ソフトウェア工学 126－6
（2000．3．9）

既存ソフトウェアのターゲットアーキテクチャ変更に関する研究

佐伯俊道† 今泉貴史† 鈴木正人††

あるマシン上で動作しているプログラムを別のマシン上で実行可能にするためにソースコードの再コンパイルやプログラムの変更といった作業を必要とする。かし、ソースが失われたレガシーコードや実行ファイルのみで配布されているプログラムの場合はその作業が困難、あるいは不可能である。本稿ではあるマシン上で実行可能なバイナリコードを、別のマシン上で実行可能なバイナリコードに変換するという操作（ターゲットアーキテクチャ変更）を自動的に行なうフレームワークを提案する。可変部とするアーキテクチャ定義を極力簡潔に記述することにより、容易に対象アーキテクチャを追加/拡張可能である。

Binary translation techniques for legacy software

Saeki Toshimichi,† Imaizumi Takashi† and Suzuki Masato††

Transformation of a binary executable code from an architecture to another architecture usually requires re-compiling the source code or re-programming. But it is difficult or sometimes impossible, especially a legacy software whose source code was already lost. In this paper we propose a framework for retargetting binary transformation. This framework supports it automatically and applicable to many architectures by giving simple specifications.

1. はじめに

現在、あるマシン上で動作しているプログラムを別のマシン上で使用可能にするには、再コンパイルや再プログラミングといった手法を用いている。このうち再コンパイルについては、コンパイル対象となるソースコードが存在することが必須となるため、ソースコードが失われたプログラムの場合これを行なうことが出来ない。再プログラミングとは、同等の動作を行うプログラムを組み直すことである。動作が単純な、簡単なプログラムの場合は問題にならないが、巨大なプログラムを白紙の状態から組みなおすことは必ずしも現実的解決策とはいえない。

これらを行なうことが出来ない場合に用いる手法として、ターゲットアーキテクチャ変更がある。ターゲットアーキテクチャ変更とは、ソースマシンアーキテクチャM_S及びソースオペレーティングシステムOS_S上で実行可能なバイナリコード$bc(M_S, OS_S)$を、異なるアーキテクチャ及びOSを持つターゲットマシン上で実行可能なバイナリコード$bc(M_T, OS_T)$に変換する操作である。この操作を正確に行えば、意味の変更を防いだ上でのバイナリコード→バイナリコードの移植作業は可能である。しかし、これを手作業で行うことは現実的ではない。

このターゲットアーキテクチャ変更を自動的に行うことができれば、ソースが失われたレガシーコードや実行ファイルでのみ配布されているプログラムを別のマシン上で使用することが容易になる。また、古いプラットフォーム上で動作している既存ソフトウェアを、最小限の労力で新しいプラットフォーム上に移植することが可能となるため、新しいプラットフォームの普及促進効果が期待できる。

ターゲットアーキテクチャ変更を自動的に行う試みは従来から盛んに研究されている。例えばFX!32[1),2)]はx86マシン上で動作しているWindowsNTアプリケーションをAlphaマシン上のWindowsNTで動作させることに成功している。しかし、従来の自動的なターゲットアーキテクチャ変更を目的とした研究・製品では、ソースマシン・ターゲットマシンともに特定のアーキテクチャを想定している。別のソース・ターゲットの組み合わせを利用する場合には、ターゲットアーキテクチャ変更方法を新たに設計・実装する必要

† 東京工業大学 大学院 理工学研究科
Graduate School of Science and Engineering, Tokyo Institute of Technology
†† 東京工業大学 大学院 情報理工学研究科
Graduate School of Information Science and Engineering, Tokyo Institute of Technology

がある。また、従来の動的なターゲットアーキテクチャ変更を目的とした研究・製品は高いマシン依存度を有しており、資源の再利用については全く考慮されていない。

2. 目　　的

本研究では、ソースマシン・ターゲットマシンの柔軟な追加拡張を可能とする、自動的なターゲットアーキテクチャ変更のためのフレームワークを提案する。

さらに、新たに対応マシンアーキテクチャの追加を行う際の利便性を考慮し、アーキテクチャ仕様の記述量の少ない方法を提案する。仕様の記述はマシンアーキテクチャから機械的に導出できるように配慮し、ソースバイナリコードで使用された言語やコンパイラからの独立性を高めることを目標とする。

3. フレームワーク

本稿で提案する既存ソフトウェアのターゲットアーキテクチャ変更の実現を目的とするフレームワークは

- Object File Parser
- IR Generator
- Code Generator

の 3 つのモジュールと 4 つの仕様の記述からなる (図 1)。

3.1 構　　成

Object File Parser は、ソースマシン上で実行可能なバイナリコード $\mathrm{bc}(M_S, OS_S)$ を、入力であるソースマシンの仕様 $\mathrm{binspec}(M_S, OS_S)$ をもとに解析する。このモジュールは、$\mathrm{bc}(M_S, OS_S)$ から $\mathrm{asm}(M_S)$ を出力する。$\mathrm{asm}(M_S)$ は以下の情報を含む。

- 実行コードの (マシン依存な) アセンブリ言語ソース
- ダイナミックシンボルテーブル情報
- データセクションの開始・終了アドレス
- データセクションのバイナリダンプ
- ジャンプテーブルの情報

ソースマシンの仕様 $\mathrm{binspec}(M_S, OS_S)$ は、M_Sのニモニック及びバイトオーダの情報、OS_Sの実行可能コードの構成情報からなる。出力 $\mathrm{asm}(M_S)$ は主にプロセッサに依存するアセンブリ言語ソースであり、上記のような付随情報を適宜添付する。例えばメモリ状態の初期化が必要でなければデータセクションのバイナリダンプは添付されず、多条件分岐がコード中に出現しなければジャンプテーブル情報は添付されない。

IR Generator は本フレームワークのコアモジュールである。入力は Object File Parser の生成する $\mathrm{asm}(M_S)$ と、マシンアーキテクチャの仕様 $\mathrm{spec}(M_S)$、出力はマシン独立な中間表現 IR である。$\mathrm{asm}(M_S)$ のアセンブリ言語ソースから、$\mathrm{spec}(M_S)$ に含まれる命令変換表、すなわち M_Sのニモニックと IR 命令列との対応表をもとに、一部アーキテクチャ依存な中間表現を生成する。さらに $\mathrm{spec}(M_S)$ に含まれるプロセッサ定義に基づき preIR の抽象度を引き上げ、アーキテクチャ独立な中間表現 IR に変換する。

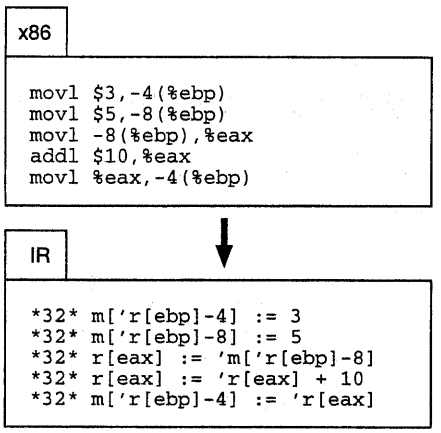

図 2　バイナリから IR への変換例

Code Generator は IR Generator の生成した IR と、ターゲットマシンアーキテクチャの仕様 $\mathrm{spec}(M_T)$ 及びターゲットマシンバイナリコードの仕様 $\mathrm{binspec}(M_T, OS_T)$ を入力とし、ターゲットマシン上で動作可能な実行コード $\mathrm{bc}(M_T, OS_T)$ を生成する。ターゲットバイナリコードの生成には大きく分けて以下のような 3 種類の方法が考えられる。

- 中間表現から実行可能コードを生成する
- 中間表現をインタプリタ上で直接実行する
- 中間表現を高級言語のソースに変換しコンパイルする

Code Generator ではコード生成プロセスを明確に定義してはいない。使用者の要求にしたがって上記 3 つの方法を選択的に用いることができる。本稿では、ターゲットバイナリコードが実際に動作を確認することを優先するため、比較的実現が容易な「高級言語への変換」を採用している。

3.2 中間表現 IR

IR は、プログラムの実行に際して必要な定義を記述する部分 (IR 定義部) と、実際のプログラムコード (IR コード) からなる。IR コードは簡単な手続き型言

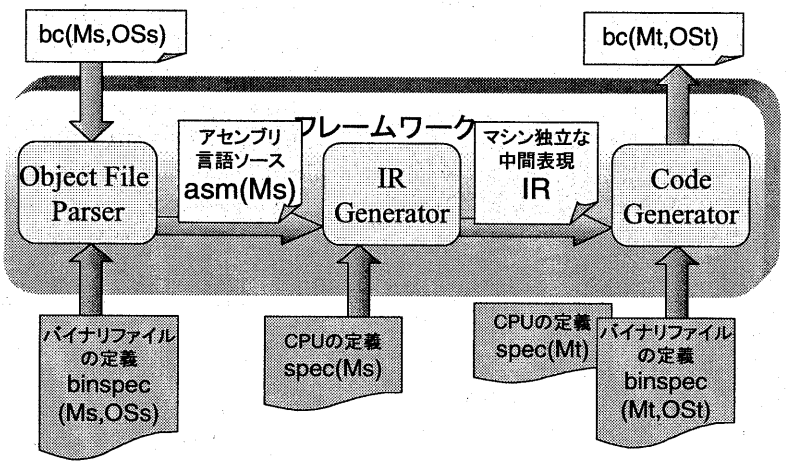

図 1 フレームワーク

語の体裁をとっており、その構成は一般的な手続き型言語と同様に、関数宣言部とそれに続く関数本体をひとまとめとしたブロックの集合である。

元となるバイナリコードを単純に代入文に置き換えて RTL 形式に変換したとしても手続きの概念は復元されないが、一般的なプロセッサには手続きの概念を実現するための命令セット (call や ret など) が用意されているため、それを解析することによって手続き型言語 IR を生成する。

手続き呼び出し部分の実現方法はプロセッサによって大きく異なる。IR のマシン独立性を維持するために IR コードはマシン依存性のない手続き呼び出し方法を持つ手続き型言語としている。

3.3 IR の文法概要

IR コードは関数宣言部と関数本体をひとつのブロックとし、そのブロックの集合として構成される。関数宣言部では関数名とパラメータのサイズを宣言し、関数本体は文の集合である。IR の文は、一般的なプロセッサを動作させるための最小限の制御文と代入文からなる。

- 無条件・条件ジャンプ
- パラメータを渡しての手続き呼び出し
- 手続きからのリターン
- 式または値の代入

変数には主に以下の種類のものがある。変数の値を表す場合、変数にプライムをつける。

- レジスタ/フラグ
  レジスタは `r[eax]` のように記述し、これはレジスタ `eax` を表す。フラグは `flag[ZF]` のように記述し、これはフラグ `ZF` を表す。
- メモリ
  `'m[10000]` は 10000 番地のメモリの値を意味する。
- 有意変数
  特定の意味を持つ変数であり、その用途・使用局面は限定されている。

式は一般的な算術・論理式をサポートする。条件式ではフラグに対してのみ演算が可能で、論理和 (|)、論理積 (&)、否定演算子 (!) を用いる。

3.4 参照ファイル

IR Generator が参照する spec(M_s) は、以下に示す 2 つのファイルから成る。

3.4.1 プロセッサ定義

プロセッサ定義 にはソースマシンアーキテクチャで使用されるプロセッサの情報が記述される (図 3)。

IR 定義部に記述される REGISTER、FLAG、

CHANGEFLAG セクションはプロセッサ定義からコピーされたものである。それぞれ、プロセッサが使用するレジスタとそのビット数、フラグ、フラグ変化条件を記述したものである。

他にプロセッサ定義にはディレイ命令の情報を記述するための DELAYINST セクションと、関数呼び出しを復元するための情報を記述する PROCEDURE セクションがある。

3.4.2 命令変換表

命令変換表は asm(M_S) におけるアドレッシングの IR 表記への変換及び命令の IR 書式にもとづく文への変換を行う際に用いられる (図 4)。

命令変換表は、asm(M_S) のオペランドを変数あるいは値に変換するための「オペランドテーブル」とマシンコードのニモニックを IR コードに変換するための「ニモニックテーブル」からなる。

4. IR Generator の変換プロセス

IR Generator は、asm(M_S) 及び spec(M_S) を入力とし、IR を生成する。変換プロセスは 2 つの段階からなる。

- **第一段階**
  ここでは spec(M_S) に含まれる命令変換表にしたがって、asm(M_S) に含まれるアセンブリ言語ソースを逐語的に IR コードに変換する。またラベリングを行い、分岐命令の飛び先をラベルに置き換える。第一段階で生成される IR は、IR コードの文法に基づかない中間的な IR コードを持つ。これを preIR と呼ぶ。
- **第二段階**
  第一段階で生成された preIR を、プロセッサ定義に従って IR コードに変換する。preIR は抽象度が低く一部アーキテクチャ依存であるため、この第二段階でマシンアーキテクチャに大きく依存する関数呼び出しの意味を復元し、preIR の抽象度を引き上げ、マシンアーキテクチャに依存しない IR コードを生成する。さらに IR コードに 4 つのセクションを付加し、最終的な IR を出力する。

4.1 関数呼び出しの復元

IR Generator 第二段階変換プロセスでは、第一段階で生成した preIR コードを、プロセッサ定義にしたがって IR コードに変換する。preIR コードは、抽象的な関数呼び出しの機構を持っていない。また、関数のエントリポイントも他のラベルと同じ扱いとなっている。

4.2 caller における引数復元

ここで、標準ライブラリ関数など、取るべき引数の個数が判明している関数について、その呼び出し側における引数復元のプロセスについて説明する。

4.2.1 プロセッサ定義の記述

IR Generator の入力となるプロセッサ定義には、ABI (Application Binary Interface) によって規定された引数/返り値を記述する。プロセッサ定義の PROCEDURE セクションに以下のような記述を行う。

PROCEDURE における引数の書式

```
argument : lower : upper : arg_spec
```

これは、*lower* 番目から *upper* 番目の引数は *arg_spec* で表される値である、ということを意味している。*lower* は 0 から始まるため、

```
argument:0:3:*
```

は 1 から 4 番目の引数を表す。また *upper* は省略可能で、その場合上限なしを意味する。

また、返り値は以下のように表される。

PROCEDURE における返り値の書式

```
return : ret_spec
```

例えば、SPARC プロセッサにおける記述は以下のようになる。

```
argument:0:5: 'r[o(number)]
argument:6::  'm['r[sp] + (number*4+68)]
return:       'r[o0]
```

ここで、括弧でくくられた部分は括弧内の式を評価した結果の値となる。number は、何番目の引数であるかをゼロベースで表している。

この記述によって例えば 3 番目の引数は 'r[o2] であり、8 番目の引数は 'm['r[sp] + 100] であることがわかる。

x86 系プロセッサにおける記述も示す。

```
argument:0::  'm['r[esp] + (number*4+4)]
return:       'r[eax]
```

この場合、例えば 3 番目の引数は 'm['r[esp] + 16] である。

4.3 引数復元プロセス

引数復元のプロセスは、ある関数呼び出しに対し、その呼び出しにおける引数の個数分、引数変数 argument への代入文を列挙する。

例えば 引数を 2 つ持つことがわかっている関数 *foo* への call 命令周辺の命令列に対応する preIR コードの例を以下に示す。(x86) この例では 1 番目の引数に

```
DELAYINST
  bne,be,bgu,b,call,ret ....
REGISTER
  eax:32:
  ebx:32:
  ...
FLAG
  ZF,CF,VF,PF,AF,OF
  ...
CHANGEFLAG
  case ADD: if (result == 0) ZF=1;
  ...
PROCEDURE
  argument:0:: 'm['r[esp] + (number*4+4)]
  ...
```

図 3　プロセッサ定義の記述の一部

```
Operand Table
  \(%(-),%(-),(-)\)  ->  m['r[str1] + 'r[str2] * value3]
  \(%(-)\)           ->  m['r[str1]]
  %(-)               ->  r[str1]
  (-)                ->  value1

Mnemonic Table
  movl  ->  *32* op2 := op1_value
  jne   ->  -    jump label(op1_value) if !flag[ZF]
  addl  ->  *32* temp := op2_value + op1_value
            -    _ChangeFlag(ADD,32,op2_value,op1_value,temp)
            *32* op2 := 'temp
```

図 4　命令変換表の記述の一部

'r[eax] 、2 番目に 0x1 を渡している。

```
*32* r[esp] := 'r[esp] - 0x4
*32* m['r[esp]] := 0x1
*32* r[esp] := 'r[esp] - 0x4
*32* m['r[esp]] := 'r[eax]
*32* r[esp] := 'r[esp] - 0x4
-    call foo
*32* r[esp] := 'r[esp] + 0x8
```

ここで call foo に対して与えられる引数は、プロセッサ定義から以下の 2 つである。

```
'm['r[esp] + 4]
'm['r[esp] + 8]
```

したがって、call 命令直前にこれらを引数変数 argument に代入する文を並べ、プロセッサ定義から得られる返り値をもとに関数呼び出し文を生成する。その結果、以下の出力を得る。

```
*32* r[esp] := 'r[esp] - 0x4
*32* m['r[esp]] := 0x1
*32* r[esp] := 'r[esp] - 0x4
*32* m['r[esp]] := 'r[eax]
*32* r[esp] := 'r[esp] - 0x4
*32* argument0 := 'm['r[esp] + 4]
*32* argument1 := 'm['r[esp] + 8]
*32* r[eax] := foo(argument0,argument1)
```

```
*32* r[esp] := 'r[esp] + 0x8
```

このIRコードを逐次実行すると、それぞれの引数変数には以下の値が代入されている。

```
argument0  =  'r[eax]
argument1  =  0x1
```

したがって、この関数呼び出しの引数は正しく抽出されている。

4.4 calleeにおける引数復元

calleeにおける引数復元も、プロセッサ定義のPROCEDUREセクションの記述に従う。

例えば2つの引数を持つ関数*foo*の例を考える。IRコードにおいては、関数*foo*が呼ばれると、引数変数argに各引数が格納される。順番に、arg1,arg2...となる。ここで、この引数変数に対してアクセスする際に、ABIで規定された方法に則るとすれば、各変数引数はそれに対応するarg_specで表される変数に代入される必要がある。したがって、calleeにおける引数の復元プロセスはそのようになされる。

*foo*が呼ばれた直後には各引数は引数変数arg1,arg2に格納されている。したがって、例えばx86系における引数は以下のように復元される。

```
function foo (4,4):
  *32* m['r[esp] + 4] := arg1
  *32* m['r[esp] + 8] := arg2
  ...
```

4.4.1 関数呼び出し復元を行わないケース

関数呼び出しの復元を行わないケースがある。その条件は、同じ関数のcaller及びcalleeがIRコード中にしか現れない場合、である。例えば標準ライブラリ関数はcalleeがIRコードに現れないため、復元を行なわないケースに該当しない。またmain関数は、callerがIRコードに現れないため、これも該当しない。

caller及びcalleeがIRコード中にしか現れない場合、この関数呼び出しはABI規則に則っているという保証が無い。したがって、ABI規則から導出したプロセッサ定義を適用することはかえって難しい場合がある。また、このような場合には引数の個数を既知とすることができないため、関数呼び出し復元は事実上不可能である。

そこで、このような場合にはスタックフレームの動作を正しく再現することで、関数呼び出しを正常に機能させることになる。そのために、命令変換表をスタックフレームの動作を意識して記述する必要がある。

5. Code Generatorの変換プロセス

Code Generatorは、IRコードを高級言語へ変換する。本研究ではIRの設計及びその生成方法に重きを置いたため、Code Generatorの実装はIRから実際に$bc(M_T, OS_T)$を生成できるかどうかを確認することのみを目的とした。したがって、特に最適化等を行なっていない。変換先の高級言語にはC言語を採用した。これは以下のような理由からである。

- IRと同じ手続き型言語である
- ポインタによるメモリ操作が可能
- コンパイラが最も普及している

5.1 メモリマネージャ

IRコードでのメモリ領域へのアクセスはメモリ変数を用いて行われる。例えば0x10a90番地にアクセスする場合IRコードでは単にm[0x10a90]といった表記となる。これはIRメモリ空間へのアクセスを意味する。IRのアドレス空間はM_Sと同一であるが、M_Tとは異なる。また、M_SとM_Tとでバイトオーダが異なる可能性もある。このようなIRにおける絶対番地参照の解決を行なうため、Code Genaratorは$bc(M_T, OS_T)$にメモリ管理を行なうための機構として**メモリマネージャ**というランタイムコードを付加する。これはIRメモリ空間へのストア及びロードを行なう関数群である。

メモリマネージャは以下の3つの関数からなる。また、IRメモリ空間は変換されたC言語ソースにおいては、IRメモリ配列としてchar型で宣言される。

- `int *GetMemoryAddress(int addr)`
  addrで指定されたIRメモリ空間のアドレス(IRMA)から、M_Tのメモリ空間へのポインタ(EMA)を得る。
- `int GetMemoryValue(int bit , int addr)`
  addrで指定されたIRMAが表すIRメモリ領域からビット数bitの値を得る。IRメモリ配列変数はchar型であるのでこの関数を用いて32bitの値を格納する場合、IRMAに対応するEMAから始まるメモリ領域から4バイトが1バイトずつ読み込まれ、それをひとつの32bitの値に直して返す。
- `void SetMemoryValue(int bit , int value , int addr)`
  addrで指定されたIRMAが表すメモリブロック配列変数にビット数bitの値valueを格納する。valueは8ビット毎に分割され、IRMAに対応するEMAから始まるメモリ領域に1バイトずつ格納される。

実際の変換プロセスは以下の通りである。

表 1 実装したプログラムの行数

Object File Parser	331 行
IR Generator	485 行
Code Generator	396 行

5.1.1 メモリから値の取得

メモリ変数'm[*addr*] は、そのまま単純に GetMemoryValue 関数への呼び出しに置き換えることができる。例えば以下のような変換が行われる。

```
  *32* r[eax] := 'm[0x1234]
→    eax = GetMemoryValue(32,0x1234);
```

メモリからの値の取得の際にはその IR メモリ空間のバイトオーダによって、読み取り開始位置、読み取り方向が変わる。メモリマネージャでは Big endian 用及び Little endian 用の GetMemoryValue 関数がそれぞれ用意されており、Code Generator の変換プロセスにおいては、バイトオーダ情報に従って、必要な GetMemoryValue 関数を C 言語ソースにインポートする。

5.1.2 メモリへの値の格納

メモリブロックへの値の格納は、SetMemoryValue 関数によって行われる。メモリへの代入文は以下のように変換される。

```
  *32* m[0x1234] := <IR-expression>
→    SetMemoryValue(32,<C-expression>,0x1234);
```

メモリへの値の格納に際してもまた、GetMemoryValue と同様にバイトオーダ毎にそれぞれの関数が用意されており、変換プロセスにおいては、バイトオーダ情報に従って、必要な SetMemoryValue 関数を C 言語ソースにインポートする。

6. 評　　価

6.1 実　　装

フレームワークの実装にはすべて Perl を用いた。各部分の行数は表 1 の通りである。

Object File Parser の実装に当たっては、GNU の binutils に含まれる objdump を利用した。これを用いて $\mathsf{bc}(\mathsf{M_S}, \mathsf{OS_S})$ をディスアセンブルし、シンボルテーブルを取得する。その結果を $\mathsf{asm}(\mathsf{M_S})$ に整形するために Perl スクリプトを用いた。今回の実装では ELF32 バイナリ形式にのみ対応している。

6.2 実　　験

6.2.1 動作の確認

表 4 に示した 2 つのマシンを用い、それぞれをソースマシン、ターゲットマシンとして、ソースマシン上で動作するバイナリコード $\mathsf{bc}(\mathsf{M_S}, \mathsf{OS_S})$ を、実装したフレームワークを用いて変換したバイナリコード $\mathsf{bc}(\mathsf{M_T}, \mathsf{OS_T})$ がターゲットマシン上で正しく動作するかどうかを実験によって確認した。ターゲットアーキテクチャ変更操作は、マシン A 及びマシン B 間で双方向に行なった。

実験に用いたプログラム 5 つで、それぞれ以下の項目が正しく動作するかどうかを確認するためのものである。

(1) 基本的手続き構造
　アッカーマン関数計算プログラム及びフィボナッチ数列計算プログラムを用いる。

(2) 構造体を含む手続き構造
　構造体を含む手続き呼び出しを行なうプログラムを用いる。

(3) 入出力
　ファイルの入出力を行なうプログラムを用いる。これは、コマンドライン引数としてファイル名を受け取り、そのファイルの先頭 10 文字を標準出力に表示するものである。

(4) 多条件分岐
　Switch 構文を用いた多条件分岐を含むプログラムを用いる。これは標準入力から受け取った値によって異なる文字列を表示するものである。

これら全てのプログラムについて生成された $\mathsf{bc}(\mathsf{M_T}, \mathsf{OS_T})$ をターゲットマシン上で動作させ、その動作が $\mathsf{bc}(\mathsf{M_S}, \mathsf{OS_S})$ と同じであることを確認した。

6.2.2 サイズと実行時間

生成された $\mathsf{bc}(\mathsf{M_T}, \mathsf{OS_T})$ のサイズと動作速度を検証するために、$\mathsf{bc}(\mathsf{M_S}, \mathsf{OS_S})$ と、ソースマシンとターゲットマシンを同一のマシンとした $\mathsf{bc}(\mathsf{M_T}, \mathsf{OS_T})$ のサイズと動作速度を計測した。

使用したプログラムはアッカーマン関数計算プログラム及びフィボナッチ数列表示プログラムで、これを gcc2.95 を用いてコンパイル (最適化無し) したものを $\mathsf{bc}(\mathsf{M_S}, \mathsf{OS_S})$、そのバイナリコードに対してターゲットアーキテクチャ変更操作を施し、Code Generator の生成した C 言語ソースを gcc2.95 を用いてコンパイル (-O4 最適化) したものを $\mathsf{bc}(\mathsf{M_T}, \mathsf{OS_T})$ とする。

アッカーマン関数に関しては ackermann(3,9) を 10 回計算したときの実行時間、フィボナッチ数列に関しては Fibo(1)〜Fibo(40) を計算したときの実行時間である。(表 2、表 3)。ただし、サイズ比及び実行時間比とは、$\mathsf{bc}(\mathsf{M_S}, \mathsf{OS_S})$ の実行時間を 1 としたときの $\mathsf{bc}(\mathsf{M_T}, \mathsf{OS_T})$ のサイズ及び実行時間である。

この表から、プログラム及びアーキテクチャに依らず $\mathsf{bc}(\mathsf{M_T}, \mathsf{OS_T})$ は $\mathsf{bc}(\mathsf{M_S}, \mathsf{OS_S})$ の約 3 倍のサイズ、約 5.5 倍の実行時間となることがわかる。

表 2 アッカーマン関数プログラムの実行時間

	$bc(M_S, OS_S)$		$bc(M_T, OS_T)$		比較	
	サイズ	時間	サイズ	時間	サイズ	時間
マシン A	5016 bytes	15.10 sec	17468 bytes	82.75 sec	3.48	5.50
マシン B	7036 bytes	32.21 sec	22372 bytes	175.54 sec	3.18	5.48

表 3 フィボナッチ数列プログラムの実行時間

	$bc(M_S, OS_S)$		$bc(M_T, OS_T)$		比較	
	サイズ	時間	サイズ	時間	サイズ	時間
マシン A	4883 bytes	30.40 sec	13996 bytes	167.14 sec	2.87	5.49
マシン B	6916 bytes	56.07 sec	20104 bytes	306.14 sec	2.91	5.45

表 4 実験に用いた環境

	プロセッサ	OS	バイナリ形式
マシン A	PentiumII 400MHz (Dual)	Debian Linux	ELF32
マシン B	UltraSPARC	Solaris	ELF32

本実装では、IR におけるメモリアクセスが全てメモリマネージャへの呼び出しに変換されているため、バイナリのサイズの増大と速度のオーバヘッドが起きる。ただし、実行時のメモリアクセスの回数は $bc(M_S, OS_S)$ と $bc(M_T, OS_T)$ で同一であるので、プログラム、マシンアーキテクチャによらずサイズ及び実行時間の増加は定数倍に収まると考えられ、実験の結果それが示された。

7. ま と め

本研究では、ソースマシン上で動作可能なバイナリコードと、ソース及びターゲットマシンアーキテクチャの仕様を入力として与えることで、ターゲットマシン上で動作可能なバイナリコードを得ることを目的とした、異なるマシン間での自動的なターゲットアーキテクチャ変更を実現するフレームワークを提案した。

このフレームワークの実現方法と、マシンアーキテクチャの仕様の記述方法を規定し、それに基づく実装を行った。また、その動作を検証した。

今後の課題として以下のものが考えられる。

- IR コードの最適化
  IR コードは命令変換表にしたがって機械語を IR 文に置き換えたものである。そこには実行に不要な文が多く含まれており、データフロー解析を行なうことで IR コードを最適化する。
- 実行コードの直接生成
  本実装では IR から変換された C 言語ソースはメモリ参照を常にメモリマネージャを通して行なうため、実行効率が著しく低下している。そこで、IR コードから直接実行コードを生成する Code Generator を設計/実装し、これを改善する。
- 複数のバイナリフォーマットへの対応
  本稿では Object File Parser の対応バイナリフォーマットとして ELF32 のみを想定している。この対応バイナリフォーマットを柔軟に拡張するための定義の記述方法及びその実現方法を考察し、実装する必要がある。

参 考 文 献

1) R.J.Hookway and M.A.Herdeg: Digital FX!32: Combining emulation and binary translation., *Digital Technical Journal 9(1):3-12* (1997).
2) T.Thompson: An Alpha in PC clothing., *Byte p195-196* (1996).
3) Cifuentes, C., Emmerik, M. V. and Ramsey, N.: The Design of a Resourceable and Retargetable Binary Translator, Master's thesis, Univerity of Queensland and University of Virginia (1999).
4) Ramsey, N. and Fernandez, M.: The New Jersey Machine-Code Toolkit, *1995 USENIX Technical Conference. January 16-20 1995 pp.289-302* (1995).
5) Ramsey, N. and Fernandez, M.: Specifying Representations of Machine Instructions, *ACM Transactions on Programming Languages and Systems Vol.19 No.3 pp.492-524* (1997).
6) CORPORATION, I.: Intel Architecture Software Developer's Manual, Technical report, INTEL CORPORATION (1997-99).
7) Weaver, D. L. and Editors, T. G.: The SPARC Architecture Manual Version 9, Technical report, SPARC International Inc. (1994).
8) B.W. カーニハン, D.M. リッチー著 石田晴久訳: プログラミング言語 C 第 2 版, 共立出版株式会社 (1989).
9) Cliffs, E.: *Unix System V: Application Binary Interface.*, Prentice Hall (1990).