 時のアドレスと変数の値からなる。

Ｃ言語の実行途中を機種独立なテキストに変換する際、関数呼び出しの状態をテキストから復元することが問題となる。戻りアドレスはプログラムのコード内部のエントリーであり、その値を直接操作することはできないからである。そこでＣ言語の組み込み機能を利用せず、自分で用意したグローバルなスタックを使い、呼び出し・戻り場所を関数名に対応させて実現する。翻訳結果のコードではこのスタックを関数ポインタのスタックとして表現し、Ｃのスタックを実質的に使わないようなコード生成を行う。Ｃの制御スタックを使わないことで、実行途中の状態をテキストに変換することが可能となる。このスタックの実行ルーチンは図１のようなものである。つまり、実行を待つ仕事を積むスタックがあり、その先頭から１つ取り出しては呼び出すという作業の繰り返しで全体が動作する。これにより、関数呼び出しについてテキスト形式の実行コンテクストを生成できる。

2.3 コーディング規約

コーディング規約とは、原始プログラムをＣ言語に変換する際の変換規則である。実行コンテクストをテキスト化するため、前述のようにコード生成はＣのスタックを使わない方針に基づいて行われる。そのため関数呼び出し、分岐、ローカル変数についてはそのままＣに翻訳せずに、ある制限に従った変換が必要となる。

2.3.1 関数呼び出し

コード生成時には、関数呼び出しがあるところで元の関数を２つに分け、関数呼び出し以前・以降の手続きをそれぞれ独立の関数とする。そして呼び出しを行うコードは戻りに対応する関数のポインタをスタックに積み、関数を終了する。これにより呼び出したい関数が実行され、そのあと呼び出し後のコードが実行される（図２）。

2.3.2 分岐（if、while 文）

分岐もそのまま変換することはできない。Ｃコードでは関数呼び出しの前後で２つの関数に分割されるので、原始プログラムのループ中に関数呼び出しを含むとそこで別の関数に分かれてしまう。そのためループ機能を維持するための構造変換が必要となる。つまり分岐が発生

```
typedef struct FS{
  int sp;
  void (*fp[4098])(void);
}F_Stack;
F_Stack fs;

int main(void)
{
   ・
   ・
  void (*func)(void);
  fs.sp=-1;
   ・
   ・
  while (fs.sp>-1)
    {
      func=fs.fp[fs.sp--];
      (*func)();
    }
   ・
   ・
}
```

図1 実行のメインルーチン

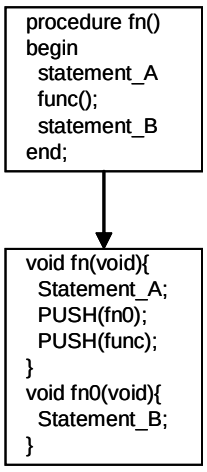

図2 構造変換例（関数呼び出し）

Improving the Performance of the Translator for Process Migration
† Masashi Kobayashi
‡ Kyoji Umemura
†‡ Department of Information and Computer Sciences, Toyohashi University of Technology

する部分で、生成するコードを２つの関数に分解する。分岐の発生する部分で関数を新たに生成し、その直前で対応する関数ポインタをスタックに積み、実行中の関数を終了する。このような操作により、分岐の機能を維持する。

2.3.3 ローカル変数

原始プログラムの関数内のローカル変数については、その関数が翻訳後には２つ以上のＣの関数に分割されている可能性があるため、そのままＣのローカル変数としては翻訳できない。そこで、前述の関数ポインタからなるスタックとは別に変数用のスタックを用意する。関数呼び出しで引数を渡すときは、呼び出し側で引数をスタックにプッシュし、呼ばれた関数側でこれをポップする方式をとる。また関数が分割されると、次に続く関数にこれまで扱ってきた変数の値を渡さなければならないので、対応する関数ポインタをプッシュすると同時に、変数の値も変数用のスタックにプッシュし、そして続く関数での変数宣言と共にその値をポップする。

3 高速化手法の提案

3.1 従来の手法の問題点

先行研究の段階では、生成されたＣコードの実行時間は通常のＣコードに比べて 10 倍以上長くなってしまっていた。これほど膨大な実行遅延が生じる原因は主に２つ考えられる。ひとつは、原始プログラムをＣプログラムに翻訳する際、原始プログラムをまずアセンブラの命令に分解し、その命令１つ１つをＣ言語で表現していたことである。もうひとつは、分岐における構造変換を無条件で常に行っていたことである。if 文、while 文で構造変換を行うのは、これらの内部に関数呼び出しや実行移送命令が含まれていた場合に対処できるようにするためであり、実際これらの命令を含まないような制御文で構造変換を行うことは余計な実行遅延が生じる原因となる。本研究ではこれらの問題を解決すべく、以下のような変更を行った。

3.2 アセンブラ分解の廃止

我々は、原始プログラムを解析する際、原始コードをアセンブラ単位の命令に分解する操作を廃止し、直接Ｃコードを生成する方法を用いた。我々の設定した原始プログラム言語の構文には、１対１対応でＣ言語に翻訳できるものもあり、これによって生成コードのサイズを小さくすることができた。この変更を便宜的にローカルな変更と呼ぶことにする。

3.3 プログラム情報の保持

制御文の構造変換を常に行わなければならなかったのは、制御コードを処理する段階でその内部に関数呼び出しが含まれているかどうか分からなかったからである。このような情報を得るために、我々はコード生成処理に入る前に一度原始プログラムをすべて読み、プログラム全体の情報を tree 構造で保持するようにした。予めプログラム全体の内容を知ることによって各コードの前後関係を把握し、制御文における構造変換を必要な部分にだけ行うようにした。これにより、余分な構造変換による実行遅延をなくすことができた。この変更を便宜的にグローバルな変更と呼ぶことにする。

4 評価実験

4.1 コードの実行

表1 各Cコードの実行時間(単位 : sec)

	プログラムA	プログラムB
通常のCコード	0.045	0.000093
ローカル変更	0.808(18 **倍**)	0.000102 (1.10 **倍**)
ローカル＋グローバル変更	0.106(2.4 **倍**)	0.000094 (1.01 **倍**)

これまでに述べた変更を行い、生成コードの実行時間が通常のＣコードと比較してどのように変化したのかを調べた。実験として、２種類の原始プログラムに対して、それぞれに３種類のＣプログラムを用意して実行時間を計測した。比較に用いた３種類のＣコードとは、実行移送できない通常のＣコード、ローカルな変更のみを加えたトランスレータが生成したＣコード、そしてローカルな変更とグローバルな変更を行ったトランスレータが生成したＣコードの３つである。この実験で得られた結果を表１に示す。ここで、プログラムＡは手続きの再帰呼び出しを利用して 8-Queen の問題を解くプログラムである。またプログラムＢは四則演算、if 文による絶対値計算、while 文によるべき乗計算を行うものである。こちらは再帰呼び出しのような、頻繁に手続きを呼び出すような処理は行わない。

4.2 考察

表１の結果から、今回新たに提案した方式で生成されたＣコードについて次のようなことがいえる。まずプログラムＢの結果を見ると、トランスレータが生成したコードと通常のコードの実行時間はほとんど同じであり、代入演算処理に関しては実行速度の低下がほとんど生じないといえる。プログラムＡではローカルな変更、ローカル＋グローバルな変更を行った生成コード両方において速度低下がはっきり見られた。ローカルな変更のみの生成コードでは制御文に対してすべて構造変換を行っているので 18 倍もの低下が生じているが、グローバルな変更も加えて不要な構造変換を排除することで最終的には２倍強にまで短縮することができた。この速度低下の原因となるのは、構造変換による関数の細分化で頻繁に行われる関数呼び出しのオーバーヘッドである。しかしながら現行の方式でこの速度低下は回避できないものであり、この遅延を最小限に抑えることが目的であるので、今回のアプローチによる効果は十分に得られたと考えられる。

5 まとめ

今回提案したコード生成方式によって、翻訳Ｃコードの実行時間は通常のＣコードの 10 倍以上から高々２倍程度にまで短縮することができた。今回の改善によって実行速度の低下がなくなったとは言えないが、現在の方針を貫く上でこれ以上の大幅な速度改善を実現するのは難しいと考える。一方、エージェント言語の仕様がまだ単純であるため、機能を拡張して更に検証を行う必要がある。

[参考文献]

[1] 下川僚子、梅村恭司：トランスレータを利用した機種非依存な実行移送方式、情報処理学会論文誌、Vol.40、No.6、pp.2553-2562(1999)