Ruby 用仮想マシン YARV の実装と評価

笹田 耕一 [†]　まつもと ゆきひろ [††]
前田　敦司 [†††]　並木　美太郎 [†]

本発表ではオブジェクト指向スクリプト言語 Ruby を高速に実行するための処理系である *YARV: Yet Another RubyVM* の実装と，これを評価した結果について述べる．Ruby はその利用のしやすさから世界的に広く利用されている．しかし，現在の Ruby 処理系の実装は単純な構文木をたどるインタプリタであるため，その実行速度は遅い．これを解決するためにいくつかのバイトコード実行型仮想マシンが提案・開発されているが，Ruby のサブセットしか実行できない，実行速度が十分ではないなどの問題があった．この問題を解決するため，発表者は Ruby プログラムを高速に実行するための処理系である *YARV* を開発している．YARV はスタックマシンとして実装し，効率よく実行させるための各種最適化手法を適用する．実装を効率的に行うため，比較的簡単な VM 生成系を作成した．発表では，Ruby の，処理系実装者から見た特徴を述べ，これを実装するための各種工夫，自動生成による実装方法について述べる．また，これらの最適化のための工夫がそれぞれどの程度の高速化に寄与したかについて評価する．

YARV: Yet Another RubyVM
The Implementation and Evaluation

Koichi Sasada [,†] Yukihiro Matsumoto [,††] Atsushi Maeda [†††]
and Mitaro Namiki [†]

In this presentation, we describe the implementation and evaluation results of YARV, next generation Ruby implementation. The Ruby language is used worldwide because of its ease of use. However, current interpreter is slow due to its evaluation method. To solve this problem, several virtual machine designs were proposed, but none of them exhibited adequate performance/functionality combination. Our implementation, called *YARV (Yet Another Ruby VM)*, is based on a stack machine architecture. YARV incorporates a number of optimization techniques for high speed execution of ruby programs. In this presentation, we describe the characteristics of Ruby from implementor's point of view, and present concrete solutions for these issues as well as implementation of optimization techniques. We also show how each of these optimizations contributed to the speed-up.

1. はじめに

スクリプト言語 Ruby[21)~23)] は、手軽にオブジェクト指向プログラミングを実現するための種々の機能を持つプログラム言語である．

Ruby はプログラミング言語として次のような特長がある[23)]．

- シンプルな文法
- 一般的なオブジェクト指向機能（クラス，メソッドコールなど）
- その他のオブジェクト指向機能（Mixin，特異メソッドなど）
- 演算子オーバーロード
- 例外処理機能
- ブロック付きメソッド呼び出しとクロージャ
- ガーベージコレクタ
- ダイナミックローディング
- 移植性の高さ

ほかにも，多くの優れたライブラリが利用できることもあげられる．これらの特長により，簡単に，楽しくプログラミングを行うことができる．[26)] これらの特長をもつ Ruby は，世界中で広く利用されており，多くのユーザを擁すプログラミング言語となっている．

しかし，現在の Ruby 処理系は，速度的な問題を抱

† 東京農工大学大学院工学教育部
Graduate School of Technology, Tokyo University of Agriculture and Technology
†† (株) ネットワーク応用通信研究所
Network Applied Communication Laboratory Co.,Ltd.
††† 筑波大学大学院システム情報工学研究科
Graduate School of Systems and Infomation Engineering, University of Tsukuba

えている[9]．この原因の一つは，現在の Ruby 処理系は Ruby プログラムをパースした結果生成される構文木を実行時にたどり実行する方式にある．現在の Ruby 処理系は評価器を再帰関数として実装しており、木をたどりながら葉をそれぞれ評価し実行していく．本方式は実装が単純になるという利点があるが，次に示す理由で速度的な問題点ある．

- 構文木をたどるためのオーバヘッドがかかる
- 仮想マシン向けに知られている最適化の適用が困難
- 例外処理の高いオーバヘッド

そこで，発表者は構文木をたどるのではなく，構文木を命令列に変換し，その命令列を解釈実行する仮想マシン（VM: Virtual Machine）である *YARV: Yet Another RubyVM*[18]（以下 YARV）を開発している．YARV は Ruby プログラムを高速に実行することを目的とした仮想マシンで，Ruby プログラムを新たに設計した命令セットにコンパイルし実行する．YARV は スタックマシンアーキテクチャで実装しており，既知の各種最適化を適用した．仮想マシン自体は単純な VM 生成系を利用することで少ない記述で VM の基本部分の構築や各種最適化を適用した．

そこで，本稿では Ruby という実用的な言語の処理系を開発した経験から，その処理系の設計および実装の詳細，その開発にあたり遭遇した問題と，その解決策・種々の最適化技法の実行速度への寄与について得られた知見を示す．

以下，2 章で Ruby 処理系開発における課題を述べ，3 章で YARV をどのように実装したかについて述べる．4 章で各種最適化手法について述べ，5 章でそれを可能にする VM 生成系について述べる．7 章で開発した処理系の最適化の結果，性能向上がどの程度であったのかベンチマークプログラムによって評価を示す．最後に 8 章で関連研究を示し，9 章でまとめ，今後の課題を示す．

2. Ruby 処理系実装の課題

本章では Ruby の言語としての特質を言語処理系実装者の視点から概観し，その実装上の課題をまとめる．

2.1 パーサ

Ruby の文法は，利用者にとって自然に感じるようデザインされているが，その反面プログラムによりパースすることが難しい．たとえば，メソッド呼び出しには必要なければ括弧を省略することができるが，形式的な文法として，単純に BNF を記述できるものではない[28]．

2.2 実行系

Ruby プログラムを実行するにあたり，とくに処理速度向の課題について述べる．

2.2.1 オブジェクト指向機能

Ruby はクラスベースのオブジェクト指向プログラミングを実現する．たとえば，基礎となるメソッド呼び出しは，`recv.method(args)` のように記述されレシーバ `recv` のクラスに定義されている `method` というメソッドを，`args` という引数で評価する．

これを実行するためには，`recv` オブジェクトのクラスについて，`method` メソッドを表現する実体を検索する必要がある．Ruby プログラムの大部分はメソッド呼び出しを行うことであるため，この検索を毎回行うのはオーバヘッドが大きい．

2.2.2 オブジェクトモデルとガーベージコレクション

Ruby のプログラムで用いられる値はすべてオブジェクトである．たとえば，プログラミング言語 Java[19] では，整数型のようなプリミティブ型を用意しているが，Ruby ではすべてオブジェクトとして表現される．そのため，整数型同士の加算 `x + y` なども `x.+(y)` というメソッド呼び出しと等価になり，再定義などが可能である．

また，Ruby はガーベージコレクションを標準として備えているため，このオブジェクト管理機能の性能は，処理速度に影響することになる．

2.2.3 ブロックつきメソッド呼び出しとクロージャ

Ruby の特長のひとつとして，ブロックつきメソッド呼び出しが挙げられる．これは，メソッド呼び出しに手続きをブロックとして渡す機能である．プログラム言語 Scheme でいえば `lambda` 式によってクロージャを生成し，関数に引数として渡す処理にあたる．ブロックを受け取ったメソッドは，`yield` 文によりそのブロックを評価することができる．

また，クロージャを表現する Proc クラスのオブジェクトを明示的に生成することも可能であり Proc オブジェクトをブロックとしても利用可能である．繰り返しの各イテレーションごとにこのブロックが評価されるため，ブロック（クロージャ）の起動はできるだけ速いことが望ましい．

2.2.4 動的な実行モデル

Ruby では多くのことが動的に決定されるため Ruby プログラムの静的な解析は困難である．たとえば変数にはクラスを静的に指定することはないことや，クラスの定義，メソッドの定義が実行時に行われ☆，また実行時の再定義などが可能（**図 1**）な点などである．

また，文字列を Ruby プログラムとして実行時に評価する `eval` メソッドがあるため，ある特定のメソッドが再定義されないという保証を得ることが不可能である．このため，コンパイル時の解析が非常に困難になっている．たとえば，数値リテラル同士の演算が再

☆ Ruby ではクラス定義文自体も実行文であり，クラス定義文中に任意の Ruby プログラムを記述することができる．

```
class C
  if cond1 then
    # m1 cond1 によっては定義されない
    def m1()
    end
  end
end

# ...

class C
  def m1() # m1 の再定義
  end
end
```

図 1 動的な評価を利用した例

```
begin
  begin
    raise "raise exception!"
  ensure
    ...# この節は例外があってもなくても
       # 必ず実行される（Java の finally）
  end
rescue
  ... # この節で例外をキャッチする
end
```

図 2 Ruby の例外処理機能を用いたプログラムの例

定義されないという保証がないため，多くのコンパイラが行う定数畳み込みやループ普遍式の除去などの処理は，現在の Ruby の文法を忠実に堅持する限り難しい．

2.2.5 例外処理

Ruby は例外処理機能を持ち，たとえば**図 2** のように記述することができる．

現在の Ruby 処理系は構文木を評価する関数を再帰呼び出しすることによって Ruby プログラムを実行している関係上，その例外処理部分（図 2 における `begin`）で毎回 `setjmp` 関数によりコンテキストを例外ハンドラとして保存し，例外の発生時（図 2 での `raise`）では `longjmp` を実行してマシンスタックの巻き戻しを行う手法により実現している☆．

この方式では，例外発生時のキャッチ部分へのジャンプは軽量に行うことができるが，例外処理部分に突入するごとに `setjmp` による例外ハンドラの登録が必要になる．一般的に，例外が発生することは稀であるため，この手法は効率が悪い．例外処理部分への突入には余計なコストがかからないことが望ましい．

2.2.6 Ruby C API

Ruby の特長のひとつに，C 言語用 API が充実しており，C 言語などで作成する Ruby を拡張するためのライブラリ（拡張ライブラリ）を容易に記述が可能ということがある．たとえば Ruby で定義したメソッドを C 言語から呼び出すには，C 言語の関数呼び出しと同様のインターフェースが自然である．また，Ruby プログラムで記述する例外処理なども，C 言語のインターフェースで自然に表現できることが望ましい．現在の処理系では Ruby C API を用意してこれらの要件をサポートしている．Ruby でのこれらを可能にした拡張ライブラリの書きやすさは定評がある．

これらの機能は，Ruby プログラムの評価自体が C の関数の再帰呼び出しとして実装しているため容易に実現できている．しかし，命令列評価関数の再帰呼び出しは，とくに例外処理と絡む場合困難である．

2.2.7 スレッドの対応

Ruby はスレッドの生成・実行をサポートしている．スレッドを実現するにはいくつか方式があるが，たとえば複数 CPU 資源を利用する場合には OS の提供するスレッド機能を使う必要がある．現在の Ruby 処理系は独自でユーザレベルスレッドを提供しているため，ひとつの Ruby 処理系上ではアプリケーションの並列実行ができない．

2.3 Ruby 処理系開発における課題

本節で述べた Ruby 処理系開発における課題，最適化を困難にする要素をまとめると次のようになる．

- パーサ作成が困難
- オブジェクト指向機能の実現
- プリミティブ型がない
- ブロック・クロージャの実現
- 静的解析が困難
- 例外処理機能の実現
- Ruby C API のサポート
- スレッドのサポート

とくに，静的解析が困難であるということから，コンパイル時の最適化ができないため実行時最適化に負うところが大きい．

3. YARV: Yet Another RubyVM

本章では，前章で述べた課題のもとに開発した Ruby 処理系の YARV について述べる．

3.1 YARV の全体像

YARV は（1）構文木を YARV 命令セットに変換するコンパイラ（2）命令列を実行する命令列評価器からなる．とくに，(2）についてはプログラムの実行時間に直結する部分であるため，最適化機構を多く盛り込んだ．

YARV の実行モデルはシンプルなスタックマシンとし，YARV 命令列もそのように設計した．

YARV では上記（1）(2）以外の部分を現在の Ruby 処理系のものを多く流用した．具体的には Ruby プログラムのパーサやオブジェクトモデルの管理（とくにガーベージコレクタ）などである．これにより，既存の Ruby 処理系用に開発された拡張ライブラリなどが

☆ 例外処理以外にも，ブロックの実行を中断する `break` などもこれを利用して実現している．

表 1 命令セットのカテゴリ一覧

変数関係	ローカル変数などの値を取得・設定（getlocal，setlocal など）
値関係	self の値や文字列・配列などを生成（putself，putstring など）
スタック操作関係	スタック上の値操作（pop，dup など）
メソッド定義	メソッドを定義（methoddef など）
クラス定義関係	クラス・モジュール定義スコープに入る（classdef など）
メソッド呼び出し関係	メソッド呼び出しや yield など（send，end など）
例外関係	例外を実装するために利用（throw）
ジャンプ関係	ジャンプ・条件分岐（jump，if，unless）

そのまま利用できる．

3.2 命令セット

YARV では，Ruby プログラムを正しく表現するための基本命令セット（**表 1**）を定義した．執筆時現在，基本命令は 52 命令である．また，このほかに後述する最適化用命令も定義する．

たとえば Ruby プログラム `a=recv.method(b)` は，次のようにコンパイルされる．

```
getlocal 2       # ローカル変数 recv を push
getlocal 3       # ローカル変数 b を push
send :method, 1 # 引数 1 でメソッド呼び出し
setlocal 1       # ローカル変数 a に pop & set
```

Ruby にはプリミティブ型のようなものがないので，数値の演算命令などは用意しない☆．また，メソッド定義，クラス定義は実行時に行う必要があるので，命令として用意している．

3.3 コンパイラ

コンパイルでは既存の Ruby 処理系のパーサを流用し，Ruby プログラムを抽象構文木へ変換する．YARV はこの抽象構文木を YARV 命令による命令列に変換する．

3.4 命令列評価器

命令列評価器はコンパイルされた命令列を実行する部分であり，YARV の中心的な機能である．命令列評価器はひとつの C 言語による関数によって実現しており，以降これを VM 関数と呼ぶ．

YARV はスタックマシンとして構成され，処理速度を高速化するため，すでに提案されているさまざまな最適化手法[5] を適用している．VM は以下の VM レジスタを持つ仮想マシンである．

PC プログラムカウンタ
SP スタックポインタ
CFP フレーム制御ポインタ
LFP メソッドローカル環境ポインタ
DFP ブロックローカル環境ポインタ

PC は現在実行中の命令の位置．SP は，各スレッドがそれぞれひとつ持つ VM スタックのトップを指す．CFP は，現在実行中のスタックフレームを指す．

LFP，DFP は現在実行中の環境を指し，それぞれメソッドローカル変数を格納する環境，ブロックローカル変数を格納する環境へのポインタである．後者は，より上位の環境へのポインタをたどる方法を用意し，最終的にはメソッドローカル環境をたどることができる．多くの Lisp 処理系のように環境へのポインタはひとつしか用意しない選択肢（この場合 DFP を利用すれば LFP の指す環境を見つけることができるため，DFP のみにすること）もあるが，メソッドローカル変数を多く参照するという Ruby の特質からそれぞれ用意した．たとえば，ブロックの深い（環境が深くネストしている）ところでメソッドローカル変数を参照する場合，LFP を利用すれば定数時間でアクセスすることができる．

メソッド呼び出しなどを行ったときには新しいスタックフレームを生成する．クロージャを生成する場合，スタックに格納されていた環境をヒープ上にコピーし，LFP，DFP を適切に書き換える．

3.4.1 例外処理

YARV では各スコープ（命令列）ごとに例外処理用の表をコンパイル時に用意することで実現した．表には具体的にはプログラムカウンタのある範囲で例外が発生した場合，どのような挙動を行うかを示す．この方式は Java 仮想マシン[24] などとほぼ同様で，例外が発生したとき，この表を参照することで例外処理をどのように行うべきかどうかを知ることができる．また，Ruby の大域ジャンプする式（retry など）もこの機構を利用する．

表を用いる方法は，例外発生時には VM スタックを巻き戻し，キャッチする部分が見つかるまで VM スタックフレームごとにこの表を検査するため，例外発生時に longjmp によって例外ハンドラへジャンプする現在の Ruby 処理系よりも不利になるが，例外が発生しない場合は実行コストがかからないという大きな利点がある．殆どの例外処理部分では例外は発生しないため，本方式のほうが有利である．

ただし，Ruby では例外処理構文なども式となり，値を持つことが可能な点や，エラーを受け取る方法の違いから，単純に Java 仮想マシンなどと同様の仕組みを利用することはできず，表の構成や例外ハンドラの扱いが異なる．たとえば，例外処理から復帰した場合にスタックポインタをどの位置に設定するかなどの情報が表に含まれている．

C で記述する Ruby の拡張ライブラリは，C 言語レベルで Ruby の例外を発生することができる．これに対応するためには YARV 命令列レベルのみに対応する表引き方式では不十分である．そのため，YARV は setjmp，longjmp による例外処理機構にも対応する．つまり，YARV では両方式を併用して例外処理機構を構築している．Ruby レベルでの例外発生時には表引

☆ この代わりに後述する最適化のための特化命令を用意する．

き，C レベルでの例外発生時には longjmp を行う．

VM 関数の開始時に setjmp を行い Ruby C API のための例外ハンドラ（H）を登録しておく．拡張ライブラリ実行中に Ruby C API によって例外を起こした場合，longjmp によって（H）へ処理を移す．(H) では表引きにより例外処理を行う．その VM 関数中で例外処理が終了せず，例外をより上位に伝播する必要がある場合，longjmp によって上位の例外ハンドラに例外を伝播させる．この機構により VM 関数の入れ子を自然に実現できる．

4. 実行速度向上の工夫

本章では YARV に適用した実行速度向上のための工夫，最適化手法について述べる．

4.1 コンパイルと覗き穴最適化

抽象構文木からまず YARV 命令列をあらわすデータを双方向リストとして作成し，各種最適化，後述する命令変換を行い最終的に実行する命令列に変換する．その途中で冗長なジャンプ命令の除去など，いくつかの覗き穴最適化を行う．

コンパイル時に利用するメモリ領域は開発当初 GC 対象の Ruby オブジェクトとして各命令ごとに割り当てていたが，コンパイル対象が数万行の大きな Ruby プログラムの場合，コンパイル時に GC のオーバヘッドが非常に大きくなった．これを避けるため，コンパイル時には GC 対象のメモリ割り当てを行わないようにした．

4.2 スレッデッドコード

命令を逐次実行させる手法はいくつかあるが，GCC[8] ではラベルを値として用いることができるため，これを利用したダイレクトスレッデッドコード[2] により，命令ディスパッチのオーバヘッドを削減する．本手法により，単純に実行する命令数の削減以外にも，間接ジャンプを行うマシン命令番地が各命令ごとに異なる場所で分岐することになるため，分岐予測精度の向上が期待できる[4]．

GCC 以外のコンパイラでは C 言語 switch 文による命令ディスパッチを行う．

4.3 特化命令

Ruby にはプリミティブ型が無いため，すべての演算はメソッド呼び出しと等価であるが，たとえば整数加算のためにスタックフレームを新たに構築するのは無駄である．そのため，YARV では特定のセレクタ（メソッド名），特定の引数の数（二項演算子ならば引数の数は 1）の場合，コンパイル時，通常のメソッド呼び出し命令から特別な命令に置き換える．この特別な命令を特化命令と呼ぶ．

たとえば，Ruby の式 a+b は，a.+(b) というメソッド呼び出しと同様であるが，このとき通常のメソッド呼び出し命令ではなく opt_plus という命令にコンパイルする☆．

```
opt_plus:
  if(a と b は整数である)
    if(整数についてのメソッド + が
       再定義されていない)
      return a + b
  return 通常のメソッド呼び出し（a.+(b)）
```

図 3 特化命令 opt_plus の擬似コード

opt_plus の実行は，まずレシーバと引数（この場合は a と b）が整数であるかどうかを確認する．もしそうであれば今度は整数同士の加算のメソッドが再定義されていないか確認する．もしされていなければ，整数加算をした結果をスタックに積む．そうでない場合は，通常のメソッド起動処理に移行する（**図 3**）．

利用頻度が高く，メソッドの自体の実行コストに比べ，メソッド起動のコストが相対的に大きい特定のメソッドについては，このような工夫をすることでメソッドとしての一般性を失うことなく高速に実行することが可能になる．

現在の YARV では有限桁整数値（Fixnum）☆☆の四則演算用メソッド，および配列アクセス用のメソッドなどを置き換える 11 の特化命令を用意している．

4.4 オペランド・命令融合

ある命令において，特定の値の命令オペランドを頻繁に利用する場合，融合して新しい命令を作ることによってオペランドフェッチのコストを削減し，また部分評価により効率的な命令を作ることができ，処理速度が向上する．たとえば，命令 A がオペランド x を頻繁に利用する場合，命令 A をオペランド x に固定した A_x という新しい命令を作る．これをオペランドの融合という[29]．

具体的にはスタックに任意の値をプッシュする命令に対し，nil や true，false などの値は命令オペランドとして頻繁に指定されるので，それぞれ put_nil，put_true，put_false のようなオペランド融合した命令にすることが考えられる．

また，ある n 個の命令の並びが頻出する場合，それらを融合し新しい命令を作ることで，命令ディスパッチ，スタックの遷移やプログラムカウンタの操作のコストを削減することができる．たとえば，命令 C のあとで命令 D が現れるような場合が頻繁にある場合，C と D の機能を実行する C_D という命令を新しく作る[29]．これを命令の融合という[29]．

YARV ではオペランド融合，および命令融合により作成した命令を利用することで実行オーバヘッドの削減を図っている．

☆ a，b のクラスは実行時に判定するためコンパイル時に静的に解析する必要は無い．

☆☆ Ruby の Fixnum 整数クラスは桁あふれがおきると自動的に多倍長整数クラスに拡張される．そのため，図 3 での加算処理には実際にはオーバーフローのチェックも行う．

```
l1:
  # インラインキャッシュを見て，ヒット
  # ならばその値をプッシュし，l2 へジャンプ
  getinlinecache l2, vm_cnt, cached
  getconstant :Const       # 定数アクセス
  # l1 にある命令に値をキャッシュ
  setinlinecache l1
l2:
```

図 4 定数アクセスで利用するインラインキャッシュ命令

融合命令は VM 生成系により半自動的に作られる．これについての詳細は次章で述べる．

これら融合操作によるインタプリタの最適化は文献 29) でとくに述べられている最適化手法である．YARV ではこの融合命令の生成を VM 生成系により半自動的に行うことが可能であるため，少ない手間で最適化を行っていくことができる．

現状ではいくつかのベンチマークプログラムで顕著であった命令を融合しており，16 のオペランド融合，12 の命令融合を行っている．

将来的には，プロファイリング（利用統計と性能統計）をフィードバックとして，自動で融合操作を行う試行を繰り返しを行うことにより，あるプログラムセットにおける最適な命令セットの自動生成が可能ではないかと考えている．

4.5 インラインキャッシュ

メソッドディスパッチでは，レシーバオブジェクトのクラス，そしてその親クラスへとメソッドの実体が見つかるまで検索をする必要があるが，この検索をメソッドディスパッチのたびに行うのはオーバヘッドが大きすぎるため，現在の Ruby 処理系では，グローバルメソッドキャッシュ[22),28)] を用いてこのコストを軽減している．

YARV では，従来のグローバルメソッドキャッシュに加え，命令列に最後に検索したメソッドの情報を埋め込むインラインメソッドキャッシュを用意する．

また，Ruby 言語の定数の参照は特定の検索パスによって検索しなければならない負荷の大きい処理である[28)]．Ruby 言語の定数は頻繁に変更することはないため，毎回検索を行うのは無駄である．そこで，一度アクセスした Ruby 言語の定数はインラインキャッシュとして保存しておき，可能であれば次にこの命令を実行したときにキャッシュした値を定数値として返す（**図 4**）．

Ruby では再定義操作が可能であるため，再定義操作が行われたときにはインラインキャッシュのクリアを行わなければならない．そこで，YARV では仮想マシンに状態カウンタを設けてキャッシュの無効化を実現した．キャッシュしてある値が有効であるかは，キャッシュ時に一緒に格納する状態カウンタの値と，現在の仮想マシンの状態カウンタの値を比較して判断する．比較が一致すればキャッシュした時点以降で再定義操作が行われておらず，キャッシュした値が有効であることがわかる．

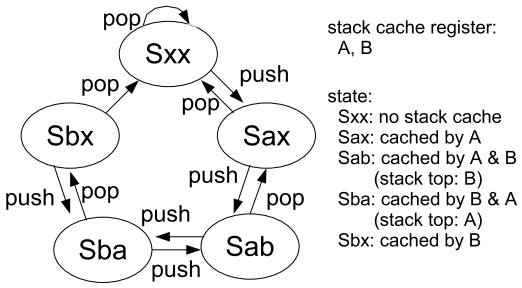

図 5 スタックキャッシングの状態遷移図

状態カウンタはメソッドや Ruby 言語の定数の定義・再定義が行われたときに 1 増加する．これらの操作頻度は基本的にまれであるため，インラインキャッシュには十分ヒットする．

インラインメソッドキャッシュにおいては，他にもキャッシュと一緒に格納しているクラス情報を確認し，これがレシーバのクラスと一致すれば，キャッシュヒットということになる．

YARV におけるインラインメソッドキャッシュついての詳細は文献 27) を参照されたい．

4.6 静的スタックキャッシング

スタックマシンの最適化にスタックキャッシング[3)]がある．スタックトップをいくつかキャッシュすることで，メモリへの書き込みやスタックポインタ操作などをある程度省略できる．本手法の詳細は文献 3) を参照されたい．

YARV では 2 個の VM キャッシュレジスタを利用して 5 状態（$S_{xx}, S_{ax}, S_{bx}, S_{ab}, S_{ba}$）で静的スタックキャッシングを行う．各状態の状態遷移を**図 5** に示す．これらの各状態で始まる命令を，VM 生成系により自動生成する．これについては次章で詳述する．

VM キャッシュレジスタがマシンレジスタに割り付けられるかどうかは，CPU やコンパイラによるが，この変数がマシンレジスタに割り付けられれば，単純な命令列ではスタック操作がすべてマシンレジスタ上で行われるため，高速な動作が可能になる．割り付けられない場合も，必要なスタックポインタの操作が減少し，オーバヘッド削減が期待できる．

状態遷移時，VM キャッシュレジスタの値をコピーすることを許せば，2 レジスタ 3 状態のスタックキャッシングとして実現することは可能だが，Intel x86 CPU のようなレジスタが少ないプロセッサの場合，マシンスタック上の値として割り付けられることが多く，スタックキャッシュ用レジスタ間の移動がメモリ間の移動になってしまい非効率となることがある．これを防ぐため，2 レジスタ 5 状態のスタックキャッシュとした．

4.7 プロファイラ

上記の最適化を効果的に行うため，YARV には実行時に次の情報を収集する機能を実装している．

- VM レジスタへのアクセス頻度
- 命令実行頻度
- オペランド出現頻度
- 命令の連結度

VM レジスタのアクセス頻度によってどのレジスタをマシンレジスタに割り付けるかなどを判断できる．命令の実行頻度を見ることで，その命令の重要度を知ることができ，どの程度最適化するべきかの指標を得ることができる．

オペランドの出現頻度はオペランド融合を行う指標とすることができる．命令の連結度は，ある命令の次にどの命令が何回実行かを示すバイグラムである．これにより，どの命令同士を連結すればよいかの参考にできる．

現在のプロファイラは情報集計のオーバヘッドが高いため，標準では本機能は無効にしてある．主に VM チューニングの指標を求めるために利用している．将来的には集計する情報を選別し軽量化することで，リアルタイムに更新される統計情報を利用した実行時最適化を行いたいと考えている．

4.8 ネイティブコンパイラ

仮想マシンの命令列をマシンコードに変換するネイティブコンパイラの実現手法にはいくつかあるが，実行時にコンパイルを行う JIT（Just-In-Time）コンパイラ，および実行前に行う AOT（Ahead-Of-Time）コンパイラに大別できる．

AOT コンパイラは，VM 生成系の応用として，YARV 命令列を C 言語プログラム列に置換し，YARV から利用する拡張ライブラリとしてコンパイルして実行する．つまり，Ruby プログラム → YARV 命令列 → C プログラム → Ruby 拡張ライブラリと変換する．この方式では Ruby の機能を損なうことなく，C コンパイラの最適化機能により高速に実行できる．

JIT コンパイラは，すべての命令のアセンブル記述を用意するには実装コストが問題になるため，Dynamic Superinstruction[15] と同様の手法で実現した．

ただし，両コンパイラともにいくつか機能が十分でないため，本稿では評価は行わない．

5. VM 生成系

本章では VM の構築，および前章で述べた最適化を容易に実現するための VM の生成系について詳細を示す．

実装した VM 生成系は基本的に文字列操作だけで実現しており，C 言語によって記述された処理部分の解析は行わず，単純である．この簡単な生成系により，いくつかの効果的な最適化が実現できることを示す．

なお，VM 生成系は主に命令実行するためのコードを主に生成し，本稿では主にこれについて述べるが，ほかにもコンパイラ，アセンブラ，逆アセンブラ，プロファイラなどを実現するための情報も生成する．

5.1 VM 記述言語と命令の生成

YARV の命令は VM 記述言語によってそれぞれ定義する．**図 6** の (1) に記述例を示す．例では `mult_plusConst` という命令の定義を示している☆．`mult_plusConst` 命令はスタックから 2 値取り出して掛け合わせ，命令オペランドで指定される値を足してスタックに値を格納する命令である．

図 6 の (1) では，`mult_plusCont` 命令のスタックから取り出す値（スタックオペランド）として `x`，`y` を `int` 型として宣言している．足し合わせる値は命令オペランドとして `int` 型の `c` という変数を宣言している．最後にスタックにプッシュする値，つまり命令の返り値を格納する変数として `int` 型の `ans` を宣言する．続くブロックで，宣言した変数を利用して，その命令が実際にどのような処理を行うか，C 言語で記述する．プログラムカウンタ（`PC`）やスタックポインタ（`SP`）の操作は記述する必要がない．

これらの変数は命令によって任意の個数宣言することができる．また，可変個の値が必要な場合，識別子 “...” を記述することで対応する．たとえば，スタック上に詰まれた任意の数の値を利用してオブジェクトを生成するような命令を作る場合はスタックオペランドに “...” を指定する．この値にアクセスする場合は，VM スタックを直接参照する．

これらの情報から VM 生成系では図 6 の (2) で示すような VM 関数のための C プログラム片を生成する．具体的にはプロローグコードとして命令オペランドの変数を定義し，命令列から値を取り出し初期化する部分，スタックオペランドを定義しスタックから値を取り出して初期化する部分，そして命令の返り値の値を格納する変数の定義を行い，`PC` と `SP` を必要なだけ増減するコードを挿入する．エピローグコードとして命令返り値をスタックに格納する処理，およびスレッデッドコードにより次命令のディスパッチを行うプログラムを生成する．

コンパイラは VM 記述で定義した命令を利用して命令列を生成する．

5.2 融合命令の生成

融合命令の作成は VM 生成系にどの命令オペランド，どの命令の並びを融合命令とするか指定することで行う．指定する命令オペランド，命令の並びは任意の数指定することができる．

オペランド融合の場合，命令名と融合する命令オペランドの組を指定する．もし 2 個以上命令オペランドがある命令で，その中の一部のみを融合する場合は，融合しない命令オペランド部分には “*” を指定する．**図 7** では `mult_plusConst` 命令に命令オペランド `c`

☆ `mult_plusConst` 命令は，理解のために定義している．YARV 自体にはこの命令は存在しない．

```
(1) VM 記述言語による命令記述
DEFINE_INSN
mult_plusConst
(int c)         // 命令オペランド定義
(int x, int y) // スタックオペランド定義
(int ans)       // 命令の返り値定義
{
  // C 言語による命令ロジック記述部分
  ans = x * y + c;
}

(2) 変換後の C 言語プログラム
mult_plusConst:
{
  int c = *(PC+1); // PC: Program Counter
  int y = *(SP-1); // SP: Stack Pointer
  int x = *(SP-2);
  int ans;
  PC += 2;
  SP -= 2;
{
  ans = x * y + c;
}
  *(SP) = ans; SP += 1;
  goto **PC; // スレッデッドコードの次命令へのジャンプ
}
```

図 6 VM 記述言語の記述例および生成された C 言語のプログラム

```
mult_plusConst_OperandUnified:
{
  #define c 5 // 融合した命令オペランド
  VALUE y = *(SP-1);
  VALUE x = *(SP-2);
  VALUE ans;
  PC += 2;
  SP -= 2;
{
  ans = x * y + c;
}
  #undef c
  *(SP) = ans; SP += 1;
  goto **PC;
}
```

図 7 生成されたオペランド融合命令

が 5 だった場合のオペランド融合命令として生成されたプログラムを示している．

生成されたプログラムでは融合する命令オペランドを単純にマクロで置換するように，`#define` マクロを挿入する．命令の挙動を記述した C プログラムは一切変更しない．

命令融合では，VM 生成系にどの命令列をひとつの命令に融合するか指定する．

図 8 の (2) では，(1) で示す `dup` 命令と `mult_plusConst` 命令を融合した場合に生成されるプログラムを示している．つまり，この命令はスタックから値をひとつ取り出し（x），$x \times x + c$ を計算する命令になる．

これを実現するため，命令オペランド，スタックオペランドの取得を融合命令の先頭で行う．各命令で命

```
(1) dup 命令の定義
DEFINE_INSN
dup
()
(int v)
(int v1, int v2)
{
  v1 = v2 = v;
}

(2) dup 命令と mult_plusConst の融合命令
UNIFIED_dup_mult_plusConst:
{
  int c_1 = *(PC+1);
  int v_0 = *(SC-1);
  int ans;
  PC += 3;
  SP -= 1;
{ // dup
  #define v v_0
  #define v1 v1_0
  #define v2 v2_0
  v1 = v2 = v;
  #undef v
  #undef v1
  #undef v2
}
{ // mult_plusConst
  #define c c_1
  #define x v1_0
  #define y v2_0
  ans = x * y + c;
  #undef c
  #undef x
  #undef y
}
  *(SP) = ans; SP += 1;
  goto **PC;
}
```

図 8 dup 命令の定義と命令融合した命令の生成

令の返り値があった場合，次の命令以降で利用するスタックオペランドとしてそれを利用する．各命令で利用する命令オペランド，スタックオペランド，命令の返り値を格納した変数は，変数名が衝突することを防ぐため，適切にマクロで置換する．

図 8 の (2) では，dup 命令の返り値 `v1`，`v2` の値を `mult_plusConst` 命令のスタックオペランドとして利用するように各変数名を置き換えている．

融合操作を C 言語のマクロを利用して適切な変数名に置換するよう実装したため，たとえば置換する変数名が構造体のメンバ名と衝突した場合，正しくコンパイルができない．これについてはいくつかの方針，たとえばマクロの代わりに新たに変数を宣言するなどを検討したが，生成されるコードが冗長になるためマクロのままで行うこととし，名前の衝突については名前規則によって回避することにした．

コンパイラで適切な融合命令に変換するため，VM

```
mult_plusConst_SC_ab_ax:
{
  int c = *(PC+1);
  int y = SC_regA; // SC 用レジスタ 1
  int x = SC_regB; // SC 用レジスタ 2
  int ans;
  PC += 2;
{
  ans = x * y + c;
}
  SC_regA = ans;
  goto **PC;
}
```

図 9 生成されたスタックキャッシング命令（$S_{ab} \rightarrow S_{ax}$）

生成系はオペランド融合命令のための命令変換プログラム，命令融合のためには連続する命令を認識するためのパターンマッチ用データをそれぞれ生成する．

5.3 静的スタックキャッシュ命令の生成

YARV では 4.6 節で示したとおり，2 レジスタ，5 状態（$S_{xx}, S_{ax}, S_{bx}, S_{ab}, S_{ba}$）の静的スタックキャッシングを実現する．このためには各命令ごとに，各状態から始まる命令を別々に定義しなければならない．つまり，ある命令 I について，$SC(I, S)$ を状態 S で開始 I 命令とすると，$SC(I, S_{xx})$，$SC(I, S_{ax})$，$SC(I, S_{bx})$，$SC(I, S_{ab})$，$SC(I, S_{ba})$ の 5 命令を新たに用意する必要がある．VM 生成系は I に基づき，この 5 命令を自動で生成する．

ここで，$SC_t(I, S_s) = S_e$ という関数を定義する．ある命令 I がスタックキャッシュの状態 S_s で実行した場合，命令終了後には状態が S_e になるという意味である．

$SC_t(I, S_s)$ は，命令 I のスタックオペランドの数 P，およびスタック返り値の数 Q により容易に決定可能である．つまり，図 5 で示した状態遷移図に従い，P 回だけ POP 操作を行った場合の状態から Q 回 PUSH 操作を行ったときにたどり着く状態が求める状態 S_e である．

実行コード生成時は，開始状態 S_s と終了状態 S_e に応じてスタックオペランドの取得部分と命令の返り値の格納部分をスタックキャッシュ用レジスタへのアクセスに置き換える．

たとえば，状態 S_{ab} で開始する `mult_plusConst` 命令は**図 9** に示す C プログラムに変換される．この場合，`SP` を介したスタック操作が一切ないため高速な命令実行が実現できる．

生成されたコードはスタックレジスタへのアクセスが冗長に見えるが，多くの C コンパイラはこのような冗長なアクセスを排除するため，最終的には最適なマシンコードが生成される．

コンパイラは命令列をスタックキャッシュ命令で置き換える．まず，i 番目の開始状態を S_i，命令を I_i，$S_0 = S_{xx}$ として，命令列 i 番目の開始状態 S_i は $S_i = SC_t(I_i, S_{i-1})$ $(i \geq 1)$ として求めることができる．このもとで I_i を $SC(I_i, S_i)$ に置換する．VM 生成系は $SC_t(I, S)$ を求めるための表を生成する．

もし I_i が j 番地へジャンプする命令だった場合，$i < j$ ならば I_j の状態を $S_r = SC_t(I_i, S_i)$ として予約しておく．このまま実行し，I_j の置換する際，予約された状態であるかどうかを確認する．$i > j$ ならば，S_j が S_r と等しいか確認する．もし違う場合，スタックキャッシュの状態の整合性を取るための命令を挿入する．

5.4 コード生成における VM コード量増加

VM 生成系は融合命令とスタックキャッシュ用命令を生成する．より正確には，基本命令から融合命令を生成し，それらに対してスタックキャッシュ用命令を生成する．基本命令数が B で融合命令数が U だった場合，$(B+U) \times 5$ 個の命令が生成されることになる．

また，融合命令についても，たとえば命令 I についてオペランド融合する命令が I_{op1}，I_{op2} とあったとき，命令融合の指示として II，つまり I を連続して実行する命令を指定した場合，オペランド融合した命令をあわせて命令融合の組み合わせを考えると 9 通りの命令が生成可能である．

このように，VM 生成系では簡単に命令数が増えてしまうため，実際にどの命令を生成するかという点で議論の余地がある．

命令数が増加すると，プログラムがプロセッサの命令キャッシュに乗りづらくなるという実行性能上の問題点と VM のコンパイル時間が著しく長くなってしまうという開発上の問題がある．

現在の YARV は基本命令，特化命令，融合命令あわせて 175 命令あり，これらに対しスタックキャッシング用命令を生成するため，総命令は 875 命令である．命令数増加による性能悪化よりも各最適化命令増加による性能向上のほうが目立っているため現在は命令数削減の工夫はしていないが，今後は文献 29) で提案されているようなシステマティックな命令選択．また，文献 6) ではスタックキャッシュのために生成する命令数を削減する手法などを取り入れていく必要があると思われる．

6. 実　　装

YARV の実装は基本的に C 言語で行った．VM 生成系は Ruby により記述した．

YARV は Intel の x86 プロセッサ，x86_64 プロセッサ，PowerPC 上で動作を確認している．OS は Linux 2.4，2.6 および Windows2000，XP での動作を確認している．

コンパイラは GCC3.3，3.4 を主に利用している．GCC 以外のコンパイラとしては Microsoft 社の VisualStudio 6.0 および 2003 付属のコンパイラで動作

(1)

```
# loop_times                    # loop_whileloop
30_000_000.times do |i|         i = 0
end                             while i<30_000_000; i+=1
                                end
```

(2)

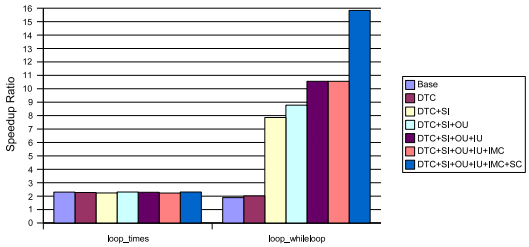

図 10 繰り返し実行速度の評価

している．

GCC3.3 以降のバージョンでは最適化オプション`-O`以上を指定した場合，crossjumping と呼ばれる最適化を行う．この最適化は関数中に出現する同一命令列をまとめ，コード量を節約する．VM 関数は，各命令ごとに多くの共通部分があるため，この最適化を有効にすると余計なジャンプ命令が挿入され，YARV 1 命令を実行するために必要な機械語命令数が増加する．また，スレッデッドコードのためのジャンプ命令が一箇所に集約してしまうため，分岐予測器で予測ミスが多発することになりスレッデッドコードの利点が半減してしまう．そのため，YARV の GCC によるビルドでは明示的にこの最適化を行わないように`-fno-crossjumping`オプションを指定した．

GCC の場合，変数に特定レジスタを占有するよう指示することが可能であるため，x86 プロセッサの場合はプログラムカウンタ（PC）を`esi`レジスタに割り当てた．また，x86 プロセッサに比べ利用可能なレジスタ数が増加した x86_64 プロセッサではプログラムカウンタに`r15`，スタックキャッシュ用レジスタに`r13`，`r14`を割り当てた．

ただし，`longjmp`による例外ハンドラへのジャンプに対応するため，PC 変更時にその値をメモリへ退避する必要があった．これは，例外ハンドラ検索のためには例外発生時の PC を知る必要があるが，`longjmp`を行うとマシンレジスタに格納している値が例外発生時の値ではなくなるためである．

この結果，ダイレクトスレッデッドコードを適用したときのもっとも単純な VM 命令`nop`（何もしない）は x86 機械語列で 4 命令となった．すなわち（1）PC を加算（2）ジャンプ先アドレスのロード（3）PC をメモリにストア（4）レジスタ間接ジャンプである．

7. 評　　価

本章では YARV の性能評価を行う．

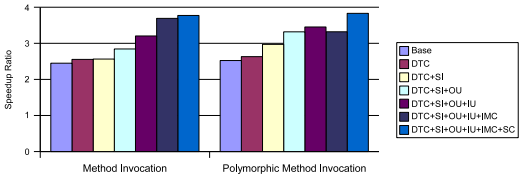

図 11 メソッド起動速度の評価

表 2 VM 基本機能の評価（各 3 千万回の試行）

	Ruby(sec)	YARV(sec)	Speedup
定数参照	10.52	0.91	15.28
例外発生しない			
ensure 節	1.48	0.00	-
rescue 節	4.16	0.44	32.99
例外発生する			
rescue 節	8.03	4.83	1.66

評価環境は Intel x86 CPU である Pentium-M 753（1.2GHz，L1 命令・データキャッシュそれぞれ 32KB，L2 キャッシュ2MB），メモリ 1GB，WindowsXP + cygwin，GCC 3.4.4 上である．また，7.2 節ではこれに加え，x86_64 CPU である AMD AMD64 3400+（2.2GHz，L1 命令・データキャッシュそれぞれ 64KB，L2 キャッシュ512KB），メモリ 1GB，Linux 2.6.8 FedoraCore 3（x86_64 版），GCC 3.3.3 の環境での評価結果も載せる．比較対象とする Ruby 処理系は ruby 1.9.0 (2005-03-04) を用いた．YARV のベースとなる Ruby 処理系も同じものである..

評価は評価対象プログラムの実行時間を 5 回計測し，中央値 3 つの平均値を計測結果とした．

各最適化は次の略語を利用する．

Base	命令実行型 VM
DTC	ダイレクトスレッデッドコード
SI	特化命令
OU	オペランド融合
IU	命令融合
IMC	インラインメソッドキャッシュ
SC	スタックキャッシング

たとえば，DTC+SI は，Base の VM 上に DTC（ダイレクトスレッデッドコード）と SI （特化命令）の最適化を施したことを示す．

とくに記述しない限り，速度向上率とは現在の Ruby 処理系の実行時間を YARV 処理系のそれで割ったものを言う．

7.1 各機能の評価

まず，**図 10** の（1）で示すような Ruby での一般的な繰り返しである`times`メソッドをブロック付きメソッド呼び出しを利用する方法と`while`文を用いた方法のプログラムを既存の Ruby 処理系と YARV で動作させ，実行時間を計測した．その結果の速度向上率を図 10 の（2）に示す．

`times`メソッドによる繰り返し（ブロック付きメソッ

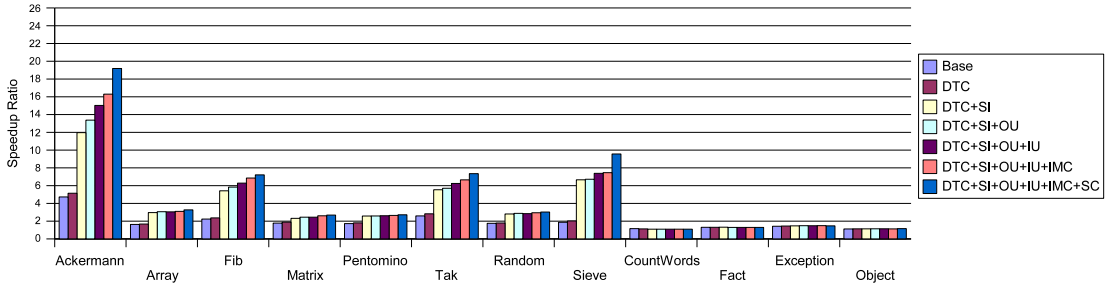

図 12 x86 プロセッサ上でのベンチマークプログラムの評価

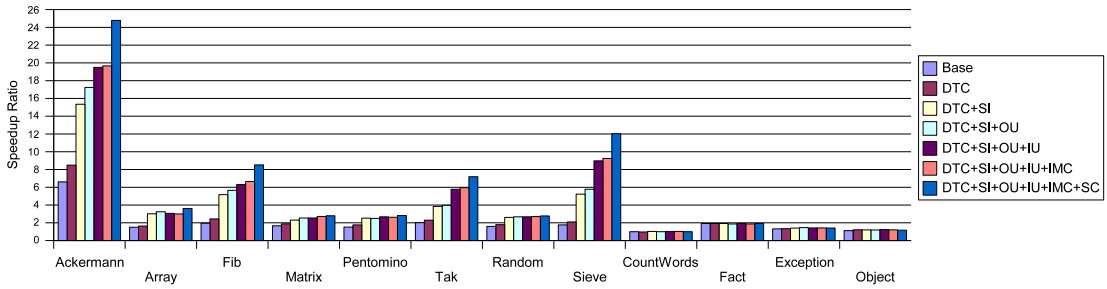

図 13 x86_64 プロセッサ上でのベンチマークプログラムの評価

ド呼び出し）は設計した VM 上では 2 倍ほど高速化したが，他の最適化がほとんど効いていない．これは，各繰り返しごとにスタックフレームを生成，破棄する必要があり，そのオーバヘッドが各最適化の効果よりも大きかったためと考えられる．

`while` 文による繰り返しは各最適化で非常に高速になった．とくに，数値の加算，比較などを特化命令で置き換えて実行し，メソッド呼び出しが一切なくなったため，大きく性能が向上している．実行する各命令が単純であるため，スタックキャッシングの効果も十分に確認できた．

次にメソッド呼び出しの性能を計測した（**図 11**）．左側は必ずインラインメソッドキャッシュがヒットする場合，右側は毎回ミスするようなプログラムになっている．左図を見るとインラインメソッドキャッシュによる性能向上が確認できる．しかし，毎回ミスすることでインラインメソッドキャッシュ検索がオーバヘッドになり，右図ではインラインメソッドキャッシュを有効にすると若干性能が低下した．

Ruby プログラムでは一般的にインラインメソッドキャッシュがヒットする場合が多い[27] ので本最適化は有効である．

その他の基本機能について計測した結果を**図 2** に示す．YARV はすべての最適化を有効にしてある．

インラインキャッシュにより Ruby 言語の定数探索コストを削減した結果，約 15 倍の性能向上が得られた．例外処理は表を用いる手法に変更したため，例外が発生しない限り，ほぼオーバヘッドなしで実行できた．例外が発生した場合も大きなオーバヘッドにならず，既存の処理系よりも高速化できることを確認した．

7.2 プログラムでの評価

いくつかのベンチマークプログラムを実行させ，計算時間を計測した結果を**図 12** に示す．また，x86_64 のプロセッサで動作させた結果を**図 13** に示す．

Ackermann や Tak，Sieve など，簡単な数値計算とメソッド呼び出しだけのプログラムの性能は VM の性能向上の影響を受け，速度向上率も大変高く，最大で 25 倍ほどの性能向上を示した．

Array や Pentomino，Random などは整数値だけでなく配列などのオブジェクトを頻繁にアクセスするため，ライブラリ関数のオーバヘッドが VM のオーバヘッドに比べ相対的に大きいため，速度向上率はあまり高くないが，それでも 2 倍以上の高速化は達成できた．

CountWords，Fact，Exception，Object はどれもあまり速度向上していない．CountWords は文字列操作，Fact は多倍長整数操作，Exception は例外オブジェクトの生成とアクセス，Object はオブジェクトの作成とアクセスが主なオーバヘッドであるため，つまり VM 以外の部分が処理時間の大部分を消費しており，VM の最適化による速度向上の効果がないためと考えられる．

ただし，このような VM のオーバヘッド以外の部分は C 言語で記述された拡張ライブラリ，たとえば

文字列に対する処理なら正規表現エンジンが主なオーバヘッドとなっており，この部分は現在の Ruby 処理系でも，他の環境と比較して大きな速度的な問題にはなっていない．

上述したとおり，Fact などのベンチマークは大半の実行時間が多倍長整数演算の計算時間であり，VM 部分の実行時間はほとんど無い．しかし，YARV にすることで 30%ほどではあるが，速度向上が確認できた．調査の結果，この速度向上の原因は現在の Ruby 処理系よりも YARV のほうがメソッド呼び出しのために利用するメモリ領域が少なく，mark & sweep 型の Ruby の GC 実行時にマークするべきルートになる部分が小さくなり，GC の実行時間が短縮されたからであることがわかった．

スタックキャッシング最適化による x86 プロセッサ上での速度向上率よりも，x86_64 プロセッサでの速度向上率が大きいのはマシンレジスタをスタックキャッシュ用レジスタとしているからである．

7.3 他言語環境との比較

いくつかのプログラムをその他のプログラミング言語で記述し，それぞれの処理系上で実行した結果と現在の Ruby，開発した YARV で実行した結果をあわせ，**図 14** にまとめた．Y 軸は実行時間（秒）を示す．

Perl[14] はもっとも多くのユーザを擁するスクリプト言語であり，その処理系は現在の Ruby 処理系と同様構文木をたどり実行するインタプリタである．Perl 6 からは Parrot[20] という命令実行型仮想レジスタマシンにする予定である．今回の評価は Perl 5.8.6 を利用した．

Python[16] は欧米で高い評価を得ているスクリプト言語である．その処理系は命令実行型の仮想マシンであり，TOS（Top of Stack）レジスタを用いたスタックマシンである．性能よりもメンテナンス性を重視するため，最適化はあまり行っていない．評価は Python 2.4.1 で行った．

最後に，Scheme 言語の処理系である Gauche 0.8.4[11] の実行も比較の対処とした．Gauche は Scheme プログラムを命令列に変換して実行する仮想マシンである．アキュムレータレジスタをひとつ持つスタックマシンとして実装している．

図 14 の Ruby は現在の Ruby 処理系，YARV は開発した YARV 上で動作した結果をあらわしている．評価は x86 プロセッサ環境で行いそれぞれ Cygwin 上で動作するものを利用した．

なお，loop はその処理系でもっとも高速に繰り返しを行う機能を利用して 3 千万回繰り返しのみを行うプログラムである．fact は 100 の階乗計算を繰り返すプログラムだが，Perl は標準ではある一定の値を超える整数値は多倍長整数を扱うことができず，浮動小数点型に変換するため，他の処理系が多倍長整数を利用することを踏まえ，計測不可として値を 0 秒とした．

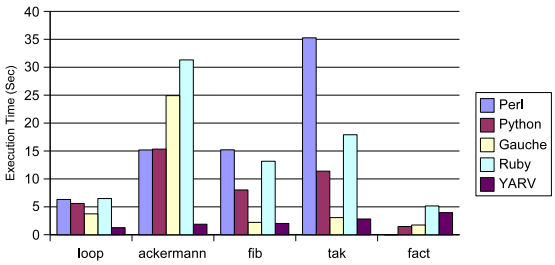

図 14 多言語との速度比較

Ruby を見ると，他の環境と比較して遅く，処理速度が Ruby の問題点のひとつであったことが確認できる．

YARV では，他の処理系よりも fact 以外で高速に実行できていることがわかる．とくに，Perl，Python の処理系と比較して十分高速であることがわかる．

fact のプログラムは 7.2 節で述べたように多倍長整数演算の処理がオーバヘッドの大部分を占めるため，たとえばより高性能な多倍長整数演算ライブラリを用意することで他言語と同等，またはそれ以上の性能にすることが可能である．

Gauche で ackermann 関数を解くプログラムである ackermann が極端に遅いのは，スタックオーバーフローが多発し，スタック伸張のオーバヘッドが処理時間の大部分を占めているからである．

8. 関連研究

Ruby 向け仮想マシンはいくつか提案されている[17] が，ほとんどのプロジェクトは Ruby 処理系としての機能が不十分，性能が現在の処理系よりも遅い，またはプロジェクト自体が消滅しているなどの状態である．

ruby2c[1] は Ruby プログラムをパースして S 式を生成し，それを C 言語プログラムへ変換する．YARV でも AOT コンパイラとして同様のことを行うが，コンパイル対象が S 式ではなく YARV 命令セットを用いる点で異なる．

rubydium[12] は YARV と同様，速度向上を目指した Ruby 処理系である．最適化については，開発者である Kellett はブロックを命令ディスパッチ時にインライン化して実行する方式を提案しているが，まだ実現できていない．YARV でもブロックのインライン化は検討している．ただし，呼ばれた側がこれを行う方式を採用する予定である．

vmgen[7] は YARV と同様，命令記述を特定のフォーマットで行い，これを利用して仮想マシンの C プログラムを自動生成する汎用的な仮想マシン生成系である．また，複数の命令を組み合わせた命令（superinstruction）を自動生成する機能も有し，どの命令を最適化するかを判断するためのプロファイラも備える．

しかし，vmgen ではオペランド融合やスタックキャッシング用の命令を自動生成する機能はない．

内山らの VMB[30] は仮想マシンの形式的な仕様記述からバイトコードインタプリタを生成する．仕様記述を解析するため，少ない記述量から適切な処理系の生成や検証を行うことができるとしている．本稿で述べた VM 生成系は，VM の挙動は C 言語で記述したものを与え，解析はしないため検証などの用途には利用できないが，形式的な表現が難しい処理を容易に記述できることができる．また，生成系の構造が単純なので，拡張は容易である．

文献 29) では，Scheme インタプリタを例に，融合操作によって体系的に命令を追加し，仮想マシンの最適化を行う方法を述べている．YARV ではこの手法を Ruby に適用している．また，YARV ではこれらの融合操作を自動化する仕組みを備えているため，この手法を適用が容易に適用可能である．

JavaVM[24] や .NET[13]，Parrot[20] などの既存の仮想マシン用命令に Ruby プログラムをコンパイルして利用する方法もある[25]．この利点は，すでに実装された優れた最適化器などをそのまま利用できることである．しかし，Ruby のモデルと微妙な違い，たとえばオブジェクトモデルの差異や例外処理の扱いの違いがあり，そのギャップを埋めるためのコストが問題になる．また，Ruby 専用の仮想マシンを開発している YARV では Ruby に特化した最適化が適用可能である．

9. ま と め

本稿では Ruby プログラムを高速に実装するための処理系である YARV について，その設計と実装方法，最適化手法の概要，そしてその性能について述べた．

Ruby 処理系開発の課題を述べ，その上で YARV をどのように設計したかについて述べた．YARV において各種最適化，およびそれを容易に実現するための VM 生成系について詳細を示した．また，実装における詳細も述べ，開発によって得た知見を示した．

ベンチマークプログラムでの評価の結果，現在の Ruby 処理系と比較して高速に実行できることを示した．そして，各種最適化を徐々に適用する評価結果を示すことで，各最適化がどの程度性能向上に寄与するか示した．また，他言語の処理系と比較し，著名なスクリプト言語である Perl や Python よりも高速に実行できることを確認した．

今後の課題として，Ruby との互換性向上，より現実的なアプリケーションでの評価を行いたい．また，今後取り組んでいきたい技術的な要素として次のようなものがある．

ブロック付きメソッド呼び出しの高速化 Ruby では繰り返しをブロック付きメソッド呼び出しで行うことが多いが，現在の YARV ではこの機能の高速化が十分に行えていない．そのため，ブロックのインライン化などを行い高速化する必要がある．

ネイティブスレッド対応 現在の Ruby 処理系はユーザレベルでスレッドを管理しているため，並列実行を行うことができない．そのため，OS が提供するネイティブスレッドで Ruby のスレッドを実行することで並列処理を可能にし，よりスケーラブルなアプリケーションを実現可能にする．

Multi-VM インスタンス 複数の独立した VM を，いわゆるプロセス内で同時に実行する機能は VM 起動のオーバヘッドおよび必要な計算資源の削減[10] が可能である．また Ruby を組み込むアプリケーション開発が容易になる．

これらの課題をクリアし，Ruby 処理系としての完成度を高めていきたい．

謝　　辞

YARV 開発にあたり，YARV 開発用メーリングリストに参加されている方々にはいつも有益なアドバイスを頂いております．感謝いたします．

本処理系の開発プロジェクトは，IPA（情報処理推進機構）の公募事業 2004 年度未踏ソフトウェア創造事業「未踏ユース」（プロジェクト・マネージャ：筧捷彦早稲田大学教授），および 2005 年度未踏ソフトウェア創造事業（プロジェクト・マネージャ：千葉 滋 東京工業大学大学院助教授）に採択され，支援を受けています．

参 考 文 献

1) Davis, R. and Hodel, E.: ruby2c Automatic translartion of ruby code to C. http://zenspider.com/ ryand/Ruby2C.pdf.
2) Ertl, A.: Threaded Code. http://www.complang.tuwien.ac.at /forth/threaded-code.html.
3) Ertl, M. A.: Stack caching for interpreters, *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, ACM Press, pp. 315–327 (1995).
4) Ertl, M. A. and Gregg, D.: Optimizing indirect branch prediction accuracy in virtual machine interpreters, *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, ACM Press, pp. 278–288 (2003).
5) Ertl, M. A. and Gregg, D.: The Structure and Performance of *Efficient* Interpreters, *The Journal of Instruction-Level Parallelism*, Vol. 5 (2003). http://www.jilp.org/vol5/.
6) Ertl, M. A. and Gregg, D.: Combining Stack Caching with Dynamic Superinstructions, *Interpreters, Virtual Machines and Emulators*

(IVME '04), pp. 7–14 (2004).
7) Ertl, M. A., Gregg, D., Krall, A. and Paysan, B.: vmgen: a generator of efficient virtual machine interpreters, *Softw. Pract. Exper.*, Vol.32, No. 3, pp. 265–294 (2002).
8) Free Software Foundation (FSF): GCC Home Page. http://gcc.gnu.org/.
9) Fulgham, B.: The Computer Language Shootout Benchmarks. http://shootout. alioth. debian.org/.
10) Heiss, J. J.: The Multi-Tasking Virtual Machine: Building a Highly Scalable JVM (2005). http://java.sun.com/developer/ technicalArticles/Programming/mvm/.
11) Kawai, S.: Gauche - A Scheme Interpreter. http://www.shiro.dreamhost.com/scheme/ gauche/index-j.html.
12) Kellett, A.: rubydium. http://rubyforge.org/ projects/rubydium/.
13) Microsoft: Microsoft .NET Information. http:// www.microsoft.com/net/.
14) O'Reilly Media, I.: Perl.com: The Source for Perl – perl development, perl conferences. http://www.perl.com/.
15) Piumarta, I. and Riccardi, F.: Optimizing direct threaded code by selective inlining, *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, ACM Press, pp. 291–300 (1998).
16) Python Software Foundation: Python Programming Language. http://www.python.org/.
17) RubyGarden: RUBY: VirtualMachineOptions. http://www.rubygarden.org/ruby? VirtualMachineOptions.
18) Sasada, K.: YARV: Yet Another Ruby VM. http://www.atdot.net/yarv/.
19) Sun Microsystems: Java テクノロジ. http:// jp.sun.com/java/.
20) The Perl Foundation: Parrot - parrotcode. http://www.parrotcode.org/.
21) Thomas, D., Fowler, C. and Hunt, A.: *Programming Ruby*, The Pragmatic Programmers (2004).
22) まつもとゆきひろ, 石塚圭樹: オブジェクト指向スクリプト言語 Ruby, 株式会社アスキー (1999).
23) まつもとゆきひろ他: オブジェクト指向スクリプト言語 Ruby. http://www.ruby-lang.org/ja/.
24) ティム・リンドホルム, フランク・イェリン: Java 仮想マシン仕様第 2 版, ピアソン・エデュケーション (2001).
25) 浅川浩紀: Ruby.NET コンパイラの開発. http://www.asakawa.net/ruby/rubynet_memo.html.
26) 松本行弘: Ruby の真実, 情報処理, Vol. 44, No. 5, pp. 515–521 (2003).
27) 笹田耕一: プログラム言語 Ruby におけるメソッドキャッシング手法の検討, 情報処理学会第 67 回全国大会, Vol. 1, pp. 305–306 (2005).
28) 青木峰郎: Ruby ソースコード完全解説, インプレス (2002).
29) 前田敦司, 山口喜教: Scheme インタプリタにおける仮想マシンアーキテクチャの最適化, 情報処理学会論文誌 (PRO), Vol. 44, No. SIG13, pp. 47–57 (2003).
30) 内山雄司, 緒方大介, 脇田建: 仮想機械の仕様記述に基づくバイトコードインタプリタ生成系, 情報処理学会論文誌 (PRO), Vol. 46, No. SIG 6, pp. 1–17 (2005).