高速な Ruby 用仮想マシンの開発

Efficient Implementation of Ruby Virtual Machine

笹田　耕一

Koichi Sasada

主査　竹内郁雄教授

東京大学大学院情報理工学系研究科

博士学位論文

(2007 年 12 月提出)

要旨

近年，多様化するソフトウェア需要に応えるため，開発効率の良いプログラミング言語が求められている．その中で，オブジェクト指向スクリプト言語 Ruby が実用的で開発効率の良い言語として注目されている．しかし，従来の Ruby 処理系は抽象構文木を辿る単純な実装のため実行が遅いという問題点があった．そのため高い性能が必要となるソフトウェアの開発には用いることができなかった．

本研究は Ruby を高速に実行する処理系の実現手法を研究し，プログラミング言語 Ruby の可能性を広げることを目的とする．そして，世界中で利用されるような実用的なソフトウェアを開発することを目標とする．

高速な Ruby 処理系を実現するための課題として，まず従来の Ruby 処理系の抽象構文木を辿る単純な実行方式の改善がある．従来の方式は，実現は容易であるが性能に問題があることが知られている．動的型システムおよび強力なリフレクション機能に代表される Ruby の動的特性は，Ruby プログラミングに柔軟性を与えるが，静的解析による最適化が不可能となるため，Ruby 処理系に適した実行時最適化技術の開拓が必要となる．さらなる高速化を目指すためには，近年とくに普及してきたマルチコア，マルチプロセッサを利用したメモリ共有型並列計算機による並列実行の実現が重要である．しかし，過去のプログラム資産を生かすためにはスレッドセーフでない機能の扱いが問題となる．

上記の課題に加え，実用的な Ruby 処理系を実現するため，互換性，保守性，移植性が開発全体での課題となった．実際的なプログラミングを行うためには過去の豊富なプログラム資産を活かさなければならない．そのため従来の処理系との互換性の維持が必須である．とくに Ruby 処理系を拡張するための C 拡張ライブラリと，C 拡張ライブラリを記述するために必要となる Ruby C API の実現が課題となった．また，言語処理系は複雑なソフトウェアであるため，保守性の向上が重要である．そして，移植性の維持は多くの計算機環境で動作する実用的な処理系とするために重要である．

高速な Ruby 処理系を実現に関するこれらの課題を解決するために, 本研究では YARV:

Yet Another RubyVM という仮想マシン方式による Ruby 用言語処理系を新たに開発した．まず，Ruby プログラムを正しく表現できる YARV 命令セットを設計し，Ruby プログラムを YARV 命令セットで構成された YARV 命令列へ変換するコンパイラを作成した．YARV 命令列を実行するための仮想マシン YARV は仮想マシンアーキテクチャとして一般的なスタックマシンとした．例外処理は実行時にオーバヘッドが不要となる表引き法を採用したが，Ruby C API の互換性確保のため C 言語での大域ジャンプを用いる方式と併用して実現した．

さらに，開発した仮想マシン YARV に実行時最適化技術を Ruby プログラムの意味を変更しないよう適用した．とくに，Ruby の動的特性を堅持しながら高速化を行うため，実行時最適化を中心とした Ruby 処理系に適用可能なさまざまな高速化手法を検討し，特化命令やインラインキャッシュを利用した高速化を行った．また，既知の仮想マシン高速化手法である命令の融合操作，静的スタックキャッシングを YARV 命令列に適用した．

仮想マシンの逐次実行の性能評価を行うため，典型的な Ruby プログラムを実行して，それぞれの最適化がどの程度性能向上に寄与しているかを調べた．総合的には，仮想マシンを利用しない場合にくらべてマイクロベンチマークで最大 20 倍，マクロベンチマークで平均 1.5 倍程度の性能向上を達成したことを確認した．これによって本研究で検討した最適化手法の有効性を示すことができた．

YARV の開発では，VM（仮想マシン）記述言語および VM 生成系を作成することで仮想マシンの開発を容易にし，ソフトウェアの保守性を高めた．作成した VM 生成系は VM 記述言語で記述された VM 記述からコンパイラやアセンブラ，評価器のプログラム片を自動的に生成する．また，命令の融合操作，静的スタックキャッシングといった機械的に適用可能な最適化に関して，VM 生成系を用いて必要な命令を半自動生成することで仮想マシンの高速化を容易にした．この工夫により，本システムの方式検討を容易にし，実際に開発期間を大幅に短縮することができた．

メモリ共有型並列計算機上での並列実行による高速化を行うため，Ruby スレッド処理機構を計算機システムが提供するネイティブスレッドを利用して構成し，Ruby スレッドの並列実行をサポートした．対応したネイティブスレッド処理機構は POSIX Thread および Windows スレッドの 2 種類である．Ruby スレッドとネイティブスレッドの対応は，移植性を重視し，1 対 1 対応とする方式をとった．ネイティブスレッドを利用した Ruby スレッド処理機構を実現するためには，とくに Ruby スレッドに対する割り込みが問題となるが，ブロック解除関数を登録する方式を採用することで解決した．

Ruby スレッドの並列化にあたっては，スレッドセーフでない既存の拡張機能が問題となった．本研究では，このような膨大なプログラム資産をそのまま利用するためにスレッ

ドセーフでない機能を呼び出すときにだけジャイアントロックを利用し，そのほかは細粒度ロックを利用して並列に動作するシステムを構築した．しかし，既存のスレッドセーフでない機能を順次スレッドセーフな処理に書き変えていくことで，段階的に並列度を向上することができるシステムとした．Ruby のようなオープンソースソフトウェアとして開発を進められている言語の性能や機能を継続的に発展させていくためには，この方法が適している．こうした上で，ジャイアントロックや細粒度ロック導入によるオーバヘッドを削減するためにデータ構造を最適化し，ロックを考慮した Ruby スレッドスケジューリングを実現した．これを実際にマルチコアプロセッサを搭載したメモリ共有型並列計算機上で実行し評価した結果，逐次実行に比べて利用するプロセッサコア数に応じた実行時間の短縮を実現出来たことを確認した．

本研究で開発した仮想マシン YARV は公式 Ruby 処理系に取り込まれることが決定しており，オープンソースソフトウェアとして公開される予定である．本研究によって世界中で多くの人に利用されるソフトウェアを開発するという実用的な面での貢献も行うことができた．今後はより詳細な解析や実行時フィードバックを利用した高速化，Ruby 向けの新しい並列化手法の検討が必要である．

目次

要旨 i

第 1 章　序論 1
1.1　背景 1
1.1.1　スクリプト言語の必要性の増大 1
1.1.2　オブジェクト指向プログラミング言語 Ruby 3
1.1.3　高速な Ruby 処理系開発の課題 4
1.2　本研究の目的 6
1.3　本論文の構成 7

第 2 章　オブジェクト指向スクリプト言語 Ruby 9
2.1　はじめに 9
2.2　オブジェクト指向スクリプト言語 Ruby 10
2.2.1　Ruby の特徴 10
2.2.2　Ruby の利用用途 21
2.2.3　Ruby 処理系 22
2.3　Ruby 処理系実装の課題 26
2.3.1　オブジェクト指向機能 26
2.3.2　オブジェクトモデルとガーベージコレクション 26
2.3.3　ブロックつきメソッド呼び出しとクロージャ 27
2.3.4　動的な実行モデル 27
2.3.5　例外処理 29
2.3.6　C 言語との連携 30
2.3.7　スレッドの実現 32
2.3.8　課題のまとめ 32

2.4 まとめ 33

第 3 章 Ruby 用仮想マシン YARV 35
3.1 はじめに 35
3.2 YARV の概要 36
3.2.1 YARV の全体像 36
3.2.2 YARV による Ruby プログラムの実行方法 37
3.2.3 YARV の処理の流れ 37
3.2.4 YARV の計算モデル 37
3.3 YARV 基本命令セット 41
3.4 コンパイラ 44
3.5 命令列評価器 45
3.6 YARV の例外処理機構 47
3.6.1 キャッチテーブル検索による例外のハンドリング 47
3.6.2 動的埋め込みタグによる例外のハンドリング 50
3.6.3 キャッチテーブルと動的埋め込みタグの併用 50
3.7 まとめ 52

第 4 章 YARV の高速化 55
4.1 はじめに 55
4.2 コンパイル時最適化 55
4.2.1 覗き穴最適化 56
4.2.2 末尾呼び出しの最適化 56
4.2.3 リフレクション機能の高速化 57
4.3 実行時最適化手法 58
4.3.1 スレッデッドコード 58
4.3.2 特化命令 59
4.3.3 オペランド・命令融合 61
4.3.4 インラインキャッシュを利用した高速化 62
4.3.5 静的スタックキャッシング 67
4.3.6 プロファイラ 69
4.3.7 ネイティブコンパイラ 70
4.4 実装 70

4.5 評価 71
4.5.1 各機能の評価 72
4.5.2 マイクロベンチマークでの評価 74
4.5.3 マクロベンチマークでの評価 76
4.5.4 他言語環境との比較 78
4.6 関連研究 80
4.7 まとめ 80

第 5 章 仮想マシン生成系 83
5.1 はじめに 83
5.2 VM 生成系 83
5.3 VM 記述言語と生成された VM コード片 85
5.4 VM 生成系を利用した最適化命令の生成 87
5.4.1 融合命令の生成 87
5.4.2 静的スタックキャッシュ命令の生成 90
5.4.3 コード生成における VM コード量増加 91
5.5 VM 以外のコード生成 92
5.5.1 コンパイラ用コードの生成 92
5.5.2 アセンブラ，逆アセンブラ用コードの生成 92
5.6 関連研究 93
5.7 まとめ 93

第 6 章 ネイティブスレッドを用いた Ruby スレッド処理機構 95
6.1 はじめに 95
6.2 Ruby スレッドの並列実行手法の検討 99
6.2.1 旧 Ruby 処理系用スレッド処理機構と問題点 99
6.2.2 ネイティブスレッドを利用した Ruby スレッド処理機構の実現手法の検討 101
6.3 ネイティブスレッドを利用した Ruby スレッドの実現 106
6.3.1 Ruby スレッドシステムの全体像 106
6.3.2 Ruby スレッド管理データ 107
6.3.3 Ruby スレッドの制御手法 108
6.3.4 Ruby スレッドのスケジューリング 109

6.3.5 Ruby スレッドへの割り込み . . . 109
6.3.6 シグナルの扱い . . . 112
6.3.7 Ruby スレッドの一時停止 . . . 112
6.4 まとめ . . . 113

第 7 章 Ruby スレッドの並列化 117
7.1 はじめに . . . 117
7.2 並列にアクセス可能なスレッド管理データ . . . 118
7.3 並列実行に対応したメモリ管理 . . . 118
7.3.1 旧 Ruby 処理系のメモリ管理 . . . 118
7.3.2 オブジェクトアロケーション . . . 119
7.3.3 ガーベージコレクション . . . 120
7.4 排他制御の導入 . . . 121
7.4.1 ハッシュ表の排他制御 . . . 121
7.4.2 ジャイアントロックの導入 . . . 122
7.4.3 スレッドセーフな処理を宣言するための API . . . 122
7.5 排他制御オーバヘッドを削減するための工夫 . . . 123
7.5.1 ロック不要なメソッドキャッシュの参照 . . . 123
7.5.2 1 スレッド実行時のジャイアントロック . . . 125
7.5.3 スピンロックの利用 . . . 125
7.5.4 利用 CPU の制限 . . . 125
7.5.5 単一実行権による逐次実行 . . . 127
7.6 評価 . . . 127
7.6.1 スレッド制御プリミティブの評価 . . . 127
7.6.2 マイクロベンチマーク . . . 129
7.6.3 利用 CPU の制限 . . . 132
7.7 関連研究 . . . 134
7.8 まとめ . . . 134

第 8 章 結論 137
8.1 本研究の成果と得られた知見 . . . 137
8.1.1 仮想マシン YARV の構築 . . . 137
8.1.2 Ruby の並列化 . . . 140

8.2 今後の課題 142
8.3 まとめ 143

付録 A YARV 命令一覧 145
A.1 基本命令 145
A.1.1 nop カテゴリ 145
A.1.2 variable カテゴリ 146
A.1.3 put カテゴリ 147
A.1.4 stack カテゴリ 148
A.1.5 setting カテゴリ 148
A.1.6 class/module カテゴリ 148
A.1.7 method/iterator カテゴリ 149
A.1.8 exception カテゴリ 149
A.1.9 jump カテゴリ 149
A.2 最適化用命令 150

謝辞 151

参考文献 153

図目次

2.1 動的な評価を利用した例 28
2.2 動的評価を利用して数値演算を書き換えた例 29
2.3 Ruby の例外処理機能を用いたプログラムの例 30
2.4 C 関数を Ruby 処理系へ登録するための手続き 31

3.1 YARV での Ruby プログラムの実行フロー 38
3.2 旧 Ruby 処理系での Ruby プログラムの実行フロー 39
3.3 例外処理の例 48
3.4 スタックポインタを調節する必要のある Ruby プログラムの例 49
3.5 例外処理の実行の様子 51

4.1 覗き穴最適化の例 56
4.2 命令ディスパッチ手法の比較 59
4.3 特化命令 opt_plus の擬似コード 60
4.4 グローバルメソッドキャッシュアルゴリズム 64
4.5 インラインメソッドキャッシュアルゴリズム 65
4.6 定数アクセスで利用するインラインキャッシュ命令 66
4.7 スタックキャッシングの状態遷移図 69
4.8 繰り返し実行速度の評価 73
4.9 メソッド起動速度の評価 74
4.10 x86 プロセッサ上でのマイクロベンチマークの評価結果 75
4.11 x86_64 プロセッサ上でのマイクロベンチマークの評価結果 75
4.12 マクロベンチマークの評価結果 77
4.13 他言語との速度比較 79

5.1 VM 生成系 84

5.2 VM 記述言語の記述例および生成された C 言語のプログラム片 86
5.3 生成されたオペランド融合命令 88
5.4 dup 命令の定義 88
5.5 dup 命令と mult_plusConst の融合命令 89
5.6 生成されたスタックキャッシング命令（$S_{ab} \rightarrow S_{ax}$） 91

6.1 Ruby におけるスレッド生成と合流・排他制御の例 96
6.2 Ruby スレッドを利用したエコーサーバプログラム (Ruby リファレンスマニュアルより引用) 97
6.3 同期粒度の方式検討 104
6.4 Ruby スレッドシステムの全体像 107
6.5 スレッド管理データ 108
6.6 他の Ruby スレッドに対し例外を発生するプログラム 110
6.7 ブロック解除関数とその利用例 111
6.8 Pthread 環境での一時停止処理 113

7.1 旧 Ruby 処理系でのメモリ管理 119
7.2 並列実行に対応したメモリ管理 120
7.3 GC のための同期 121
7.4 メソッドキャッシュ実現手法 124
7.5 ジャイアントロック獲得処理（Pthread 環境） 126
7.6 コンテキストスイッチの実行時間 129
7.7 並列プログラムのマイクロベンチマーク結果（プログラムごと） 130
7.8 並列プログラムのマイクロベンチマーク結果（コア数での性能推移） . . 131
7.9 フィボナッチ数を求める Ruby プログラム 132
7.10 利用 CPU 制限の効果の確認 133

表目次

2.1 Ruby 用ライブラリの一部 20

3.1 命令セットのカテゴリ一覧 41
3.2 例外処理用の表のエントリ 48

4.1 最適化の略語一覧 72
4.2 VM 基本機能の評価（各 3 千万回の試行） 74
4.3 マイクロベンチマーク一覧（括弧内は主な処理） 76
4.4 マクロベンチマーク一覧 77

7.1 スレッド制御プリミティブの性能評価 128
7.2 GL 獲得・競合頻度 133

第 1 章

序論

本研究では，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高性能な仮想マシンを開発した．開発した仮想マシンには Ruby プログラムを高速に実行するための種々の機構を検討し搭載した．また，Ruby スレッドの並列実行を行うための仕組みについて検討し，実際にメモリ共有型並列計算機上での評価を行った．

本章では，まず本論文の背景，とくにオブジェクト指向スクリプト言語 Ruby とその性能的問題点を述べ，本研究の目的を示す．そして最後に本論文の構成を述べる．

1.1 背景

1.1.1 スクリプト言語の必要性の増大

近年，計算機の性能向上は目覚ましい．科学技術計算の性能向上は言うにおよばず，業務用，個人用計算機としても安価に高性能な計算機を導入することが可能である．このような豊富な計算機資源を背景に，社会の計算機システムへの期待はますます高まっている．

一方，計算機システムを構成する両輪の片方であるソフトウェアは，実行効率を重視したプログラミング言語で開発されることが多かった．たとえば，C/C++ 言語 [65, 50] などの低水準言語，Java 言語 [52] などの静的型付きオブジェクト指向言語などを利用してソフトウェア開発を行うなどである．また，性能を重視する場面では計算機に依存する機械語を用いたプログラミングが行われている．これらの実行効率を重視したプログラミング言語を用いたソフトウェア開発では，いかに短時間でソフトウェアを開発するか，という開発効率の問題を生じる．Ousterhout[37] による分類では，このような効率を重視し

たプログラミング言語をシステムプログラミング言語と分類している．システムプログラミング言語はデータ構造やアルゴリズム等を記述するために設計されており，メモリ等の計算機資源を効率よく利用することができるよう，強い型付け言語となっている．

一方，開発効率のよいプログラミング言語としてスクリプト言語が知られている．Ousterhout はスクリプト言語をグルー，つまり糊付け言語として定義している [37]．スクリプト言語は型付けが無く，強力な部品を組み合わせて迅速にプログラミングを行うことができる．また，システムプログラミング言語と比較して，プログラムの記述量を短くするための構文糖衣を用意したり，処理系をインタプリタ言語としてコンパイル等の余分な処理が不要にして，開発を迅速に行うための工夫を多く取り込んでいる[*1]．

計算機が利用され始めた初期から，システムプログラミング言語とスクリプト言語は補完的な存在であった [37]．例えば，IBM の OS/360 で利用されていた JCL(Job Control Language) は，Fortran や PL/I などのシステムプログラミング言語に対するスクリプト言語であった．UNIX 環境では，C をシステムプログラミング言語として利用する傍ら，シェルスクリプトを利用してシステムを構築してきた．近年の個人用計算機上では，C や C++ でシステムプログラミングを行い，VisualBasic 等のスクリプト言語でアプリケーションが開発されてきた．インターネットが広く利用される近年では，Java がシステムプログラミング言語で Perl[36] や JavaScript[5] などがスクリプト言語として広く利用されている．システムプログラミング言語で部品を開発し，その部品をスクリプト言語で組み立てるという具合でシステムプログラミング言語とスクリプト言語は相補的な関係となっている．

従来は性能や言語の機能不足の問題から，記述量は多いが性能の良いシステムプログラミング言語を利用することが多かった．しかし，最近はスクリプト言語が利用可能な場合が多くなってきている．まず，計算機の性能向上に伴い，今まで問題であったスクリプト言語の処理性能があまり問題にならなくなってきたため，従来はシステムプログラミング言語と併用していた用途でもスクリプト言語のみでプログラムを記述することが増えてきた．また，低機能な文法機能しか備えなかったスクリプト言語が，例えばオブジェクト指向機能を備えるなど高機能化しており，プログラム全体をスクリプト言語で構築可能とするほど記述性が向上した．そして，GUI アプリケーションやウェブアプリケーション等，スクリプト言語を用いることでほぼ実現できるアプリケーションの需要が多くなった．こ

[*1] なお，この分類はすべてのプログラミング言語を正確に分類する方法ではない．例えば，Lisp はシステムプログラミング言語およびスクリプト言語の長所それぞれの実現を目指しており，単純に分類することが出来ない．本研究はプログラミング言語の体系的な分類を目的としているわけではないため，この程度の分類に留めておく．

れらの背景の下，システムプログラミング言語に比べて，スクリプト言語を重視したソフトウェア開発が注目されている．

1.1.2 オブジェクト指向プログラミング言語 Ruby

そのような背景の下，高機能なスクリプト言語として，オブジェクト指向スクリプト言語 Ruby[63, 55, 64] が実用的で開発効率の良いプログラミング言語として注目されている．松本等によって開発された Ruby は，Java や C++ 等のよく知られたオブジェクト指向言語と比べ，少ない記述量でオブジェクト指向プログラミングやメタ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ミング言語である．

Ruby はプログラミング言語として次のような特長がある [64].

- シンプルな文法
- 一般的なオブジェクト指向機能（クラス，メソッド呼び出しなど）
- 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
- ガーベジコレクタ
- ダイナミックローディング
- 移植性の高さ
- 各分野のために開発された Ruby 用ライブラリ

松本は，文献 [70] において，Ruby が世界的に人気をもつプログラミング言語であり，多くのユーザを擁していると紹介している．そして，Ruby が人々に受け入れられた理由が，上述した特長により，簡単に，楽しく Ruby プログラミングを行うことができるため，と分析している．

Ruby の長所，とくにオブジェクト機能を利用した記述の容易性による開発効率の良さから，近年とくにウェブアプリケーションなど，サーバサイドアプリケーションの記述言語として多く利用されている．他にも，汎用言語としてその他の用途，たとえば GUI アプリケーションやゲーム開発，業務システムの構築などに広く利用されている．

しかし，従来の Ruby 処理系は，他のスクリプト言語などの処理系に比べ実行速度が十分でないという問題がある [19]．この原因の一つは，従来の Ruby 処理系が 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って実行する方式を採用している

点にある．従来の 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．この方式は，実装が容易であるという利点があるが，高い性能が出ないことが知られている．

計算機の性能向上により，スクリプト言語において性能が問題になることは少なかったが，言語処理系の性能が直接影響するような計算に利用するには，やはり性能上の問題で Ruby を適用することができなかった．言語処理系の性能が影響する分野としては，数値計算や言語処理系の実現を含む記号処理が挙げられる．これは，Ruby が利用できる範囲に制限があるということを示している．記述能力はこれらの応用分野に十分適用可能だが，性能上の制約から Ruby が活用範囲に制限があることは解決すべき問題である．そこで，Ruby プログラムを高速に実行するための処理系が必要である．

1.1.3 高速な Ruby 処理系開発の課題

オブジェクト指向スクリプト言語 Ruby で記述されたプログラムを高速に実行する Ruby 処理系を実装する課題として，まず従来の Ruby 処理系の抽象構文木を辿る単純な実行方式の改善がある．Ruby プログラムを実行する本体である評価器は，Ruby プログラムを構文解析して生成する抽象構文木を木構造データとして再帰呼び出しをする関数として実現されている．この方式では前述したとおり高い性能が出ないことが知られている．また，既知の言語処理系向けの最適化技術を適用することができない．そのため，高速な言語処理系を構築可能なモデルに評価器を置き換える必要がある．

高速化における重要な課題として，変数に型がない動的型システムや，実行時情報へのアクセスを行うための強力なリフレクション機能に代表される Ruby の動的特性への対応がある．動的型システムは変数の型を指定する必要がないので，コンポーネントの組み合わせや再利用が容易となる．また，リフレクション機能を利用することで実行時に入力データに応じて定義を追加する，機能を変更するといったメタプログラミングが容易になる．このように，動的特性は Ruby プログラミングに柔軟性を与えるが，静的解析による最適化を不可能にする．例えば，リフレクション機能を利用してプログラムの定義を任意のタイミングで変更すると，実行前の静的解析による最適化が大きく制限される．このように，動的特性を持たない言語で知られている言語処理系最適化技術を利用することができず，プログラミング言語 Ruby に適した実行時最適化技術の検討が必要となる．

Ruby 処理系のさらなる高速化を目指すためには，近年とくに普及してきたマルチコア，マルチプロセッサを利用したメモリ共有型並列計算機による並列実行の実現が重要である．Ruby は言語レベルでマルチスレッド実行をサポートしているため，Ruby レベル

で生成する Ruby スレッドを並列計算機上で並列実行させるモデルを提供するのが自然である．しかし，従来の Ruby 処理系は独自に開発したユーザレベルスレッド型のスレッド処理機構を用いているため，単純には並列実行をすることができない．Ruby スレッドの並列化を実現するには，移植性を考慮して OS が提供するネイティブスレッド処理機構を利用する必要がある．また，実際に Ruby スレッドを並列に実行するためには過去のプログラムの互換性が大きな問題となる．実用的なプログラミング環境を構築するために不可欠な過去のプログラム資産，とくに C 言語で開発された Ruby 用拡張機能は，従来の Ruby 処理系が並列化を考慮していなかったためスレッドセーフではない．このようなプログラム資産をスレッドセーフな処理に一挙に書き換えるのは現実的ではないため，過去のプログラム資産を利用可能のまま並列実行可能な処理系とすることが課題となる．

上記高速化の課題に加え，実用的で高速な Ruby 処理系を実現するためには，以下に述べるように互換性，保守性，移植性がの維持が開発を通じての大きな課題となる．

Ruby にはすでに多くのライブラリが開発されており [31, 43]，これらライブラリを利用することで複雑なソフトウェアの構築を容易にしている．Ruby の実用的な特徴を維持するためには，過去の処理系との互換性に注意を払い，出来る限り修正を必要とせずに過去のプログラム資産を利用可能にすることが重要である．とくに Ruby 処理系を拡張するための C 拡張ライブラリと，C 拡張ライブラリを記述するために必要となる Ruby C API を，従来の Ruby 処理系と同等に利用可能にすることは，処理系の再構築にあたって大きな課題である．例えば，現在の Ruby 処理系での例外処理は，再帰関数を利用した評価器の性質をうまく利用して，C 関数の `setjmp/longjmp` を利用して実現しているが，再帰関数構成を変更してそのような処理手法を維持するのは難しい．並列化を行う上でも，C 言語で記述したスレッドセーフでない機能をどのように利用可能にするか，というのが大きな課題となる．

また，言語処理系は複雑なソフトウェアであるため，保守性の向上は重要な課題である．本研究で開発するソフトウェアを，実用的な処理系として継続して利用されるものにするためにも保守性が高いことが重要になる．

そして，移植性の維持は多くの計算機環境で動作する実用的な処理系とするために重要である．性能向上のためには各システム上で特有の機能，例えば特定のプロセッサの特定の命令を直接利用するなどの，特化した高速化手法が考えられるが，そのような機能に依存すると，その他の計算機で利用することが困難になる．また，数ある計算機環境ごとに，専用の高速化技術を開発するには大きな労力が必要となる．そのため，環境に依存しない，一般的な高速化技術を開発することが重要となる．

多くの言語処理系の研究では，特定計算機環境を前提した高速化や，ソフトウェアの保

守性を考慮しない複雑なアルゴリズムの提案であることが多い．しかし，本研究はすでに多くのプログラム資産がある Ruby 処理系というソフトウェアに存在する莫大なソフトウェア資産との互換性等の種々の制約を守りながら，実際に世界中で広く利用され得る一般的で実用的な言語処理系を目指しているという点が従来の言語処理系開発と異なる点であり，本研究特有の課題である．

1.2 本研究の目的

本研究では Ruby プログラムを高速に実行する言語処理系を実現することを目的とする．そして，研究のためのソフトウェアとして終わらせることなく，世界中で利用されるような実用的なソフトウェアを開発することを目標とする．

具体的な方針としては，オブジェクト指向スクリプト言語 Ruby で記述されたプログラムを高速に実行するために抽象構文木を再帰的に辿る評価器ではなく，命令列を解釈実行する仮想マシンを用いた言語処理系を開発し，仮想マシン向けの高速化手法を適用する．また，さらなる高速化手法として，メモリ共有型並列計算機上での Ruby スレッドの並列実行を実現する．開発した処理系は従来の Ruby 処理系と出来る限り互換性を持たせ，ソフトウェアの保守性，移植性に関しても注意して開発する．

まず，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）*YARV: Yet Another RubyVM* [45] を開発する．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，知られている仮想マシンの高速化手法，とくに動的性質を持った Ruby に適した最適化手法を提案し実装する．

仮想マシンは複雑なソフトウェアであるため，仮想マシンを構築するために単純な VM 生成系を開発し利用することで仮想マシンの開発効率の改善，および保守性を向上する．VM 記述言語で基本的な VM の各命令の処理を記述して，仮想マシンのレジスタ操作等の必要な処理を含んだ仮想マシン構築に必要なプログラムを自動的に生成する．また，高速化を行う際に必要な最適化命令の半自動生成も行う．プログラムの自動生成を用いることで開発の手間を低減し，手作業によるバグが混入する機会を防ぐことで，ソフトウェアの保守性の向上を目指す．

また，Ruby を並列に実行するために計算機システムが提供するネイティブスレッド処理機構を利用して並列実行が可能な Ruby スレッド処理機構を実現する．ただし，並列処理をサポートする従来の言語処理系の研究開発とは違い，逐次実行を前提として開発され

たネイティブメソッドなど，過去のプログラム資産を利用可能としたまま並列実行による性能向上を実現するという互換性の維持に着目し，実用的に並列実行可能なプログラミング言語処理系の実現を目指す．

1.3 本論文の構成

本論文では，Ruby プログラムを高速に実行するための処理系である YARV の構成とそこで採用した高速化手法について述べる．また，メモリ共有型並列計算機を利用した Ruby の並列実行について述べる．

以下，2 章ではオブジェクト指向プログラミング言語 Ruby とその処理系の紹介を行い，Ruby プログラムを高速化する上での課題について，分析する．

3 章で開発した Ruby プログラムを高速に実行するための Ruby 用仮想マシンである YARV: Yet Another RubyVM の概要を述べる．4 章でスクリプト言語 Ruby を高速に実行するための工夫について述べ，実際にベンチマークプログラムを利用してそれぞれの最適化がどれだけ性能向上に寄与したかについて述べる．5 章で処理系の開発，保守を容易にするために開発した VM （仮想マシン）生成系および VM 記述言語について説明する．

6 章でネイティブスレッドを用いた Ruby のスレッド処理機構について述べる．7 章で Ruby 用仮想マシン YARV を，既存のプログラム資産を生かしながら並列化するための仕組みを述べ，実際に並列実行してどの程度性能が向上した評価する．

8 章で本研究をまとめ，今後の課題を示す．

第 2 章

オブジェクト指向スクリプト言語 Ruby

2.1 はじめに

オブジェクト指向スクリプト言語 Ruby[63, 55, 64] は，松本によって設計されたプログラミング言語である．オブジェクト指向プログラミングを容易に活用可能とするためのオブジェクトシステムを搭載し，そのほかガーベージコレクション，マルチスレッド機能など，おおよそ必要とされる言語機能を搭載している．

Ruby を実行するために必要な Ruby 処理系は，松本を中心に 1993 年より開発が進められており，オープンソースソフトウェアとして現在も精力的に開発が進められている．以前は Ruby と言えばこの松本らの処理系を指したが，最近は他にも Java 言語で開発されている JRuby，C#で開発されている Ruby.NET や IronRuby，Ruby で記述されている循環評価器となる Rubinius など，そのほかの処理系が登場している．

そこで，他の Ruby 処理系と区別するために，松本等によって開発されている処理系は C 言語を用いて開発されていることから CRuby と表記することにする．また，本研究では CRuby に仮想マシンを搭載し高速化を行うため，搭載後の処理系と区別するために，現在一般に広く利用されている，仮想マシンを搭載していない CRuby を旧 Ruby 処理系と表記することにする．なお，松本らによる Ruby 処理系はその他の Ruby 処理系のリファレンス実装として扱われている．

本章では，目標とする高速な Ruby 処理系開発についての問題分析を分析するために，オブジェクト指向プログラミング言語 Ruby に関する次の 3 点について述る．まず，本研究が対象とするプログラミング言語 Ruby について，本論文を読み進める上で必要となる

特徴や文法を述べる．そして，CRuby や JRuby など，既存の Ruby 処理系の実装を紹介し，それぞれの特徴とそれぞれの処理系が抱えている問題点を述べる．とくに，本研究の開発対象となる CRuby について詳しく述べる．そして，最後に Ruby 処理系を実装にあたり問題となる点について，とくに高速化に関する課題についてまとめ，本研究の課題を明らかにする．

2.2　オブジェクト指向スクリプト言語 Ruby

プログラミング言語 Ruby 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ミング言語である．Awk[58] や Perl[36] などのスクリプト言語の手軽さを継承しながら，C++/Java などでおなじみのクラスベースのオブジェクトシステムを備えている．また，特異クラスなどの仕組みによるメタプログラミングもサポートしている．そのほか，例外処理やガーベージコレクションなどをサポートしている．

本節では Ruby のプログラミング言語としての特徴を概説する．そして，現存する Ruby 処理系について紹介する．

2.2.1　Ruby の特徴

プログラミング言語 Ruby の特徴を以下にまとめる．なお，Ruby の詳細を述べるのは本論文の趣旨ではないため，文法など言語仕様の詳細は他の文献 [55, 64] を参照されたい．

以下に示す特徴により，オブジェクト指向スクリプト言語 Ruby を用いることで，松本は簡単に，楽しくプログラミングを行うことができるとしている [70]．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．

■シンプルな文法　自然な中値記法を基本としている．書きやすさを向上させるため，不要なセミコロン，括弧などが省略可能である．

■豊富なリテラル　配列，ハッシュ，正規表現オブジェクトなど，いろいろなオブジェクトをプログラム中に直接リテラルとして記述可能である．例えば，配列やハッシュ，正規表現は次のように記述できる．

```
[1, 2, 3]           # 要素 1, 2, 3 を持つ 3 要素の配列
{:a => 1, :b => 2} # :a, :b をキーとし，それぞれ 1, 2 を値とするハッシュ
/abc/               # 文字列 abc にマッチする正規表現
```

■強力な文字列操作・正規表現 sed や awk，Perl に代表される，正規表現 [18] の利用を含んだ強力な文字列操作機能をビルトイン機能として備えている．

文字列操作機能は，部分文字列の抽出，部分文字列の変換のような一般的な操作から，大文字小文字の変換や文字列両端の空白文字の除去など，特定の用途だが実際によく利用される便利なメソッドなどを用意している．これらの機能は組み込みクラスである String クラスのメソッドとして備えている．

正規表現は Perl 由来の拡張された正規表現を実現しており，/foo/のようなリテラルとして記述できる．正規表現は文字列のマッチングに利用可能である．

■オブジェクト指向機能 クラスベースのオブジェクトシステムを備えている．単一継承をベースとしたクラス構造を構築することができる．クラスにはメソッドが定義でき，クラスのインスタンスを生成することが可能である．

次のプログラムでは，(A) クラス C，およびそのインスタンスメソッド m を定義し，(B) C のインスタンスを生成し変数 obj へ代入する．そして，生成したオブジェクトをレシーバとしてメソッド m を呼び出している．

```
# (A)
class C
  def m()
    print 'Hello World!'
  end
end

# (B)
obj = C.new  # クラス C のインスタンスを生成
obj.m()      # クラス C のインスタンスメソッド m を呼び出し
```

変数 obj には型の指定を行っていないことに注意されたい．つまり，変数 obj にはどのようなクラスのインスタンスも格納することが可能である．メソッド呼び出しは，レシーバオブジェクトのクラス情報を元にメソッド探索を実行時に行い，適切なメソッドを実行する．

なお，クラス C のインスタンスメソッド m を C#m と記述する．

■先進的なオブジェクト指向機能　Ruby には，C++ や Java といった広く利用されているオブジェクト指向プログラミング言語に加え，次に述べるような先進的なオブジェクト指向機能を備えている．

Ruby では，モジュールという概念を用いることで，単一継承ベースのオブジェクトシステムに柔軟な機能追加を可能としている．単一継承は多重継承システムと違い菱形継承などの問題が無く単純であるという利点があるが，実装の共有が困難であるという問題点がある．そこで，Ruby ではモジュールという概念を導入し，実装の共有を容易にした．モジュールはクラスと同様にメソッドを定義できるが，インスタンスを生成することはできない．モジュールは include という操作によってクラスにモジュールの機能を付与することができる．

```
module M
  def say_hello
    print 'hello'
  end
end

class C
  include M
end

class D
  include M
end
```

この例では，モジュール M とクラス C，—verb—D—を定義している．モジュール M では say_hello メソッドを定義しており，クラス C，D ではモジュール M を include する

ことにより，どちらのクラスでも say_hello メソッドが利用できる．モジュールは Java 言語での interface に似ているが，実装も共有できるという点で異なる．

モジュールは include でクラスに実装を取り込む以外にも，特定のオブジェクトのみにそのモジュールの機能を付与する extend という操作も可能である．

また，特異メソッドという概念を導入し，オブジェクトごとに特別なメソッドを定義することが出来るようになっている．

```
obj = Object.new # (A)

# (B) メソッド foo の追加
def obj.foo()
  print 'Foo'
end

# (C) メソッド object_id の追加
def obj.object_id()
  rand()
end
```

(A) で生成したオブジェクトのみに (B) メソッド foo を定義, (C) メソッド object_id を再定義している．ここではメソッド foo を追加しているが，そのクラスに定義されていたメソッドを上書きすることもできる．

特異メソッドやモジュールの extend による特定オブジェクトへの機能付与など，特定のオブジェクトごとに挙動を変更することができる機能は，JavaScript[5] や SELF[57] のようなプロトタイプベースのオブジェクト指向言語に似ている．

Ruby のクラス定義は任意の時点で変更可能である．これをオープンクラスということもある．

```
# (A) define C class
class C
  def m()
    print 'm'
  end
end

# (X)
obj = C.new
obj.m #=> m を出力

# (B) reopen C class
class C
  def n()
    print 'added n'
  end
  def m()
    print 'redefined m'
  end
end

# (Y)
obj.m #=> redefined m を出力
obj.n #=> added n を出力
```

この例では，まず (A) にてクラス C を定義している．(X) の時点では，(A) で定義した内容が利用される．その後，(B) にてクラス C の内容を再定義している．具体的には，メソッド n の追加，およびメソッド m の再定義である．その結果，(Y) での実行結果は再定義を反映したものになる．

実行時に任意のオブジェクトに任意のメソッドを定義可能であり，すでに定義したクラスの再定義が可能であるという動的特性は，柔軟なプログラミングを可能にしている．例

えば，設定ファイルの内容によってクラスの定義を変更するような場合や，長時間不停止実行を求められるデーモンなどのソフトウェアにおいて，バグのあるメソッドを動的に置き換える，などである．しかし，この動的特性は静的解析を不可能にし，高速な処理系の実装を困難にする．この点については次節で述べる．

■演算子オーバーロード 演算子はすべてメソッド呼び出しと解釈される．そのため，任意のクラスで演算子を定義することが可能である．次の例では，新たに定義したクラス C に + メソッドを定義している．そして，(A) の `obj + obj` という式で定義した + メソッドが呼び出される．

```
class C
  def +(o)
    self.to_i + o.to_i
  end
end

obj = C.new
print(obj + obj) # (A) print(obj.+(obj)) とコンパイルされる
```

演算子は他のメソッド呼び出しと同様なので，再定義が可能である．つまり，数値リテラル同士の計算，例えば 1+2 のような式も，`Fixnum#+` メソッドが呼び出されることになり，このメソッドが再定義されることで 3 以外の結果を返したり，副作用を起こすように変更することが可能である．たとえば，Ruby と比較されることが多いオブジェクト指向スクリプト言語 Python[42] では，性能を考慮してこのようなメソッドは再定義出来ないようになっているが，Ruby では例外なく再定義可能となっている．

■例外処理機能 例外オブジェクトを発生させ，呼び出し元に伝播させる機能と，それをとらえる機能を有している．C++/Java 言語でいう `try/catch` 構文にあたる．また，例外によって大域脱出が発生するしないにかかわらず実行するプログラム片を実行させるための機能，Java 言語でいう `finally` に相当する機能を有する．

Ruby における例外処理を利用したプログラムを以下に示す．

```
begin
  # (A)
rescue FooError => e
  # (B)
rescue
  # (C)
else
  # (D)
ensure
  # (E)
end
```

Java でいう `try` 節が (A)，`catch` 節が (B), (C) にあたり，finally 節が (E) にあたる．このプログラムでは，まず (A) に記述したプログラムを実行する．(A) 実行中に例外が発生したら，(B), (C) で例外の補足を行う．まず，発生した例外が `FooError` であれば (B) が実行される．そうではなく `StandardError` であれば，(C) が実行される．例外が発生しなかった場合には (D) が実行される．そして，(B), (C), (D) のどれが実行しても，しなくても，(E) が実行される．

例えば，ファイルを開こうとして，ファイルが存在しない場合にエラーメッセージを出力するプログラムは次のように記述できる．

```
begin
  file = open('foo')
resuce Errno::ENOENT
  puts 'There is no file: foo'
end
```

■ブロック付きメソッド呼び出しとクロージャ　メソッド呼び出し時，ブロックという形で処理を渡すことができる．このブロックは Proc オブジェクトという，関数型言語でいうところの第一級関数として取り扱うことが可能である．

ブロックは，メソッド呼び出し時に次のように記述する．

```
m(){|a|
  # block
}

m() do |a|
  # block
end
```

中括弧もしくは do/end で挟まれた Ruby プログラムがブロックである[*1]．ここでは，m メソッドの呼び出しにブロックを渡している．ブロックには引数を指定することができ，|a|のように記述する．

ブロック内からアクセスできるローカル変数をブロックローカル変数，メソッド実行中はどこでも参照できる変数をメソッドローカル変数と呼ぶ．たとえば次の場合，変数 a はメソッドローカル変数であり，method の実行中はブロック内からも参照可能である．また，変数 b はブロックローカル変数であり，ブロック実行中でのみアクセスできる変数である．

```
def method()
  a = 1
  iter{|b|
    p a # メソッドローカル変数 a を表示
    p b # ブロックローカル変数 b を表示
  }
end
```

渡したブロックは，渡されたメソッド中での yield 文によって起動される．ブロックを呼び出すようなメソッド m の定義を次に示す．yield 文には引数を加えることができ，ブロック引数に渡される．

[*1] なお，中括弧と do/end は，文法規則上の優先順位以外どちらも同じ意味となる．

```
def m()
  yield 'argument'
end
```

この機能は，Scheme などでいう lambda 式を，lambda というキーワードなしで利用するためのシンタックスシュガーと言える．たとえば，上記メソッド m の定義，および m の呼び出しは次のような Scheme の式と同じような意味となる．

```
(define (m block)
  (block 'argument'))

(m (lambda (a) ; block
  ))
```

この「メソッド呼び出し時に 1 つのプログラム片を渡す」というプログラムは，Ruby プログラミングでは様々な場面で利用される．たとえば，配列や範囲オブジェクトなどの各要素への繰り返しは，それらのクラスの each メソッドによって行われる．each メソッドは各要素に対してブロックを起動する．例えば，配列の各要素ごとの繰り返しは [1, 2, 3].each{|e| ...} のように記述する．

単なる N 回の繰り返しにもブロックが利用される．次の例では，3 回ブロックを繰り返す例である．ここでは Ruby で記述した場合と，同様の Scheme プログラムを示す[*2]．

[*2] times という，渡された関数を繰り返す関数が定義されているとする．

```
# Ruby の場合
3.times{
  # このブロックが 3 回実行される
}

;; Scheme の場合
(times 3 lambda((n)
  ;; ここが 3 回実行される
))
```

なお，このブロックを Ruby の Proc クラスのオブジェクトとして取り扱うことが可能である．ブロックから Proc クラスのオブジェクト，そして Proc クラスのオブジェクトからブロックへの相互変換を行うことができる．Proc オブジェクトとして保存されたブロックからは外側の変数を常に参照することができる．つまり，レキシカルクロージャとしての性質が保証される．そのため，例えば次のようなプログラムでは，inc メソッドの引数 n が，生成する Proc クラスのオブジェクトと同様の寿命を持つこととなる．

```
def inc(n)
  Proc.new{
    n+=1
  } # Proc オブジェクトを生成して
end

i = inc(0)
i.call #=> 1
i.call #=> 2
```

■ガーベージコレクタ Ruby では不要となったオブジェクトはガーベージコレクタによって自動的に回収される．そのため，明示的なメモリ管理は一切不要である．

■マルチスレッド 言語レベルでマルチスレッドに対応しているため，ネットワークプログラミングなど，本質的に並行性を有するアプリケーションを容易に記述することができ

表 2.1　Ruby 用ライブラリの一部

dRuby	Ruby 用の分散計算フレームワーク
ERb	HTML などに Ruby プログラムを簡単に埋め込むためのフレームワーク
net/	net/http や net/ftp など各種ネットワークプロトコルアクセス
open-uri	http ごしに，容易に色々な資源へアクセスするための拡張
OpenSSL	OpenSSL の Ruby 用ラッパ
SOAP4R	SOAP を扱うためのライブラリ
REXML	XML のパースや作成
RSS	RSS のパースや作成
test/unit	ユニットテストのためのライブラリ
WEBrick	http サーバフレームワーク
Win32OLE	Windows の OLE（COM）の機能を利用するためのライブラリ
XMLRPC	XMLRPC へのアクセスやサーバの作成
YAML	YAML 文書のパースや作成

る．なお，マルチスレッドプログラムの例は 6 章で改めて紹介する．

■豊富なライブラリ　Ruby には既存のライブラリが豊富にあるため，定型的なプログラミングは容易である．Ruby 用のライブラリの一部を表 2.1 に挙げる．

Ruby は UNIX 上で開発されてきたため，UNIX のシステムコールを呼び出す機能を備えている．そのため，Ruby 上でのシステムプログラミングも容易である．UNIX 以外の，例えば Windows などのシステム上でも可能な限り同じシステムインターフェースを利用できるように整備されている．

同様に，ネットワークプログラミングにおいて標準的な Socket インターフェースへアクセスする手段を標準で有する．例えば，daytime プロトコルへアクセスする Ruby プログラムは以下の 2 行だけで記述できる．

```
require 'socket'
print TCPSocket.open(host, 'daytime').gets
```

また，CRuby では C 言語で記述された，C 言語をインターフェースとして利用するようなライブラリを Ruby から利用可能にするための機能を備えている．この機能で利用可能なライブラリとしては，例えば各種データベースへの接続用ライブラリや画像を取り扱

うライブラリ，ウィンドウシステムを操作するためのライブラリ，データ圧縮を行うようなライブラリなどである．とくに CRuby にはこの C 言語の機能を Ruby から利用可能にするための拡張が容易にする機能を備えているため，すでに多くのライブラリの Ruby 用インターフェースが開発されている．

2.2.2 Ruby の利用用途

Ruby は，その文字列処理の機能の豊富さから，文字列を扱う処理全般に利用されてきた．例えば，UNIX での grep に代表されるようなフィルタ系プログラムや，HTML を扱う CGI プログラミング，ウェブアプリケーション開発である．

とくに，ウェブアプリケーション開発分野では，Ruby を利用したウェブアプリケーションフレームワークである Ruby on Rails[22] が広く利用されており，Ruby の主要用途の一つとなっている．ウェブアプリケーション開発は，現在大きな需要がある分野であり競争も激しいため，開発効率の高い言語および開発環境が求められている．そこで Ruby の開発効率の良さが注目され，よく利用されている．

他にも，UNIX システムプログラミングやネットワークプログラミングを容易に行うインターフェースを有していることから，簡単なシステムプログラミングやネットワーク対応ソフトウェアに利用されている．例えば，バックアップ支援ソフトウェアである pdumppfs[53] や簡易メーリングリスト管理運営ソフトウェアである QuickML[71] などは Ruby を利用して開発されている．

その他，C 言語で記述された機能を，Ruby から利用するためのラッパライブラリを用いて Ruby レベルで組み合わせて利用するような，グルー言語としても多く利用されている．

従来，Ruby の利用用途は，このようなライブラリ処理が処理時間の大半を占め，言語処理系の速度は問題とならないプログラムの開発だった．例えば文字列処理であれば，C 言語で記述した文字列処理機能，正規表現検索機能の処理時間が実行時間の大半を占め，単純なネットワークプログラミングにおいてはネットワーク処理の時間が主なオーバヘッドとなる，という具合である．これらの用途では，他のプログラミング言語で記述してもあまり処理時間は変わらない．

逆に，ライブラリの処理時間にくらべて言語処理系のオーバヘッドが問題となるようなソフトウェアの開発には，Ruby 処理系の速度の問題から Ruby が用いられることは少なかった．このような用途は，例えば数値計算や記号処理が挙げられる．この傾向を顕著に示している例として，言語処理系の局所的な性能を測るためのマイクロベンチマークで用

いられる素数を計算する例が挙げられる．素数を求める計算は簡単な数値計算やメソッド呼び出し，配列へのアクセス時間などの言語処理系が備えるプリミティブな操作の実行時間がプログラム全体の実行時間に大きな影響を与える[*3]．このような，言語処理系自体の性能が影響する計算を Ruby は苦手とする．

2.2.3　Ruby 処理系

長い間，Ruby 処理系は，松本らが開発を進めてきた C 言語で実装した Ruby 処理系（CRuby）のみが存在するという状況であった．しかし，最近はその他の処理系の実装も開発が進められている．本節ではこれら Ruby プログラムを実行するために必要な Ruby 処理系について述べる．

CRuby

CRuby は松本らによって 1993 年より継続して開発が進められてきた Ruby 処理系である．現在の安定版は 2007 年 3 月にリリースされた 1.8.6 というバージョンである．松本のニックネームが Matz ということから，Matz Ruby Interpreter，略して MRI と呼ぶこともある．

長い間，その他の Ruby 処理系実装がなかったため，事実上の参照実装として扱われている．言語仕様を明文化していない Ruby において，CRuby の実装がプログラミング言語 Ruby の仕様であるとされてきた．これは現在も同様である．CRuby は，名前のとおりほぼすべて C 言語によって開発されている．そのほか，構文解析のために yacc 用の文法定義によって構成されている．

メモリ管理はシステムが提供するメモリ管理ライブラリ（malloc/realloc/free）を積極的に利用し，ガーベージコレクタは保守的 GC を独自に実装している．Ruby のスレッド処理機構はユーザレベルで独自にポータブルな方式で実装している．このユーザレベルスレッド方式の詳細については 6 章で詳しく述べる．

CRuby は移植性が高いことも特徴である．実装にはアセンブラなどの記述は出来る限

[*3] マイクロベンチマークは実用的なプログラムの例としては不十分であるが，多くの利用者にとって目に見える計測結果はインパクトがあり，「Ruby はマイクロベンチマークが遅い言語」であるという印象が広く普及している．余談だが，本研究はこれらのマイクロベンチマークの性能を大幅に向上する．そのため，世間一般のプログラミング言語 Ruby に関する「遅い」というイメージを払拭することができる可能性がある．「遅い」というイメージが先行し，Ruby の利用を躊躇してきたプログラマに対し，Ruby 利用のハードルを下げる効果が，本研究の大きな貢献となる可能性がある．

り行わないような配慮が行われている[*4]．実際に，Linux や FreeBSD，Solaris など，ほとんどの UNIX OS のほか，Mac OS X，Windows 95/98/Me/NT/2000/XP/Vista/CE，DOS，BeOS，OS/2 などの環境で動作させることが可能であり実用的な言語処理系となっている．ただし，環境に強く依存した機能は利用できる環境に制限がある．たとえばネットワークでの通信機能などは，ソケットインターフェースがない環境では利用できない．

C 言語で記述された拡張機能を容易に記述することが出来るのも特徴である．C 言語レベルから Ruby 処理系の機能を利用するために，CRuby は C 言語レベルから Ruby 処理系へとアクセスするための，使いやすい Ruby C API を豊富に用意している．Ruby C API を利用すれば，C 言語で記述した関数を，Ruby プログラムからメソッドとして呼ぶことができる．この，Ruby プログラムから呼べる C 言語で記述されたメソッドをネイティブメソッドという．

Java 仮想マシンで同様な拡張を行うには，JNI[20] といった仮想マシンごとに特有の仕様に従った記述が必要になるが，CRuby 用 JNI といえる Ruby C API は JNI よりも決まり事が少なく容易に C 言語で CRuby を拡張することができる．例えば，CRuby では保守的 GC を採用しているので，C 言語上でのオブジェクト参照などを明示する必要はない．また，C 言語の関数呼び出しと同じインターフェースで Ruby メソッドを呼び出す手法や，C 言語レベルで Ruby と同様の例外処理を記述するためのマクロなどが用意されている．

Ruby C API を利用して Ruby を拡張するためのライブラリを総じて C 拡張ライブラリと呼ぶ．C 言語で利用するために提供されている各種ライブラリ（前述したように，各種データベースへの接続用ライブラリや画像を取り扱うライブラリ，ウィンドウシステムを操作するためのライブラリ，データ圧縮を行うようなライブラリなど）を Ruby から利用するために開発された C 拡張ライブラリがすでに多く提供されている．この豊富な C 拡張ライブラリはプログラミング言語 Ruby のグルー言語としての価値をますます高めている．

CRuby の開発方針として，保守性，拡張性，移植性を重視してきたので，ソフトウェアの速度向上の努力があまり行われてこなかった．そのため，Perl や Python などの，Ruby と比較される同種のプログラミング言語の処理系と比べても実行速度は遅い．

とくに，CRuby の低い性能の原因として，抽象構文木を直接辿る評価器という点が挙

[*4] ガーベージコレクションを実装するために，ごく一部でアセンブラが利用されている．具体的には Sparc プロセッサのレジスタウィンドウのフラッシュなどである．

げられる．CRuby は，まず Ruby プログラムを構文解析によって抽象構文木へ変換する．現在の CRuby はこの木構造の抽象構文木を再帰的に辿って実行する [75]．この方式は，中間表現をこれ以上考慮する必要が無く，木を辿る再帰関数を実装することで容易に言語処理系を実現することが出来るという利点がある．しかし，たとえばバイトコードへ変換して実行する処理系と比べて実行速度が遅くなることが知られている．他にも，バイトコード実行系のための良く知られた最適化技法が使えないなど，性能面で問題がある．

JRuby, IronRuby

プログラミング言語用の仮想マシン環境としては，Java 仮想マシン [66] や.NET フレームワーク [32] など，すでに優れた環境がいくつも存在する．最近，これらの既存のプログラミング環境上に Ruby 処理系を構築したものが注目されている．

JRuby[35] は，Java 仮想マシン上に搭載した Ruby 処理系であり，Sun Microsystems 社の Charles Oliver Nutter と Thomas Enebo が中心となって開発を進めている．JRuby は完全に Java 言語で記述されており，Java アプリケーションが動作する環境で動作することができる．最新バージョンは 1.0.1 であり，Ruby 1.8.5 と完全な互換性を実現しているという．

IronRuby[29] は Microsoft の.NET フレームワーク上で動作する Ruby 処理系であり，Microsoft の John Lam らによって開発が進められている．.NET フレームワークの CLR（Common Language Runtime）[33] 上に構築した動的言語サポート用レイヤである DLR（Dynamic Language Runtime）[24] を利用して実装している．まだ正式なリリースはされていないが，近いうちに公開される予定である．

これらの既存のプログラミング言語環境を利用した Ruby 処理系の実装の利点は，Java や C#など，そのプログラミング言語環境用に用意されてきた豊富なプログラム資産を利用することができるという点である．

Ruby 処理系として，たとえばスレッドや GC 機能などは Java 仮想マシンなどの実装をそのまま利用することが可能であるため，新しく開発する必要はない．また，Java や C#用に用意された豊富なライブラリ資産を利用することが容易となる．たとえば，Java 環境，.NET 環境はそれぞれ優れた GUI フレームワークを備えているが，そのフレームワークに Ruby プログラムから容易にアクセス可能となる．同様に，Java などで構築されたアプリケーションに JRuby などを組み込むことにより，Ruby を利用した拡張機能を提供することができる．

この Ruby 処理系実現方式の欠点は，Java 仮想マシンや.NET フレームワークなど，既存のプログラミング言語環境とプログラミング言語としての Ruby とのギャップの解決が

困難，もしくは不可能という点である．たとえば，Java 仮想マシンは Java 言語を実行するために過不足ない機能を実装しており，Java の機能で容易に実現できる範囲の機能を提供するのは問題ないが，そこから逸脱した機能を実現するには大きなコストがかかる．

Ruby 用ではない，他の言語処理系による Ruby 処理系実装で問題になる一例として，ObjectSpace の実現がある．Ruby にはヒープ上に存在する，ガーベージコレクタによって回収されていないオブジェクトすべてへアクセスする ObjectSpace という機能があるが，この機能はメモリ管理を独自に行わず，既存環境に任せている JRuby や IronRuby で実現するのは難しい．そのため，IronRuby や JRuby では ObjectSpace の利用はサポート外，もしくは処理系の実行開始時に利用するかどうか選択するようになっており，もし利用する場合は大きなオーバヘッドを伴って実現する，という手段を取っている．

実行速度の面でいえば，JRuby も IronRuby も CRuby を参考に実装されているため，構文木を辿る単純なインタプリタとなっており，CRuby と同様に実行速度は遅い．ただし，どちらのプロジェクトも，Ruby プログラムを Java 仮想マシンもしくは.NET CLR の仮想機械命令に一部もしくは全部を変換して実行するという作業が進められており，今後は速度改善されていくと思われる．

その他の Ruby 処理系

単独の Ruby 処理系としては，Ruby プログラムによって Ruby 処理系を記述するという Rubinius[39] というプロジェクトが Evan Phoenix を中心に進められている．現在はまだ開発中の段階のため，公式リリースは行われていない．

このプロジェクトでは，Ruby 処理系や Ruby プログラムを C 言語へ変換するコンパイラなどを Ruby で記述するという，Ruby によるメタサーキュラインタプリタを構築しようという意欲的なプロジェクトである．

この方式では，C 言語よりも開発効率の良い Ruby を利用することにより，処理系自体の拡張性やメンテナンス性を向上させることができる．しかし，C 言語へ変換する際にどの程度効率を良く出来るかで，処理系全体の実行効率が決定する．後述するように，Ruby 言語の仕様をすべて満足させるためには大きなオーバヘッドなしに実装することはできない．そのため，例外的な場合の対処を省略するような Ruby のサブセットを定義し，その上でシステムを構築することで実行速度を向上させる必要がある．

その他，Ruby プログラムを Java 仮想マシンのバイトコードへ変換する xruby[60] や，同じく.NET フレームワークの CLR バイトコードへ変換する Gardens Point Ruby.NET Compiler[49] というコンパイラの開発が進行中である．JRuby や IronRuby よりも高速に実行することが期待できるが，それらと同様に Ruby と基盤言語環境とのギャップが問

題となる．

2.3　Ruby 処理系実装の課題

本節ではプログラミング言語 Ruby の，言語処理系開発者から見て実装が困難となる仕様について述べる．特に，高速な Ruby 処理系の開発を困難にする仕様について紹介する．ここで挙げる課題は，本研究が開発の対象とする CRuby（旧 Ruby 処理系）について述べる．また，旧 Ruby 処理系との互換性を保つために必要な Ruby C API を実現するにあたって問題となる点についても述べる．

なお，Ruby の文法は単純に BNF を記述できるものではない [75] ため，字句解析機と構文解析が連携してプログラムの解析を行わなければならず処理系開発の大きな課題となるが，本研究では旧 Ruby 処理系の構文解析器を用いることでこの問題には立ち入らない．

2.3.1　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．

Ruby では変数や式に型を指定しないため，実行前にオブジェクトのクラスを解析することが出来ない．そのため，`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メソッド呼び出し時に検索する必要がある．Ruby プログラムはメソッド呼び出しの繰り返しで構成されているため，この検索を毎回行うのは大きなオーバヘッドになる．

2.3.2　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，Java 言語では，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．このような，数値演算ごとにメソッド探索を行いメソッドディスパッチ処理を実行しなければならないのは実行速度的に問題である．

また，Ruby はガーベージコレクタを標準として備えているため，その性能は，処理速

度に大きく影響する．現在の Ruby のガーベージコレクタは保守的マークアンドスイープ型の GC を採用している．

この方式では，マークフェーズのルートを処理系が管理しているデータ領域およびマシンスタックとしている．Ruby 処理系内部，および Ruby 用 C 拡張ライブラリを C 言語で記述するとき，GC のために Ruby のオブジェクトを指すポインタに対して GC のための特殊な対処をしなくても良い．つまり，Ruby 用 C 拡張ライブラリ開発者は，たとえば JNI[20] のようなガーベージコレクタを正しく動作させるために必要な処理を記述する必要がなく，実際に拡張ライブラリで実現したい処理を簡潔に記述することができる．

2.3.3 ブロックつきメソッド呼び出しとクロージャ

Ruby ではブロックを利用するイディオムが多数ある．そのため，高速な Ruby 処理系を実現するためにはブロックをメソッドに渡すオーバヘッド，およびブロックの起動（`yield` 文の実行）オーバヘッドを小さく抑えることが必要である．

また，ブロックを `Proc` オブジェクトに変換するときには，ブロックからアクセスできるローカル変数などの環境を，`Proc` オブジェクトの寿命と同等とするため，ヒープなどに正しく保存する必要がある．

通常のメソッド呼び出しは，その呼び出しから返ると同時に環境が消滅するので，スタックによって環境を管理するのが効率が良い．しかし，スタックで管理していた環境からヒープなどの GC で管理する領域で正しくコピーしなければならない．環境は，一般的にはいくつかのコンテキストから参照があるので，この操作には複雑なポインタ操作が伴うことがある．

2.3.4 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非常に困難である．たとえば変数の型（クラス）を静的には指定しないことや，クラスの定義，メソッドの定義が実行時に行われ[*5]，また 2.2.1 で示したように，実行時の再定義などが可能（図 2.1）な点などである．

また，実行時に任意の環境で文字列を Ruby プログラムとして評価する `eval` メソッドの存在がある．このメソッドを正しく実現しようとすると，あるタイミングで特定のメ

[*5] 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 は 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 2.1　動的な評価を利用した例

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となる．以下に例を示す．

図 2.2 では，1 度目の `foo()` メソッド実行後にこの実行時再定義機能を利用して `Fixnum#+` という，数値の加算を再定義して減算にした例である．一度目の `foo()` メソッド呼び出しでは 3 を表示するが，2 度目の呼び出し時には数値演算の処理が書き変わっているため出力が -1 となる．このようなあからさまな例は実際に利用されることはほぼ無いが，たとえばメソッドを書き変えてロギングを行ったり，不正な値が渡されてこないかのチェックを動的に挿入したりするメタプログラミングを行うためには必要な，そして強力な機能となっている．

このような Ruby の動的特性はコンパイル時の解析を非常に困難にしている．上記の例では，Ruby の演算子がメソッド呼び出しであり，これが再定義可能という特徴から，数値リテラル同士の演算が再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不変式の除去などの処理を，現在の Ruby の文法を忠実に堅持する限り実現するのは難しい．

```
def change! # Fixnum#+ を書き変えるメソッド
  Fixnum.class_eval %q{
    def +(n)
      self - n
    end
  }
end

def foo
  p(1+2)   # 1+2 の実行結果を出力
  change!
end

foo() #=> 3
foo() #=> -1
```

図 2.2 動的評価を利用して数値演算を書き換えた例

2.3.5 例外処理

旧 Ruby 処理系は抽象構文木を評価する関数 `rb_eval()` が抽象構文木を再帰呼び出しで辿り，Ruby プログラムを実行していくという方式を取っている．そのため，旧 Ruby 処理系では，例外処理をその例外処理対象部分（図 2.3 における (A)，`begin` の節）で毎回 `setjmp` 関数により実行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 2.3 では `raise` メソッドで例外を発生させている）では `longjmp` を実行してマシンスタックの巻き戻し，例外ハンドラ（図 2.3 における (B)）を実行するという手法により例外処理を実現している．なお，例外処理以外にも，ブロックの実行を中断する `break` など，その他の大域脱出の機能もこれを利用して実現している．

この旧 Ruby 処理系での例外処理の実現方式では，例外発生時の例外ハンドラへのジャンプは `setjmp` 関数を呼び出すだけなので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

```
begin
  begin
    # (A)
    raise 'Sample Exception'
  ensure
    # (B)
  end
rescue
  ... # (C)
end
```

図 2.3　Ruby の例外処理機能を用いたプログラムの例

ることは稀であるため，例外処理部分へ入るたびに実行時コストがかかるこの手法は効率が悪い．そのため，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．

2.3.6　C 言語との連携

CRuby の利点のひとつに，C 言語用から Ruby 処理系を操作するための Ruby C API が充実しており，C 拡張ライブラリによって Ruby を容易に拡張できることは先に述べた通りである．

Ruby が標準で備えている文字列操作や正規表現検索などを行うビルトインライブラリは，この仕組みを利用して C 言語で記述されているという点でも Ruby C API は重要である．文字列操作以外にも，リフレクションやスレッド処理など，Ruby 処理系へアクセスする機能も同様の機構を利用して C 言語で記述されている．

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 言語でも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 を整備してこれらの機能をサポートしている．

例えば，C 言語の関数で記述した機能を Ruby から利用するためには図 2.4 のように記述する．ここでは，Ruby レベルで `foo()` メソッドを呼び出すと，C 言語で記述した

```
static VALUE
foo_func(VALUE self)
{
  VALUE value;
  /* ... */

  /* C 関数から bar という Ruby メソッドを呼び出し */
  rb_funcall(self, rb_intern("bar"), 0);

  /* ... */
  return value;
}

void
Initi_foo()
{
  rb_define_method(rb_cObject, "foo", foo_func, 0);
}
```

図 2.4 C 関数を Ruby 処理系へ登録するための手続き

関数 foo_func() を呼び出す場合を考える．また，C 言語関数 foo_func() は Ruby の bar() メソッドを呼び出す必要がある，という状況を考える．

まず，実際に行いたい処理を C の関数として記述する（例では foo_func()）．foo_func() の記述の中では，rb_funcall() という Ruby C API を利用して Ruby のメソッドである bar() を呼び出す処理を実現している．rb_funcall() 関数には，レシーバ，呼び出したいメソッド ID，渡す引数の数（0）を渡している．メソッド ID を得るために，C 文字列を Ruby 内部で利用する ID 型へ変換するために rb_intern() 関数を利用している．

その後，Init_foo() という登録関数を用意し，その中で rb_define_method() という Ruby C API を利用して，Ruby レベルでのメソッド foo() を C の関数 foo_func()

と関連付ける．Init_foo() は C 拡張ライブラリをロードした時点で自動的に呼ばれる．Ruby レベルでは foo() と呼び出すことで，C の関数 foo_func() が呼ばれる．そして，rb_funcall() 関数によって Ruby のメソッド bar() が呼ばれる．

この例は，Ruby のメソッドが C 言語の関数を呼び出し，その関数が Ruby のメソッドを呼び出す，という再帰構造となっている．旧 Ruby 処理系は，抽象構文木を C 言語で記述した評価関数で再帰呼び出しする構造から，setjmp/longjmp を用いることで Ruby メソッドと C 関数の入れ子状態でも例外処理機構を実現することができたが，この再帰呼び出しによる実行方式を仮想マシン方式など，他の実行方式に変更する場合問題となる．

例では示さなかったが，rb_ensure() などの Ruby C API を利用することで，C 関数内で Ruby レベルの begin，rescue，ensure などと同等の例外処理機能を利用することができる．

すでに多くの Ruby C API を利用した C 拡張ライブラリが存在するため，互換性を維持するためには Ruby C API のインターフェースを維持する必要がある．しかし，Ruby C API は旧 Ruby 処理系の抽象構文木を辿る実装に依存しているため，処理系の構造を変更しても十分に互換性を保つのは大きな課題となる．

2.3.7　スレッドの実現

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 資源を有する並列計算機上で並列に実行するにはオペレーティングシステムの提供するスレッド機能を使う必要がある．しかし，旧 Ruby 処理系は移植性を考慮して独自にユーザレベルスレッドを実装しているため，ひとつの Ruby 処理系上でのプログラムの並列実行ができない．これは，今後ますます普及するだろう並列計算機で並列実行が出来ないため問題である．

また，C 拡張ライブラリを構成するネイティブメソッドは，旧 Ruby 処理系での逐次実行に依存しており，スレッドセーフとするための処理を一切行っていない．そのため，ネイティブメソッドを並列に実行することができないという問題がある．

2.3.8　課題のまとめ

本節で述べた Ruby 処理系開発における課題を次にまとめる．

- プリミティブ型がない言語仕様
- 動的特性により静的解析が困難な言語仕様

- オブジェクト指向機能の実現
- ガーベジコレクタの実現
- ブロック・クロージャの実現
- 例外処理機能の実現
- C 言語との連係機能の実現
- Ruby スレッド処理機構の実現

プリミティブ型がない言語仕様，静的解析が困難な言語仕様といった Ruby の動的特性は実行前の静的解析を不可能とするため，とくに高速な処理系開発を困難にしている．オブジェクト指向機能の実現，ブロック・クロージャの実現は一般的なオブジェクト指向言語，関数型言語に共通の実装上の課題である．そして，互換性を維持するために必要な C 言語との連係機能の存在は，効率的な例外処理機能の実現，および Ruby スレッド処理機構の実現を困難にしている．

効率的なガーベージコレクタの実現は Ruby 処理系開発の大きな課題であるが，本研究では現在の CRuby の実装をそのまま利用し，この問題には立ち入らない．Ruby における効率的なガーベージコレクタの研究については木山等の研究 [80, 81] がある．

2.4 まとめ

本章では，本研究が対象とするプログラミング言語 Ruby の主な特徴を述べた．そして，現在利用されている CRuby，JRuby などの Ruby 処理系について，その実装とそれぞれの開発方針についてまとめ，その得失を論じた．また，現在広く利用されている CRuby の処理系が，構文木を単純に辿るという評価器の構造を取っているために性能が悪いという点を指摘した．

さらに，Ruby 処理系開発を行う上での課題となる仕様や本研究で対象する CRuby の実装に関する問題点を挙げた．とくに，Ruby 処理系の高速化を妨げる要因について，Ruby の動的性質による静的解析の困難さや，実用的なソフトウェアの実現のため，互換性を維持する必要があり，そのためには C との連携を行うための Ruby C API の実現が必須であるが，Ruby 処理系の実装に制限を加えることを述べた．

次章以降で，本章でまとめたプログラミング言語 Ruby とその処理系を，どのように高速な処理系として実装していくかについて，その各種手法を検討し実装していく．

第 3 章

Ruby 用仮想マシン YARV

3.1 はじめに

前章ではオブジェクト指向スクリプト言語 Ruby を紹介し，現在広く利用されている処理系である CRuby（旧 Ruby 処理系）について，なぜ性能が悪いのか，そして開発にはどのような課題があるかについて述べた．とくに，木構造の抽象構文木を再帰的に辿る単純なインタプリタであるため，Ruby プログラムの実行時間が長くなるという問題があることを述べた．

本研究では，これらの問題を解決する手法を検討し，実際に Ruby プログラムを高速に実行するための処理系である YARV: Yet Another RubyVM[45][*1]（以降 YARV）を開発した．

YARV は Ruby プログラムの実行時間を短縮することを目的として設計したスタックマシンモデルの仮想マシンである．まず，Ruby プログラムを YARV バイトコードへ変換し，そのバイトコードを仮想マシンで実行する．仮想マシンには高速化手法を Ruby 文法の意味を保つように適用した．

本章では，YARV のアーキテクチャについて，その構成を述べる．YARV に搭載した，Ruby プログラムを高速に実行するための最適化手法については次章で述べる．

[*1] YARV: Yet Another RubyVM の名前の由来を述べておく．開発当初（2004 年元旦）には，いくつか Ruby の VM として知られていたが，しかし不完全であったり効率がよくないような実装がいくつか存在した．そのため，Yet Another という名前を選んだ．また，パーサジェネレータである YACC が大変広く利用されていることにあやかり，自分の開発する処理系が事実上標準になれば，という思いを込めて名付けた．また，Google などの検索エンジンでユニークな名前として検索可能であることにも注意した．

3.2　YARV の概要

前章において，旧 Ruby 処理系は抽象構文木を再帰的に辿る単純な処理系であることが性能上大きな問題点であることを述べた．そこで，そこで，本研究では抽象構文木を直接たどるのではなく，抽象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である YARV を開発した．

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の適用を行った．

本節では YARV の利用方法や処理の流れについて解説し，YARV の概要を述べる．

3.2.1　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）命令列を実行する仮想マシン（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接影響する部分であるため，最適化機構を多く盛り込んだ．

YARV は旧 Ruby 処理系と同様，実行時に構文解析，およびコンパイルを行うため，Java 仮想マシンのように事前にプログラムをコンパイルしておく必要はない．YARV の実行モデルはシンプルなスタックマシンとし，YARV 命令列はスタックを用いて計算を進めるように設計した．

（1）（2）以外の部分においては，旧 Ruby 処理系の多くのプログラムをそのまま流用した．具体的には Ruby プログラムのパーサやオブジェクトの管理（とくにガーベジコレクタ），IO 管理などである[*2]．また，C API を旧 Ruby 処理系と同等の仕様で提供するようにしたので，既存の Ruby 処理系用に開発された拡張ライブラリをそのまま利用出来る．

[*2] ガーベージコレクタについては，並列実行をサポートするためにいくつかの修正を行ったが，その変更については 7 章で詳説する．

3.2.2 YARV による Ruby プログラムの実行方法

Ruby プログラムのユーザから見た YARV の利用方法は旧 Ruby 処理系と同様にインタプリタとして利用できる．つまり，ruby コマンドにコマンドライン引数としてプログラムのファイル名および必要な引数を渡して実行する．たとえば，コマンドラインで `ruby foo.rb 1 2` として実行すれば，Ruby プログラムを記述したファイル `foo.rb` にコマンドライン引数 `1 2` を渡して実行する，という意味になる．そのほかのコマンドラインオプションや利用方法も旧 Ruby 処理系と同様である．

Java 仮想マシンとクラスファイルの関係のように，事前に Ruby プログラムをバイトコードにコンパイル（プリコンパイル）して外部ファイルに出力しておき，実行時にその外部ファイルを読み込んで実行するという手法も考えられるが，まだその機能を実装していない．Ruby はインタプリタとして利用されてきたため，事前コンパイルを必要とする機能はあまり多く利用されないだろうと見積もったためであるが，もう変更することが無いようなライブラリ等を事前コンパイルしておき，コンパイル時間を削減するのは有用と思われるので，今後の課題である．

3.2.3 YARV の処理の流れ

処理系内部では図 3.1 のように実行される．

まず Ruby プログラムを構文解析器によって抽象構文木へ変換する．そして，抽象構文木をコンパイラによって YARV 命令列に変換し，仮想マシン（評価器）へ渡す．仮想マシンでは命令列を命令ごとに解釈実行して Ruby プログラムを実行する．なお，比較のために旧 Ruby 処理系でのプログラムの実行フローを図 3.2 に記す．旧 Ruby 処理系では，抽象構文木を直接実行していることがわかる．

3.2.4 YARV の計算モデル

仮想マシンを実装するにあたり，仮想マシンの計算をどのようなモデルで行うかが問題になる．

仮想マシンの計算モデルは大きくわけてスタックマシンモデル，およびレジスタマシンモデルが知られている．スタックマシンモデルはスタックを計算領域として利用して計算を進めていく．この計算モデルは，Java 仮想マシン [66] やマイクロソフトの.NET[32]，古くは Pascal 用 P-Code 仮想マシン [38] などで広く採用されている．スタックマシンで

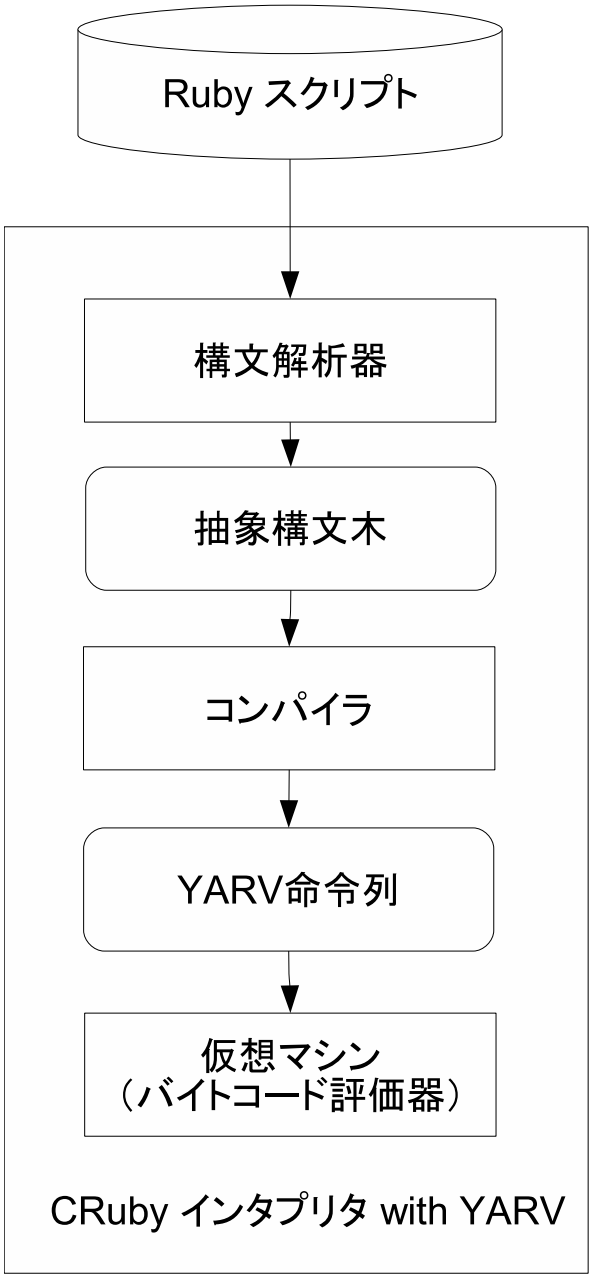

図 3.1　YARV での Ruby プログラムの実行フロー

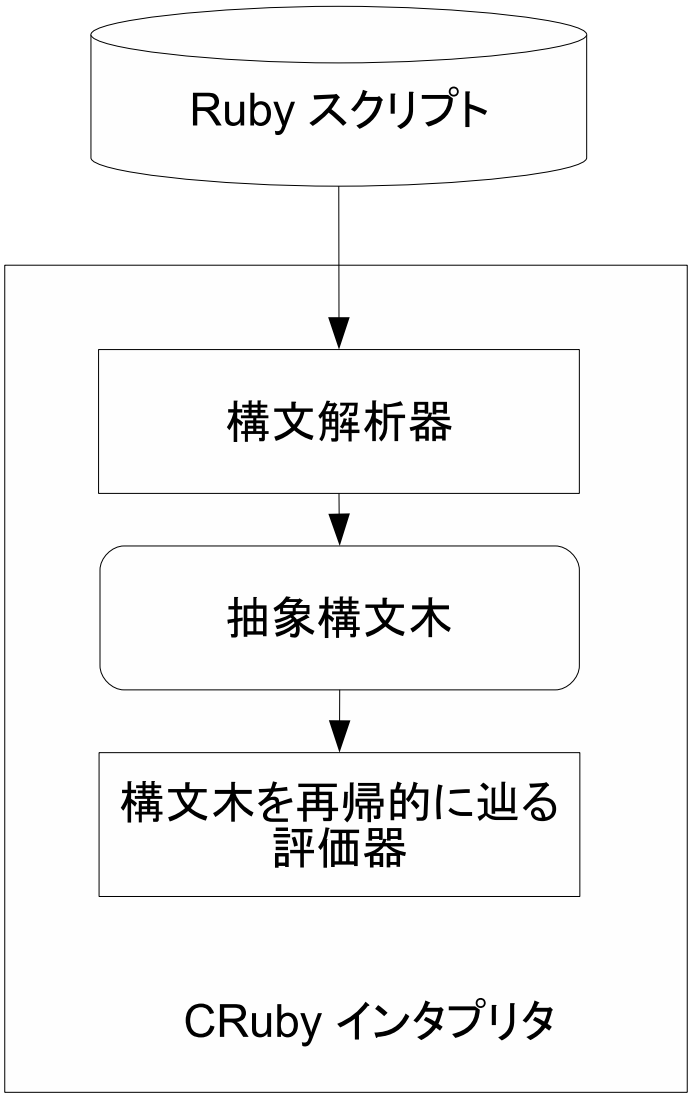

図 3.2 旧 Ruby 処理系での Ruby プログラムの実行フロー

は，各命令は計算対象がスタックトップにあるため計算対象を指定する必要がなくコンパクトである．しかし，計算に必要な値をスタックに設定する必要があるため，冗長となり命令数は増える．

レジスタマシンモデルは現代のプロセッサが利用している計算モデルで，有限個のレジスタという計算領域を利用して計算する．レジスタマシンモデルを採用している仮想マシンは Parrot[54] や Lua[26] がある．レジスタマシンの各命令は，計算に利用するレジスタを指定する必要があるため 1 命令の長さが大きくなる．しかし，計算対象を陽に示すことで少ない命令数で記述できることが知られている．ただし，全体の命令列のサイズは，

各命令のサイズが大きいため，スタックマシンよりも大きいことが多い．

Shi らは仮想マシンの実装モデルとして，スタックマシンとレジスタマシンがどちらが適しているかについて，実行速度の面から比較している [47]．評価によると，レジスタマシンモデルが，スタックマシンモデルと比べて 47% 効率的だとしている．これは，(R-a) スタックマシンよりもレジスタマシンのほうが命令数が少なくなるので，仮想マシンのオーバヘッドの 1 つである命令ディスパッチの機会が減少したためである．

その他，レジスタマシンの利点を考えると，(R-b) 計算をレジスタごとに行うため命令レベル並列性を抽出することが容易である，(R-c) 物理プロセッサと同じ計算モデルなので，仮想マシン命令をさらにコンパイルする Just-In-Time コンパイラの適用が容易となる，(R-d) 既存の機械語コンパイラ用に開発された最適化技術が適用可能，というものがある．

スタックマシンの利点は，(S-a) 抽象構文木からの変換が容易であり，コンパイル時間が短くなる，(S-b) 命令列自体のサイズが小さくなる，という点が上げられる．

Ruby において，これらスタックマシン，レジスタマシンの得失について考察する．

まず，(R-b) については，命令レベル並列性を活かすような，粒度の小さい並列性を活用することは現在の計算機環境では難しいため利点とはならない．(R-c) は，コンパイルする際にはレジスタマシンでもスタックマシンでも再度命令列を解析する必要があるため，とくにレジスタマシンのほうが有利というわけではない．この解析がレジスタマシンのほうが容易に行うことが出来る可能性はあるが，この解析をプログラム開始時に行うか（レジスタマシン），実行時コンパイルのタイミングで行うか（スタックマシン），の違いとも言える．そのため，とくに利点とは言えない．(R-d) については，深い解析が必要となるため，コンパイル時間が増大する可能性がある．とくに，Ruby はインタプリタなので，コンパイル時間に対する制約は厳しい．そのため，(R-d) の利点を活かすためには，プログラムの実行時間がかかる部分に対して，とくに最適化を行う適応型最適化 [3] と組み合わせるなどの工夫が必要となると思われる．しかし，適応型最適化ではネイティブコードを出力することが多いため，仮想マシン用命令にレジスタマシンを (R-d) のために利用する必要はない．

スタックマシンの利点である (S-b) については，現状ではネットワーク上でのバイトコードのやりとりなどは考えておらず，省メモリ化については本論文の主旨ではないため高速な仮想マシンにおける利点とは言えない．ただし，今後の解析において命令列のキャッシュミスが確認できた場合，この性質は利点となる可能性がある．

(R-a) については，高速な仮想マシンの実装方式として魅力的な利点である．しかし，Ruby ではプリミティブ型がなく，単純な数値計算でもメソッド呼び出しが行われるとい

表 3.1 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）

う点でスタックマシンが有利であるといえる．メソッド呼び出しではレシーバや引数の値をスタックに積むことで呼び出すメソッドに対して値を渡すが，Ruby においてはこの操作が他言語用の仮想マシンよりも多くなる．つまり，スタックへのアクセス頻度が Shi[47] らの対象とした Java や OCaml といった言語と比べて多い．レジスタマシンによって，本質的な計算部分は減っても，スタックへ格納する命令が増えてしまい，レジスタマシンモデルにした意味がなくなる．

この定性的な考察の結果，YARV ではスタックマシン型の仮想マシンを採用することが妥当と判断した．また，(S-a) の，コンパイルの容易さによる時間短縮もインタプリタとして構成する YARV にとっては大きな利点である．

3.3 YARV 基本命令セット

YARV は仮想マシンの計算モデルをスタックマシンモデルとしたため，命令セットもスタックを基本に計算するように設計した．命令セットは大きく分けて基本命令と最適化用命令を用意した．前者は Ruby プログラムを十分に表現するための命令，後者は基本的には不要であるが，実行速度を向上するために用意した命令である．

本節では YARV 命令セットの基本命令について述べる．基本命令はいくつかのカテゴリ（表 3.1）を定義した．執筆時現在，YARV の基本命令は 55 個定義している．また，このほか次章で詳述する，高速化を行うための最適化用命令も定義した．各カテゴリごとにまとめた命令の一覧を付録 A に掲載したので参照されたい．

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．ただし，後述する最適化のための特化命令を用意した．

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，命令として用意した．たとえば，Ruby では次のようなプログラムを記述することが可能である．

```
if cond then
  def foo()
    doFoo()
  end
else
  def foo()
    doBar()
  end
end
```

このプログラムでは条件 cond によって定義するメソッドが変わる．Java 言語などではコンパイル時にすべてのメソッドがほぼ決定するため，静的な解析が可能だが，Ruby の場合はこのように実行時にメソッド定義，クラス定義を変更することができるため静的な解析が困難である．そのため，例えば上記の例では if の then パート，else パートの両方にメソッド定義命令が含まれるようなコードが生成される．

Ruby プログラムではメソッド呼び出しが多用されるため，メソッドディスパッチを行う YARV 命令である send 命令がもっとも大事な命令となる．高速化のためにはこの命令の実行コストをいかに小さくするか，もしくはこの命令の実行頻度をいかに減らすかが課題となる．具体的な send 命令の高速化手法は次章で述べる．

ここで，いくつかの Ruby プログラムから YARV 命令列へのコンパイル例を示す．たとえば Ruby プログラム a=recv.method(b) は，次の YARV 命令列（アセンブラ）にコンパイルされる．

```
# a = recv.method(b) のコンパイル結果
getlocal recv   # ローカル変数 recv を push
getlocal b      # ローカル変数 b を push
send :method, 1 # 引数 1 つでメソッド呼び出し
setlocal a      # ローカル変数 a に pop & set
```

なお，この YARV アセンブラは，各行に**命令** [**オペランド** 1[, **オペランド** 2...]]

として表記する．行頭が:で始まっていれば，その行はラベル行である．# から行末まではコメントである．

スタックマシンなので，getlocal 命令で得られた値はスタックへ積まれる．また，setlocal 命令はスタックから値をポップし，ローカル変数をその値に設定する．メソッドディスパッチを利用する send 命令はオペランドとして渡される引数の数とレシーバ用の値分，つまり引数の数 +1 だけスタックからポップし，メソッドディスパッチ処理を行う．この場合，recv をレシーバ，b を引数にして method という名前のメソッドを呼び出している．

条件分岐を含むプログラム a = a > 0 ? a : a * -1 は，3 項演算子（?:）を利用して a の絶対値を取るプログラムである．これをコンパイルすると以下のようになる．

```
# a = a > 0 ? a : a * -1 のコンパイル結果
  getlocal     a
  putobject    0
  send        :>, 1
  branchunless else_label
# then 部分
  getlocal     a
  jump         end_label
else_label:
# else 部分
  getlocal     a
  putobject   -1
  send        :*, 1
end_label:
  setlocal    a
```

条件分岐命令（branchunless 命令，スタックトップが nil もしくは false でなければジャンプ）と無条件ジャンプ命令（jump 命令）によって Ruby の条件分岐を構成していることがわかる．

3.4　コンパイラ

コンパイラは，まず旧 Ruby 処理系でも利用していた構文解析器を利用して Ruby プログラムを抽象構文木へと変換する．そして，抽象構文木に対しいくつかのパスで変換をかけて YARV 命令によって構成される命令列に変換する．

YARV では，Ruby プログラムを構文解析器によって抽象構文木に変換する．この構文解析は字句解析器と連携して行われる．抽象構文木はコンパイラに渡され，YARV 命令列に変換される．この変換は以下の手順で行われる．

1. 抽象構文木から基本命令列を利用したラベル付き命令列へ変換，および後述するキャッチテーブルの作成
2. 基本命令列から最適化用命令を利用した命令列へ変換
3. 命令列にピープホール最適化を適用
4. ラベルアドレス解決

抽象構文木からラベル付き命令列へ変換するときには双方向リスト構造のデータ構造に変換する．コンパイル中は命令の挿入，削除，順序の入れ替えなどを行うことが多いため，リスト構造が便利である．この構造は最後にラベルアドレス解決が行われるまで利用され，ラベルアドレス解決のときにメモリ上に配列として展開される．

最終的に生成される，命令列を表現する配列の 1 要素は計算機のポインタサイズとした．つまり，いわゆる 32bit CPU であれば 4 バイト，64bit CPU であれば 8 バイトの大きさになっている．YARV 命令列は狭義のバイトコード，つまり命令長が 1 バイトであるような命令列ではなく，ワードコードというような構造になっている．これは，計算機によってはバイトアラインメントよりもワードアラインメントのほうがアクセス速度が高速だからである．

最適化用命令を利用した命令列への変換では，基本命令列へ変換した結果を次章で述べる特化命令，融合命令，スタックキャッシングに対応した最適化用命令へ変換する．また，ラベルアドレスを解決後，ダイレクトスレッデッドコードに対応した環境であれば，YARV 命令を示す番号を命令番地へ変換する．ダイレクトスレッデッドコードについても次章で述べる．

なお，YARV の開発当初はコンパイル時に利用するデータを GC 対象の Ruby オブジェクトとして各命令ごとに割り当てる方式としていた．しかし，数万行の Ruby プログラムをコンパイルすると，コンパイル時に大量の Ruby オブジェクトを生成することにな

り，GC のオーバヘッドが非常に大きくなることがわかった．これを避けるため，コンパイル時には GC 対象となる Ruby オブジェクト生成を行わないようにした．つまり，コンパイル時に利用するメモリ領域は，コンパイル専用メモリ領域として独自に管理することにした．

コンパイル専用メモリ領域を用意する前は，旧 Ruby 処理系の挙動にくらべて 5 倍，上記のような数万行規模で最悪のケース（現実的なプログラムでは考えづらいが，数万行が 1 メソッドである，など）で GC が多発し，100 倍ほど遅くなっていた．しかし，専用メモリ領域を容易することでコンパイル時間が旧 Ruby 処理系に比べて 1.5 倍から 2 倍程度の速度低下で済むようになった．

Ruby プログラムが数万行もあるケースは稀であるため，現在はコンパイルの速度はあまり問題になっていない．大きなプログラムにおいてコンパイル時間が問題になった場合でも，コンパイル結果を保存しておくプリコンパイルなどの手法で十分回避可能である．

3.5 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，処理速度を向上するために，すでに提案されているさまざまな最適化手法 [12] 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　制御フレームポインタ
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，つまり 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 のみにすることも出来る．しかし，経験上，ブロックローカル変数よりもメソッドローカル変数を多く参照することが多いため，それぞれ LFP，DFP を用意した．たとえば，ブロックの深い，環境が深くネストしているプログラム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，クラス定義，ブロック呼び出しでの新しいスコープを展開する際には新しいスタックフレームを生成する．スタックフレームを管理するために，通常の値用スタックのほかに制御フレームスタック（CF スタック）を用意した．スタックフレームを生成するときには CF スタックに新しいエントリを積む．

具体的には，新しいスコープを指す仮想マシンレジスタコンテキスト（PC，SP，LFP，DFP）をまとめて CF スタックへ積む．現在の CF スタックのトップは CFP により指し示す．PC，SP 等のレジスタコンテキストは CFP を経由してアクセス可能となる．メソッド呼び出しから戻る際にはこの CF スタックをポップすることで呼び出し前のレジスタコンテキストを復帰できる．

クロージャを生成する場合，つまり現在の環境をメソッド呼び出しから戻ったあとも参照できるように寿命を伸ばす場合，スタックに格納されていた環境をヒープ上にコピーし，CF スタック上の LFP，DFP を適切に書き換える．ヒープ上に確保された環境は GC によって不要になった時点で回収される．クロージャ化が不要なブロック呼び出しは，値スタック上に環境を保持するためヒープ上へのメモリ確保オーバヘッドは不要である．

CF スタッフを値スタックと別に用意したのは，設計を単純にするためである．クロージャの生成では，環境をヒープへとコピーし，メソッドフレームを辿ってヒープへコピーする前の，値スタックを指し示している DFP，LFP をヒープ上のポインタへ書き換えるという作業が発生する．この作業は複雑なポインタ操作が必要になるため，実装してみると，プログラミングが困難でバグが混入しやすい部位であった．そのため，この部分のプログラミング難度を下げるべく CF スタックを別途用意した．

実装上の工夫としては，値用スタックと CF スタックは連続のメモリ領域として確保するよう設計した．つまり，値用スタックは確保したメモリの先頭からメモリ番地の昇順に，CF スタックは領域の最後からメモリ番地の降順になるようにスタックをレイアウトした．このレイアウトにすることで，スタックオーバーフローのチェックが値スタックのトップを示す SP と，CF スタックのトップを指す CFP の比較だけで済むようにした．つまり，$SP > CFP$ となったらスタックオーバーフローであることを検知できる．この工夫により，値スタックと CF スタックを分けたために生じたオーバヘッドはほぼ無く

なった.

3.6 YARV の例外処理機構

Ruby は例外処理，および `break` や `catch/throw` などを利用した大域脱出を標準でサポートしている．例外処理は，ある領域で発生した例外の種類に応じて適切な例外ハンドラを起動する仕組みである．

YARV では，これらの例外処理を実現するために，キャッチテーブル（例外表）検索による例外のハンドリング，および動的埋め込みタグを用いる例外ハンドリング，それぞれを場合に応じて利用するという方式で例外処理機構を実現した．キャッチテーブル検索による例外処理は，例外が発生しない限り例外処理に関するオーバヘッドがないため高速な例外ハンドリング手法である．しかし，旧 Ruby 処理系では C 言語で記述したネイティブメソッド実行中に例外ハンドリングを行ったり，ネイティブメソッドから Ruby メソッドを呼び出し，VM 関数がネストするという状況が発生するため，例外表を用いる方式だけでは例外処理が実現できなかった．そこで，旧 Ruby 処理系でも利用していた setjmp/longjmp による動的埋め込みタグによる例外ハンドリング手法も利用することで YARV 全体の例外処理機構を構築した．

なお，説明のために大域脱出もあわせて例外処理と呼び，これを処理する仕組みを例外処理機構と呼ぶ．

キャッチテーブルを利用する手法は Java 仮想マシンなどで利用されている例外ハンドリング手法とほぼ同様であるが，Ruby の文法の要求を満たすために例外表を拡張している．動的埋め込みタグを利用した例外ハンドリングでは，ネイティブメソッドと Ruby メソッド混在環境での例外ハンドリングを実現している．

3.6.1 キャッチテーブル検索による例外のハンドリング

YARV では各スコープを表現する命令列ごとに，例外を処理するためのキャッチテーブルをコンパイル時に用意する．表の各エントリには，プログラムカウンタの範囲で起こった例外や大域ジャンプについて，どのように対処するか記述してある．このエントリまとめたものを表 3.2 に示す．

図 3.3 のようなプログラムがあった場合，`rescue` で指示された例外処理を行うため例外表のエントリが 1 つ作成される．エントリは，`type` がそのキャッチテーブルエントリが何をハンドリングするのか，その種類を示す値となり，`start PC` は (a) 地点でのプロ

表 3.2 例外処理用の表のエントリ

type	このエントリが対象とする大域ジャンプの種類を示す値
start PC	対象範囲の開始位置を示すプログラムカウンタ
end PC	対象範囲の終了位置を示すプログラムカウンタ
cont PC	このエントリにヒットした場合に利用する継続 PC
cont SP	このエントリにヒットした場合に利用する継続 SP
cont ISEQ	このエントリにヒットした場合に利用する命令列

```
begin
  # (a)
  foo()
  # (b)
rescue
  # (c)
  bar()
end
# (d)
```

図 3.3 例外処理の例

グラムカウンタの値，end PC は (b) 地点のプログラムカウンタの値を示す．cont ISEQ は (c) 地点から rescue 節が終了するまでの命令列を指し，cont PC，cont SP はそれぞれ (d) 地点でのプログラムカウンタ，スタックポインタの値となる．

実際に例外表をどのように利用するか説明する．図 3.3 のメソッド foo() で例外が起こった場合を考える．例外発生時点での VM のプログラムカウンタは start PC より大きく end PC より小さいため，事前に作成したキャッチテーブルエントリにヒットする．対応する例外ハンドラを起動するには，戻り先を cont PC，戻った時のスタックポインタを cont SP に設定して cont ISEQ の命令（例外ハンドラの命令列）を実行する．cont ISEQ の実行が終了，つまり例外ハンドラの実行が終了したら，戻り先として設定していた cont PC へ処理を移し，プログラムの実行が再開される（プログラムの実行を

```
print(1 +
      begin
        foo()
      rescue
        2    # (A)
      else
        3    # (B)
      end)
```

図 3.4 スタックポインタを調節する必要のある Ruby プログラムの例

(d) 地点から再開する).

もし，ヒットするキャッチテーブルエントリを発見できなければ，CF スタックを遡って命令列ごとにキャッチテーブルの検索を続行していき，例外・大域ジャンプを伝播していく．最後まで発見できなければ，例外を捕捉できなかったというエラーを発生する．

このような表を用いる方式は Java 仮想マシン [66] などで用いられている例外表とほぼ同様である．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．

ただし，Java 等のよく知られたオブジェクト指向言語とは違い，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に注意しなければならない．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．

スタックポインタを調節しなければならない具体的な例を図 3.4 に示す．この Ruby プログラムでは，`foo()` の実行中に例外が発生すれば (A) によって例外ハンドリングされ 2 を返すので，`print` メソッドによって 3 が表示される．例外が発生しなければ (B) によって 3 が返されるので 4 が表示される．

`foo()` 実行中に例外が発生し，(A) の例外ハンドラから復帰するときに，値スタックに積んでいた 1[*3]を無視してスタックポインタをクリア（スタック上に何も積んでいない状

[*3] 正確には，print メソッド呼び出しのためのレシーバオブジェクトも，同様に例外復帰時にスタック上に積まれていなければならない．

態へ変更）するとプログラム的に不整合が生ずる．そのため，キャッチテーブルエントリに `cont SP` を用意してこの不整合を防ぐ．

表を用いる方法は，例外発生時には CF スタックを巻き戻し，キャッチする部分が見つかるまで CF スタックに積まれた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旧 Ruby 処理系よりも不利になる．しかし，例外が発生しない場合は例外処理のための実行コストがかからないという大きな利点がある．例外が発生する頻度は稀であるため，例外処理機構としては本方式のほうが有利である．

3.6.2 動的埋め込みタグによる例外のハンドリング

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．また，C 言語レベルで Ruby メソッドを呼ぶ場合，VM 関数が再帰的に呼ばれるという状態が発生する．これに対応するためには YARV 命令列レベルでの例外発生だけに対応する表引き方式では不十分である．そのため，YARV は，旧 Ruby 処理系で行われている `setjmp`，`longjmp` による実行時コンテキストを実行時に保存する動的埋め込みタグ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 レベルの例外伝搬には表引き，C レベルでの例外伝搬には `longjmp` を行う．

具体的には，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）を動的埋め込みタグとして登録しておく．ネイティブメソッド実行中に例外が発生した場合，`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．

3.6.3 キャッチテーブルと動的埋め込みタグの併用

例外処理機構に命令列ごとに用意するキャッチテーブルを利用する手法と動的埋め込みタグを利用する手法を併用している状況をまとめると，図 3.5 のようになる．図の下方向への矢印はメソッド呼び出し，上方向への矢印は例外の伝播を示す．

この例では，Ruby method 1 から 2, C mehtod 1, Ruby method 2, 3, C method 2, というようにメソッド呼び出しをしていった状態で，C method 2 で例外を発生させた状

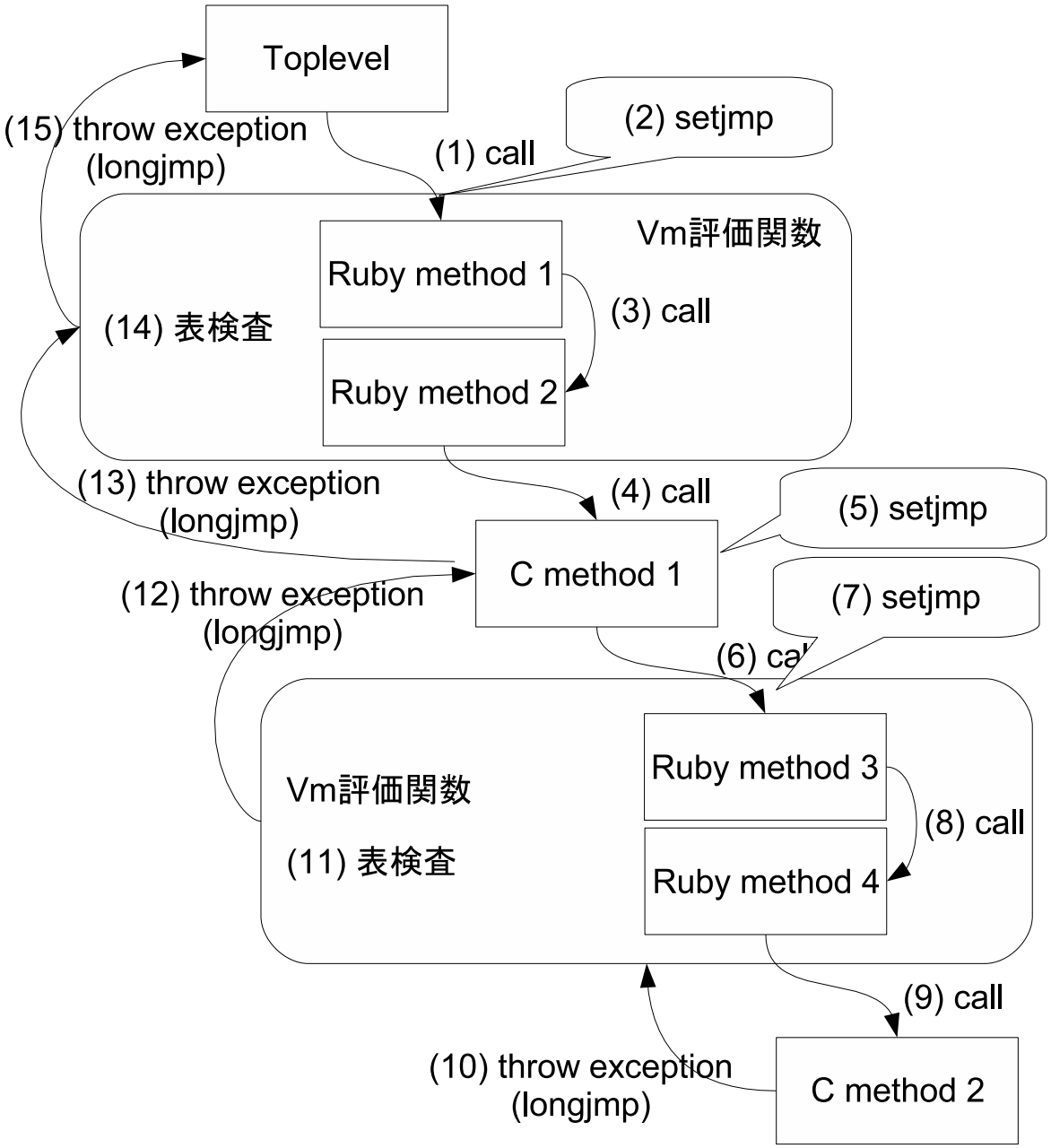

図 3.5 例外処理の実行の様子

態を示している．C 言語レベルでの VM の入れ子関数が発生するため，(2)，(5)，(7) で setjmp を行っており，ここで動的埋め込みタグによる例外ハンドリングに備えている．(3)，(8) の Ruby レベルでのメソッド呼び出しでは，例外表による例外ハンドリングが可能であるため，例外に対処するための実行時の処理は一切必要ない．

図 3.5 では (10) で例外を発生している．まず，(10) の `longjmp` によって，(7) で VM 評価関数開始時に行った `setjmp` の部分へ巻き戻り，(11) で表検査を行う．表検査によって，Ruby method 4, 3 それぞれが対応する例外ハンドラに対応する処理を行い，さらに伝搬する必要があることを確認すると，(12) の `longjmp` で (5) の C method 1 の実行中に `setjmp` を行った部分へ戻る．(5) のようにネイティブメソッドの途中で例外をトラップすることが可能である．例外をそれ以上伝搬する必要がある場合は，(13)〜(15) のように，同様に伝播を繰り返す．toplevel まで例外が伝播された場合は，それに応じたエラー処理を行い，処理系を終了する．

例外伝播の途中でその例外に対して行う例外ハンドラを発見すれば，例外ハンドラへ処理を移し，例外の伝播を終了する．

Java 仮想マシンなどで利用されている例外表のみでの例外ハンドリングは，実行時に例外処理に対応するための処理を行う必要がないため高性能である．しかし，ネイティブメソッド中での例外の補足，および VM 関数の入れ子を実現することが出来ない．過去のプログラム資産を利用するためにはこれらに対応しなければならなかったが，YARV では例外表以外に動的埋め込みタグを併用することで，性能と実用性を同時に実現した．

3.7 まとめ

本章では，実行速度が遅いという問題点のあった旧 Ruby 処理系を置き変える仮想マシン YARV の設計について説明した．

YARV では，旧 Ruby 処理系の速度的な問題点の一つであった抽象構文木を単純にたどる形式ではなく，Ruby プログラムを YARV 命令の並びへコンパイルして実行する仮想マシンを実現した．計算モデルは，Ruby の特徴をふまえた検討の結果，スタックマシンモデルとすることにした．変換する YARV 命令は Ruby プログラムをすべて表現可能な命令セットとして設計し，55 命令となった．

仮想マシンの本体は，仮想マシン用レジスタおよび値用のスタック，そしてメソッドフレーム情報を管理するためのコントロールフレームスタックを利用して計算を行うようにした．コントロールフレームスタックを別途用意したのは，クロージャ生成などでスタック操作が発生したとき，値スタックのみでスタックフレームを管理するよりも単純な作業

で出来るからである．この，スタックに対する操作は多くのポインタ操作が伴うため，この箇所を単純にすることで保守性を向上している．

Ruby C API の設計上，評価器は再帰呼び出しをしなければならないため，例外処理機構はキャッチテーブルを利用する方式，および動的埋め込みタグを利用する方式を併用して実現した．キャッチテーブルを用いる場合，例外処理対象部分を実行するときには一切オーバヘッドが不要となるため，例外処理に関しては旧 Ruby 処理系よりも効率がよい．ただし，例外発生時の補足にはスタックフレームごとに例外表の検査が必要となるためより時間がかかるが，例外発生は名前の通り例外的な事象であるため，本方式が適していると言える．

次章ではこの YARV をいかに高速に実行させるかについて，その最適化，高速化手法を述べる．また，マクロベンチマークを利用した評価により仮想マシン化，および各種最適化がどの程度性能改善に寄与したかについて述べる．

第 4 章

YARV の高速化

4.1 はじめに

Ruby プログラムを高速に実行するために開発した YARV: Yet Another RubyVM について，前章で概要を説明した．本章ではその YARV をどのように高速化したかについて述べ，具体的に評価を行う．

2 章にて，プリミティブ型がない言語仕様，動的特性により静的解析が困難な言語仕様といった，Ruby の動的特性が高速な Ruby 処理系実装の課題であることを述べた．そこで，YARV では静的解析を用いなくても行うことができる実行時高速化手法や，仮想マシンを高速化するための最適化用命令の導入を行い，高速な Ruby 処理系を実現した．

本節で実装した高速化手法が実際にどのように性能に影響を与えるかを評価するためにマイクロベンチマークおよびマクロベンチマークを行い評価を行い，実際に高速化を確認した．

以降，2 節でコンパイル時に行うことができる最適化処理について述べ，3 節で仮想マシンで行う実行時最適化について述べる．4 節で実装時に行った環境に依存した最適化について述べる．5 節で評価を行い，6 節で言語処理系についての最適化技術について関連研究を述べる．7 節でまとめ，今後の課題を示す．

4.2 コンパイル時最適化

動的特性を備える Ruby の言語仕様から，他の言語処理系でよく利用されるコンパイル時最適化はあまり出来ない．しかし，覗き穴最適化などのナイーブな最適化は可能なので，そのような冗長な命令の除去は行う．

```
# 最適化前
  jump L1  # (1)
  ...
L1:
  jump L2  # (2)

# 最適化後
  jump L2  # (1)
  ...
L1:
  jump L2  # (2)
```

図 4.1 覗き穴最適化の例

また，末尾呼び出しやリフレクション操作といった，コンパイル時に解析を行い，その解析結果を実行時に利用して高速化に活用する，という最適化も行った．

4.2.1 覗き穴最適化

YARV のコンパイラは冗長なジャンプ命令の除去に代表される，いくつかの覗き穴最適化を行う．覗き穴最適化の具体的な例を図 4.1 に示す．この例では最適化前には冗長であった (2) のジャンプ命令を，最適化後ではスキップするように変更している．

YARV コンパイラでは，このような冗長なパターンを複数登録しており，コンパイル時に命令列とパターンを照合し最適な命令へと変換する．執筆時現在，6 パターンの覗き穴最適化を登録している．

4.2.2 末尾呼び出しの最適化

言語処理系において，末尾のメソッド呼び出し時のメソッドフレームの保存を省略できるということは広く知られている．そこで，YARV でも末尾呼び出しの最適化を実装した．

この最適化は，コンパイル時にそのメソッド呼び出しが末尾呼び出しであるかどうかを

解析し，もし末尾呼び出しであればメソッド呼び出しを行う VM 命令である send 命令にその旨を示すフラグを設定する．send 命令実行時，このフラグが有効であれば新しいメソッドフレームの生成を抑制する．

末尾呼び出しの最適化を利用することで，たとえば状態遷移などをメソッド呼び出しで気軽に記述することができるようになる．

単純な例であるが，以下のような Ruby プログラムを旧 Ruby 処理系などで実行すると，スタックを使い尽くしてスタックオーバーフローが発生する．しかし YARV で末尾呼び出しの最適化を有効にすることで正しく実行を終了することができる．

```
def func(n)
  func(n-1) if n > 0
end
func(10000)
```

ただし，スタックトレースが異なるという旧 Ruby 処理系との非互換が発生するのでデフォルトではこの機能はオフにしてある．発生するいくつかの非互換とは，メソッドフレームの情報が欠落するため，バックトレースとして得られる情報が減ってしまう，スタックオーバーフローによって失敗していたプログラムが動くようになり，従来はすぐに検知出来ていたエラーが検知できなくなる可能性がある，などである．これらの問題を許容するかどうかは Ruby の言語設計の問題であり，本研究の範囲から逸脱するのでこれ以上は議論しない．

なお，同様に広く知られている末尾呼び出しの最適化は行わなかった．この最適化は，スコープ末尾での再帰呼び出しはジャンプに帰着することができるという最適化である．Ruby レベルではその末尾再起が自分自身のメソッドへのジャンプであるかどうかの判断が，動的に定義が変化するという Ruby の言語的特性から不可能なため，実現することができなかった．

4.2.3 リフレクション機能の高速化

Ruby の send メソッドは Ruby のリフレクション機能の中核を担うメソッドである．このメソッドは，起動するメソッドを実行時に動的に選択することを可能とする．たとえば，send(msg) というプログラムは，変数 msg に格納されているメソッド名のメソッドを起動する．つまり，send メソッドを利用すれば，実行時に起動するメソッドを切り替

えることが可能になる．

send メソッドの動的特性は，メタプログラミングなどにおいてとてもよく利用されるが，通常のメソッド呼び出しに比べ send メソッド起動のオーバヘッドがかかるという問題もあった．

そこで，コンパイル時点で，あるメソッド呼び出しが send メソッドの呼び出しであることがわかったならば，メソッド呼び出し命令に send メソッドであるというフラグをたてておき，実行時には send メソッド起動時に通常のメソッド呼び出しを行うように send 命令を改良した．これで，通常のメソッド呼び出しとほぼ変わらない速度で send メソッドによる間接的なメソッド呼び出しを実現した．

4.3　実行時最適化手法

Ruby の特性上コンパイル時に最適化を行うことができないため，出来る限り実行時コストを低くすることが求められる．YARV ではバイトコード実行型の仮想マシンとして知られている高速化手法を適用し，Ruby プログラムを高速に実行するように設計した．

本章では YARV に搭載した実行時最適化手法について述べる．

4.3.1　スレッデッドコード

バイトコード実行型の仮想マシンにおいて，命令を逐次実行させる手法はいくつかあるが，単純な C の switch 文による命令分岐を用いるとプロセッサの分岐予測器が失敗を繰り返すため，性能的に問題となる．GCC[17] 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 [9, 15] を利用して命令ディスパッチのオーバヘッドを削減した．

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の番地が各命令ごとに異なる場所に分散するため，プロセッサの分岐予測精度の向上が期待できる [11]．

GCC 以外のコンパイラではもっとも移植性の高い C 言語の switch 文による命令ディスパッチを行う．

図 4.2 に (A) switch/case を利用した命令ディスパッチ，および (B) ダイレクトスレッデッドコードを利用した命令ディスパッチプログラムを載せる．(A) に比べ，(B) では一度 (あ) へ戻る処理を省略できることがわかる．(い) の処理は機械語レベルではレジスタ間接ジャンプへコンパイルすることが期待できるため，switch/case による条件分岐より

```
/* (A) switch/case を利用する場合 */
while (1) {
  switch(iseq[pc]) { /* (あ) */
     case INSN_A:
       doA();
       pc += 1;
       break; /* (あ) へ戻る */
       ....
     case INSN_B:
  }
}

/* (B) ダイレクトスレッデッドコードを利用する場合 */
INSN_A:
  doA();
  pc += 1;
  goto *iseq[pc]; /* (い) 次の命令に直接ジャンプする */
...
INSN_B:
  ...
```

図 4.2 命令ディスパッチ手法の比較

も高速であると期待できる.

4.3.2 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.

```
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 4.3　特化命令 opt_plus の擬似コード

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．なお，a，b のクラス（型）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．

opt_plus の実行は，まずレシーバと引数（この場合は a と b）が整数値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値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 4.3）．再定義のチェックは，Ruby の演算子の挙動の再定義を認めるという仕様上必要である．再定義のチェックは再定義が行われているかどうかを示す値とのビット演算による比較だけで済むので，通常のメソッド呼び出し処理に比べて十分計量である．

この特化命令の導入により，単純な演算についてはメソッドフレームを生成する必要がなくなり，とくに Fixnum 同士の数値演算について高速に実行することが可能となった．

利用頻度が高く，メソッド自体の実行コストに比べ，通常のメソッド起動コストが相対的に大きいメソッドについては，このような工夫をすることで演算子の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）[*1]・浮動小数点数値の演算用メソッド（加減乗除，モジュロ，比較）および配列・ハッシュアクセス用のメソッド（読み込み，および設定），配列の長さを返すメソッドを置き換える 11 の特化命令を用意している．これらのメソッドはメソッド起動コストに比べ，必要な計算量が小さく，また出現頻度も大きいた

[*1] 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 4.3 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．

め，有効な最適化といえる．

なお，opt_plus 命令は，実際には前述のとおり整数値であるかどうかチェックした後，整数値でなければ浮動小数点数値についても有限桁整数値と同様にチェックする．このように，複数の型について特化命令を追加していくことができるが，用意する型を不用意に多すると，その他の型のためのメソッド呼び出し処理が遅くなる可能性があるため，ある型における特化命令で代替したいメソッドの Ruby プログラムにおける出現頻度と，得られる高速化をプロファイリングなどを行いよく確認する必要がある．

つまり，Fixnum 同士の加算は高速化されるが，たとえば文字列同士の加算（この場合，連結した文字列が返る）を行う場合には Fixnum かどうかのチェックが無駄に必要となるため，ただの文字列連結を行った場合よりも遅くなる．ただし，文字列の連結処理よりも Fixnum かどうかのチェックは十分高速に行えるため問題にならない．

4.3.3 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命令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 が命令オペランド x を頻繁に利用する場合，命令 A を命令オペランド 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 [78]．

具体的には，スタックに任意の値をプッシュする命令 putobject 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 のような命令オペランド融合は性能向上に寄与すると期待できる．

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 が現れるようなことが頻繁にある場合，C と D の機能を実行する C_D という命令を新しく作る．これを命令の融合という [78]．

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時オーバヘッドの削減を図っている．

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詳述する．

これら融合操作によるインタプリタの最適化は文献 [78] 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 VM の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な影響があった命令を対象として，16 のオペランド融合，12 の命令融合を選択して実装した．現在，選択作業は手作業であるが，プロファイリングを行い機械的に融合候補を提案するような仕組みを今後実現したいと考えている．

具体的には，オペランド融合は (1) ローカル変数にアクセスするための命令における特定インデックスで，JavaVM[66] の `iload_0` 命令にあたる融合，(2) スタックに即値を置く命令において，整数値 0，1，`nil`，`true`，`false` オブジェクトを置く場合のオペランドを融合，(3) 通常のメソッド呼び出しにおける特定の引数の数（0〜3 個）だった場合，そのパラメータを示すオペランドを融合という命令を用意した．命令融合は，(1) スタックに即値・文字列を置く命令の連続を融合 (2) ローカル変数のロードとスタックに即値を置く命令の連続を融合した．

現在用意している融合操作により生成した命令は特定ベンチマークのみの命令列を観察した結果であるため，Ruby プログラム一般に対しては最適な融合操作ではない可能性がある．そのため，後述するプロファイラ，および VM 生成系の機能を利用してさらに適した命令セットを模索する必要がある．

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，融合対象を計算し融合操作を行うという試行を自動で繰り返し行うことで，ユーザが利用す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．

4.3.4　インラインキャッシュを利用した高速化

一度計算した値を再利用するのはソフトウェアの高速化の一般的な手法であるが，YARV でもインラインキャッシュという形でそれを利用している．

具体的には，メソッド探索および定数への参照に利用している．インラインキャッシュはキャッシュが無効になったときの処理が問題になるが，VM 状態カウンタを用いることでキャッシュが無効であるかどうかを判断できるようにした．

本節ではそれぞれのインラインキャッシュの利用法，実現手法について述べる．

インラインメソッドキャッシュを利用したメソッド探索の高速化

オブジェクト指向のプログラミング言語処理系の実装では，メソッド探索の最適化が重要であり，さまざまな方式が検討されている [74]．

Ruby は変数やメソッドの返値の型が明示されないことや，メソッドの再定義が可能であるなどの動的特性により，静的解析によって実際に呼び出されるメソッドの特定が不可

能である．そのため，実行時メソッド探索のオーバヘッドを削減する手法が必要である．

メソッド探索のオーバヘッド削減のための最適化手法として，メソッドキャッシュが挙げられる．メソッドキャッシュは，メソッド検索の結果をキャッシュしておき，次のメソッド呼び出しで利用するという手法である．メソッドキャッシュは実装が容易であり，さまざまな処理系で利用されている．

旧 Ruby 処理系では，インタプリタ 1 つに対し，1 つのメソッドキャッシュ用の表を用意するメソッドキャッシュ手法を用いている．この手法をグローバルメソッドキャッシュといい，そのアルゴリズムを図 4.4 に示す．この方式は，同じセレクタ，レシーバクラスによるメソッド呼び出し時には検索結果のメソッド実体は同じである，という性質を利用している．この方式では 95% 以上のメソッドキャッシュヒット率をもつ [64, 75]．

Ruby 処理系一つに用意されたグローバルメソッドキャッシュ表（コード中の `table`）にはメソッドディスパッチ時のレシーバクラスとセレクタをキー，メソッド定義を値とするエントリによって構成される．メソッド呼び出し時には，まずレシーバクラスとセレクタのハッシュ値をとり，そのハッシュ値に対応したグローバルキャッシュ表のエントリを得る（コード中の (A)）．そして，実際のレシーバクラス・セレクタと，エントリのキーを比較することでエントリが有効であることを確認する．メソッドの定義や再定義が発生したときにはグローバルメソッドキャッシュ表の再構成を行う．

グローバルメソッドキャッシュは，命令の呼び出し場所のローカリティを活用出来ない，大きな表をアクセスするのは時間がかかるなどの問題がある．そこで，YARV では命令列にインラインメソッドキャッシュを埋め込むという手法を採用した．

メソッドディスパッチ命令 `send` のオペランドにメソッドキャッシュ用のオペランドを用意し，その領域にレシーバのクラスとメソッド定義情報を格納する．ディスパッチ時，レシーバのクラスをキャッシュしたクラスと比較し，等しければヒットとする．

もしキャッシュミスが起こった場合は，従来のグローバルメソッドキャッシュを用いて検索を行う．1 つのメソッドディスパッチに注目すると，おおむね前回と同じメソッドが呼ばれるという実行特性から [74]，インラインメソッドキャッシュを用いることで高いキャッシュのヒット率を期待できる．

Ruby プログラムでは，任意の箇所でメソッド定義を変更することができるため，実際の定義との一貫性を保つため，インラインメソッドキャッシュの内容をクリアする必要がある．

たとえば，クラス `A` を継承したクラス `B` があり，メソッド `A#m`（クラス `A` に定義されているメソッド `m`）がある状態で，クラス `B` のオブジェクト `b` に対しメソッド `m` を実行（`b.m()`）した場合，`A#m` を実行し，この定義をキャッシュする．その後，`B#m` を新たに定

```
get_method_with_gmc(recv_class, selector) {
  /* recv_class: receiver's class                      */
  /* selector  : selector                              */
  /* table     : method cache table (global variable) */
  entry = table[HASH(recv_class, selector)]; /* (A) */
  if (entry->recv_class == recv_class &&
      entry->selector   == selector) {
    /* キャッシュヒット */
    return entry->method_body;
  }
  else {
    /* キャッシュミス */
    entry->recv_class = recv_class;
    entry->selector = selector;
    entry->method_body = lookup(recv_class, selector)
    return entry->method_body;
  }
```

図 4.4 グローバルメソッドキャッシュアルゴリズム

義した場合，b.m() は B#m を実行するべきだが，A#m をキャッシュしているため，誤って A#m を実行してしまう．

これを防ぐためには，(1) 再定義時にキャッシュをクリアする (2) 再定義されていればキャッシュを利用しない，の二つの方式が考えられる．(1) では，コンパイルしたすべての命令列の中のキャッシュエントリを走査しクリアする必要があるため，再定義のオーバヘッドが大きい，キャッシュエントリの集合の管理が必要，などの問題がある．グローバルメソッドキャッシュでは，クリアするべき表が 1 つしか無かったので (1) の適用が容易であるが，インラインメソッドキャッシュエントリは YARV 命令列中に散在しているため，そのすべてをクリアしてまわるのは困難である．

そこで，YARV では (2) にて一貫性維持を実現した．具体的なキャッシュアルゴリズムを図 4.5 に示す．まず，仮想マシン自体に *VM* **状態カウンタ** (VMSC: Virtual Machine

```
get_method_with_imc(recv_class, selector, entry) {
  /* vmsc : VM state counter (global variable) */
  /* entry: inline method cache entry          */
  if(entry->recv_class == recv_class &&
     entry->vmsc == vmsc){
    method = entry->method;
  } else{
    entry->recv_class = recv_class;
    entry->vmsc = vmsc;
    entry->method = method =
      lookup(selector, recv_class);
  }
  return method;
}
```

図 4.5 インラインメソッドキャッシュアルゴリズム

State Counter) を設けた．VMSC の値はメソッドが再定義されるごとにインクリメントされる．インラインメソッドキャッシュに書き込む際には，キャッシュ時の VMSC の値もメソッド定義とともにエントリに格納する．キャッシュを参照するときには，現在の VMSC とキャッシュエントリに格納された VMSC の値を比較する．もし値が違うときにはキャッシュエントリを無効として扱うことにより，一貫性を保つことができる．

VMSC を用いる手法 (2) は，仮想マシンにカウンタを 1 つ追加するだけなので，(1) に比べ実装が非常に簡単であるという利点がある．しかし，カウンタは 1 つしか存在しないので，無効化する必要のないインラインキャッシュエントリを無効にするという問題がある．しかし，メソッドの定義・再定義は一般的にプログラムの最初に集中して行われるため大きな短所にはならない．

キャッシュアクセスのオーバヘッドを考えると，(1) にくらべ (2) ではインラインキャッシュの参照時に VMSC のチェックのための分岐がひとつ増えるが，再定義操作は一般的にまれであるため，最近のプロセッサが搭載している分岐予測機能により遅延隠蔽が期待できる．

```
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 4.6　定数アクセスで利用するインラインキャッシュ命令

インライン定数キャッシュ

Ruby の定数の参照は特定の検索パスによって値を検索しなければならない負荷の大きい処理である [75].

Ruby 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 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ように定数アクセスを実装した（図 4.6）．キャッシュを保持し，可能ならその値を返す命令が `getinlinecache` 命令であり，新しくキャッシュをセットする命令が `setinlinecache` である．

Ruby では定数の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，インラインメソッドキャッシュと同様に，VM 状態カウンタ（VMSC）を利用して実現した．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 VMSC の値と現在の VMSC 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．

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のヒット率は十分高い．

4.3.5 静的スタックキャッシング

スタックマシン方式の仮想マシンの最適化技法の一つにスタックキャッシング [10] 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 [10] に詳しいが，ここで簡単に紹介する．スタックマシンはすべての計算がスタックを用いて行われる．しかし，スタックトップ数個のみを利用する場合，スタックアクセスの時間が無駄になる．そこで，スタックトップはアクセスが速い特別なレジスタを用意して計算を進めることで，無駄なスタックアクセスが減り実行効率が向上する．これがスタックキャッシングである．

例えば，$a+b$ という計算を考える．スタックマシンでは，(1) a の値をスタックにプッシュ，(2) b の値をスタックにプッシュ，(3) スタックトップの 2 つをポップし，加算した結果をスタックにプッシュ，という操作が発生する．このとき，通常のスタックを用いた計算では，スタックアクセスが 5 回（(1)，(2) では各 1 回，(3) では 3 回）スタックポインタは 3 回（(1)，(2)，(3) で各一回）更新される．ここで 2 個のスタックトップレジスタ X，Y を用意した場合，(1) は X への代入，(2) は Y への代入，(3) は X と Y を利用した計算となり，一度もスタックへのアクセスが発生しない．

スタックマシン型仮想マシンでは C 言語のローカル変数をスタックトップレジスタとして利用する．アセンブラが利用できる場合はマシンレジスタをスタックトップレジスタに割り当てることも可能である．

ここで，(a) 用意するスタックトップレジスタの数，(b) どのようにスタックキャッシュ状況を管理するか，を設定する必要がある．(a) については，スタックトップレジスタの数が少なければスタックへアクセスの回数が増えるが，多すぎると逆に管理が複雑になる．(b) については，動的スタックキャッシングおよび静的スタックキャッシングという手法のどちらかになる．

動的スタックキャッシングは，実行時にスタックトップレジスタの利用状況を管理し，各仮想マシン命令はそのスタックトップレジスタの状況に応じて挙動を変更するという手法である．たとえば，上述の例での「(1)a の値をスタックにプッシュ」という操作では，すでにスタックトップレジスタに値が格納されていた場合，スタックトップレジスタの一部を実際のスタックへ待避しなければならない．この判断を各命令ごとにスタックトップレジスタの利用状況を確認しながら実行する．この手法は命令数はそのままで済むが，実行時にスタックトップレジスタの利用状況確認のためのオーバヘッドが必要となる．

静的スタックキャッシングは，あらかじめスタックキャッシュレジスタの利用状況ごとに命令を生成しておく手法である．例えば，「(1)a の値をスタックにプッシュ」する命令は，(i) X, Y ともに利用可能だった場合，(ii) X が利用中だった場合，(iii) Y が利用中だった場合，(iv) X, Y が利用中だった場合という 4 種類の状態が考えられるが，それぞれの状態に対処するための命令をあらかじめ用意しておく手法である．コンパイル時にスタックの利用状況を解析し，その状態のための命令に置き換える．スタックトップレジスタを実行時に管理，参照する必要がないので性能的に有利である．しかし，事前にスタックトップレジスタの状態の数だけ命令を用意する必要がある．例えば，仮想マシンの命令が N 命令で，状態数が M であるとき，必要な命令数は $N \times M$ 個になる．

YARV では実行時のオーバヘッドを削減し効率の良い仮想スタックマシンを構築するため，静的スタックキャッシングを採用することにした．状態数が増えると状態ごとの命令を実装するのは大きな手間となるが，本研究では VM 生成系を利用して自動的に生成することで容易に実現した．VM 生成系については次章で述べる．

Ruby や YARV 命令の性質を考慮し，スタックトップレジスタとして 2 個の VM キャッシュレジスタ（a，b）を利用することとした．そして，スタックトップレジスタは 5 状態（$S_{xx}, S_{ax}, S_{bx}, S_{ab}, S_{ba}$）取ることにした．各状態の意味と状態遷移を図 4.7 に示す．S_{xx} はスタックトップレジスタを利用していない状態，S_{ax}，S_{bx} はそれぞれ a，b のみを利用している状態，S_{ab}，S_{ba} はそれぞれ a, b ともに利用している状態だが，スタックの先頭がそれぞれ b，a という意味である．例えば，S_{xx} の状態で値 v をスタックにプッシュしようとする場合，状態は S_{ax} となり，レジスタ a に v が設定される．状態 S_{ab} の場合では，まずレジスタ a の値をスタックにプッシュし，a に v を代入し状態を S_{ba} に変更する．

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，VM キャッシュレジスタが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，VM キャッシュ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．

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．

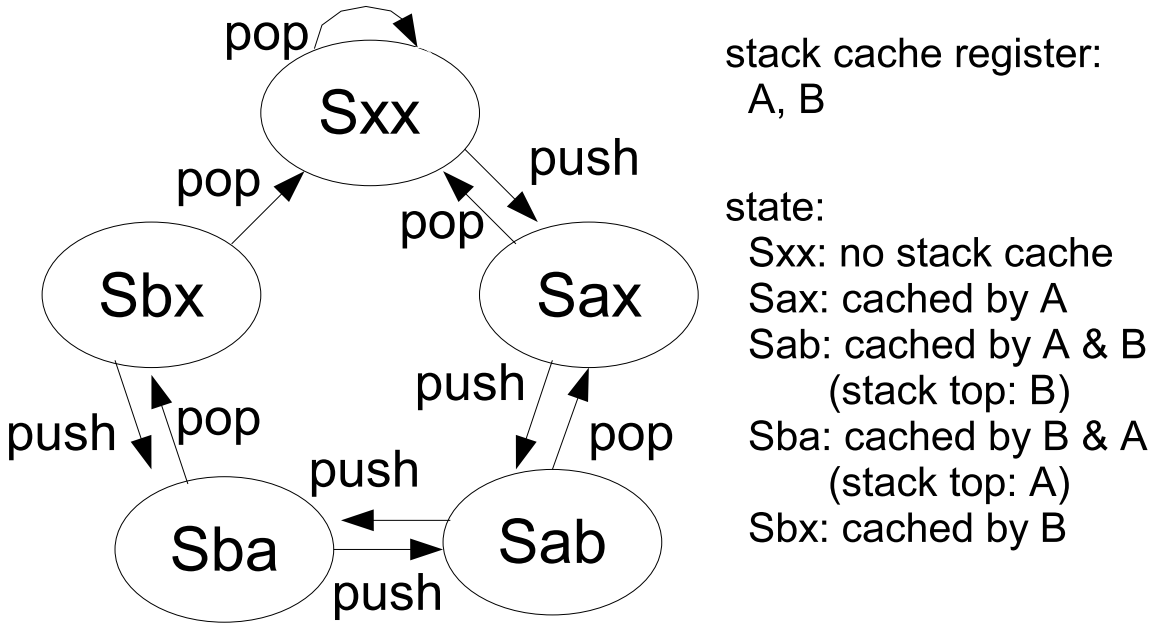

図 4.7 スタックキャッシングの状態遷移図

4.3.6 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 bigram 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大きいため，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．

4.3.7 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，次章で述べる VM 生成系を利用し，YARV 命令列を C 言語プログラム列に変換し，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，各仮想マシン命令の機械語での記述を用意するかわりに，実装コストを節約するため，Dynamic Superinstruction[41] と同様の手法で実現した．

ただし，両コンパイラともに実装が不十分であり，デバッグが十分に出来ておらず多くの Ruby プログラムを実行することが出来ないため，本研究では評価を行わない．

4.4 実装

ここで，YARV の実装についていくつか触れておく．

YARV の実装は基本的に C 言語で行った．次章で述べる 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，MIPS プロセッサ上で動作を確認している．OS は Linux 2.4, 2.6 および Windows2000, XP, Vista での動作を確認している．

コンパイラは GCC3.x，4.x 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 2003，2005 付属のコンパイラで動作していることを確認している..

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 CPU 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 CPU 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．

4.5 評価

本節では YARV の性能評価を行う．

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 (2005-10-12) 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，もっとも速いものを計測結果とした．

各最適化は表 4.1 次の略語を利用する．

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 処理系での実行時間を YARV の

表 4.1　最適化の略語一覧

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

それで割ったものを言う．

4.5.1　各機能の評価

まず，図 4.8 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 4.8 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，各繰り返しごとにブロックの起動のため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である．

`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 4.9）．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，右図では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，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

(1)

```
# loop_times
30_000_000.times do |i|
end

# loop_whileloop
i = 0
while i<30_000_000
  i+=1
end
```

(2)

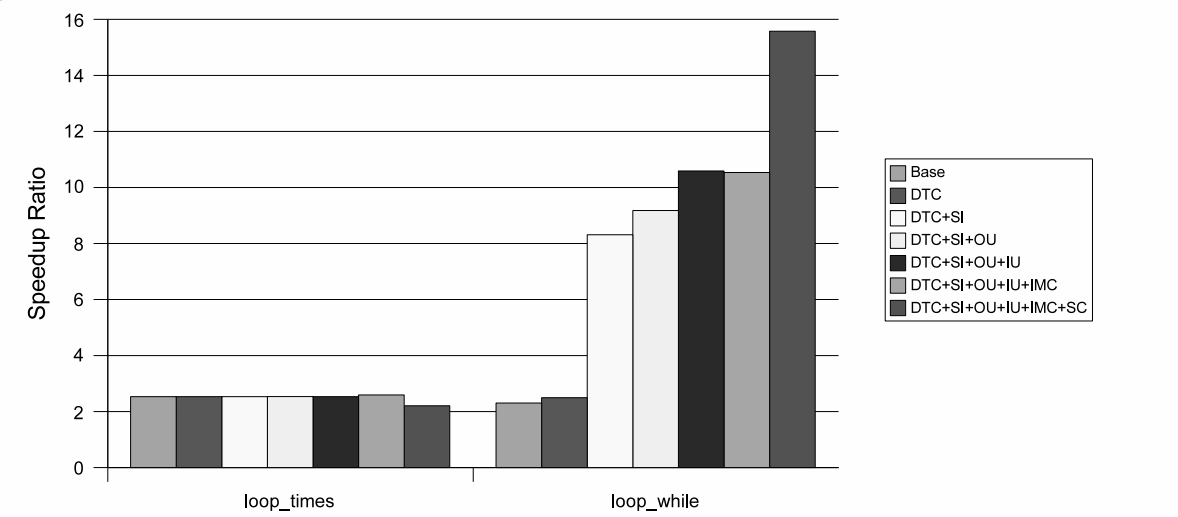

図 4.8 繰り返し実行速度の評価

[73] 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 4.2 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 10 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，現在の処理系より遅くなるのはバックトレース文字列を生成する部分に問題があるためだが，今後必要になる時点

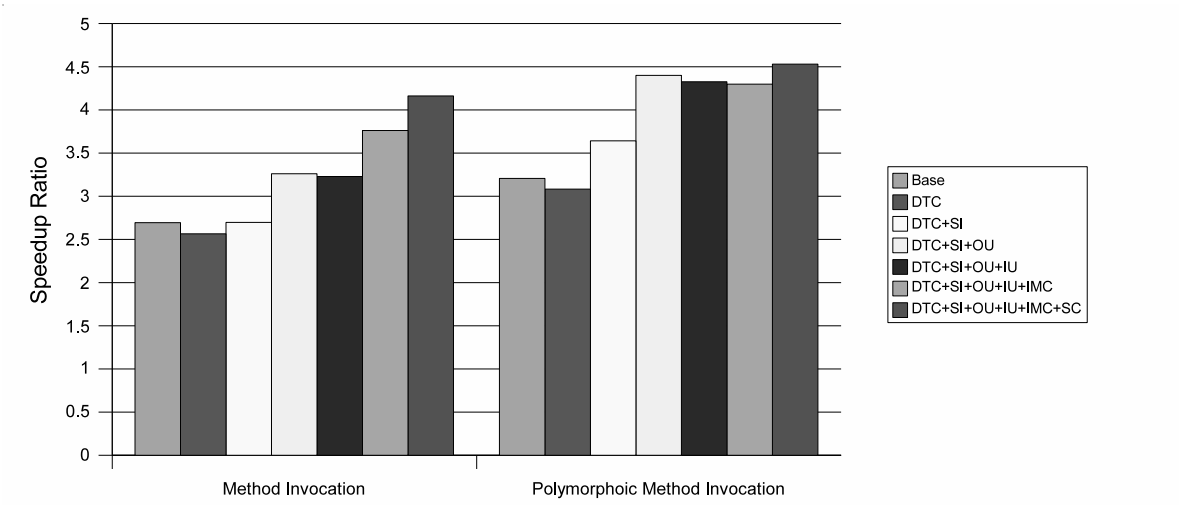

図 4.9　メソッド起動速度の評価

表 4.2　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	10.00	0.85	11.78
例外発生しない			
ensure 節	1.77	0.00	-
rescue 節	4.67	0.15	31.11
例外発生する			
rescue 節	3.65	5.51	0.66

まで文字列の生成を遅延させるなどの工夫を行い解決することを予定している．

4.5.2　マイクロベンチマークでの評価

いくつかのマイクロ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 4.10 に示す．また，x86_64 CPU で動作させた結果を図 4.11 に示す．評価に利用したマイクロベンチマーク一覧を，主な計算時間を消費する処理とともに表 4.3 に示す．

Ackermann や Fib，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，速度向上率も高く，最大で 25 倍ほどの性能向上を示した．

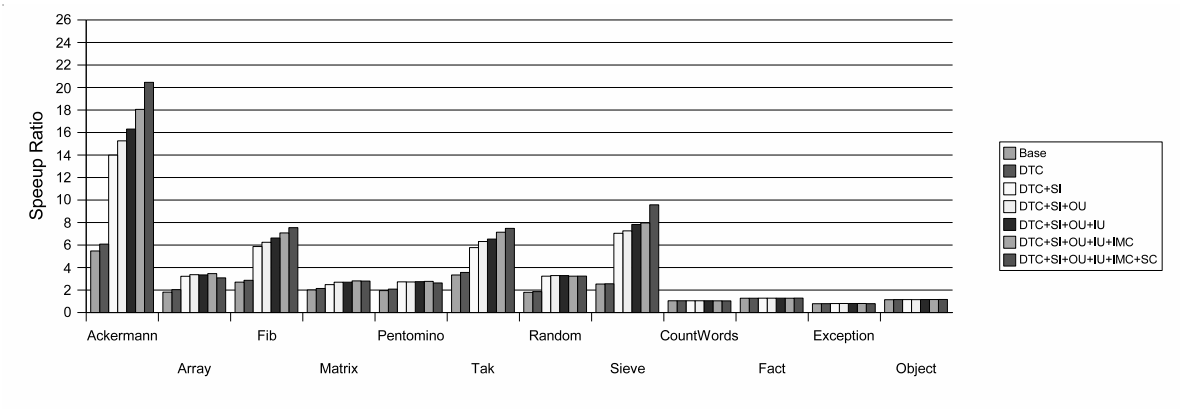

図 4.10 x86 プロセッサ上でのマイクロベンチマークの評価結果

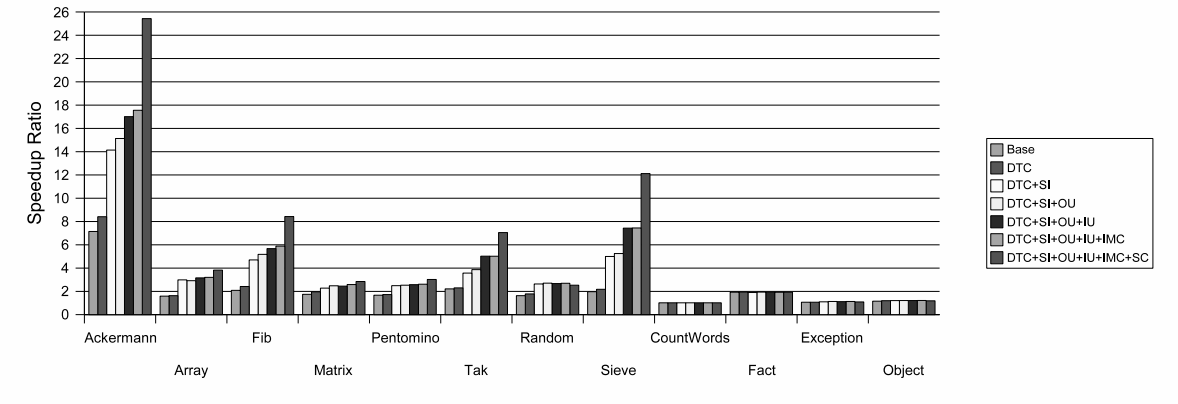

図 4.11 x86_64 プロセッサ上でのマイクロベンチマークの評価結果

Array や Pentomino，Matrix 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，ライブラリ関数の実行時間が VM の実行時間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 はオブジェクトの生成とアクセスにほぼすべての計算時間が費やされている，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である．

ただし，このような VM 以外の計算は 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計算となっており，この部分は現在の Ruby 処理系でも，他の言語処理系と比較して速度的な問題にはなっていない．

表 4.3　マイクロベンチマーク一覧（括弧内は主な処理）

Ackermann	アッカーマン関数（メソッド呼び出し）
Array	配列アクセス（配列アクセス）
Fib	フィボナッチ数（メソッド呼び出し）
Matrix	行列乗算（数値計算，配列アクセス）
Pentomino	ペントミノパズル（配列アクセスほか）
Tak	竹内関数（メソッド呼び出し）
Random	乱数生成（浮動小数点演算）
Sieve	エラトステネスのふるい（数値計算）
CountWords	単語数え上げ（文字列処理）
Fact	階乗計算（多倍長演算）
Exception	例外を大量に発生（例外生成）
Object	オブジェクトを大量生成（オブジェクト生成）

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 では 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 CPU 上での速度向上率よりも，x86_64 CPU 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．

4.5.3　マクロベンチマークでの評価

表 4.4 に示す Ruby で記述された比較的大規模なプログラムを利用してマクロベンチマークとした．この実行結果を図 4.12 に示す．

tDiary はもっとも有名な Ruby アプリケーションの一つで，ウェブ上で日記を管理するシステムである．主な処理はテキスト処理，およびファイル入出力，そして `eval` メソッドなどのリフレクション機構を利用する処理である．トップページを生成する処理を

表 4.4 マクロベンチマーク一覧

tDiary	ウェブ上で日記を管理するシステム（テキスト処理・ファイル入出力）
Scheme	Ruby によるナイーブな Scheme インタプリタ（記号処理）
XML Diff	二つの XML を読み込み，木を辿り diff を取る（テキスト処理・記号処理）
Mandelbrot	複素数計算により，マンデルブロ集合を求める（複素数計算）

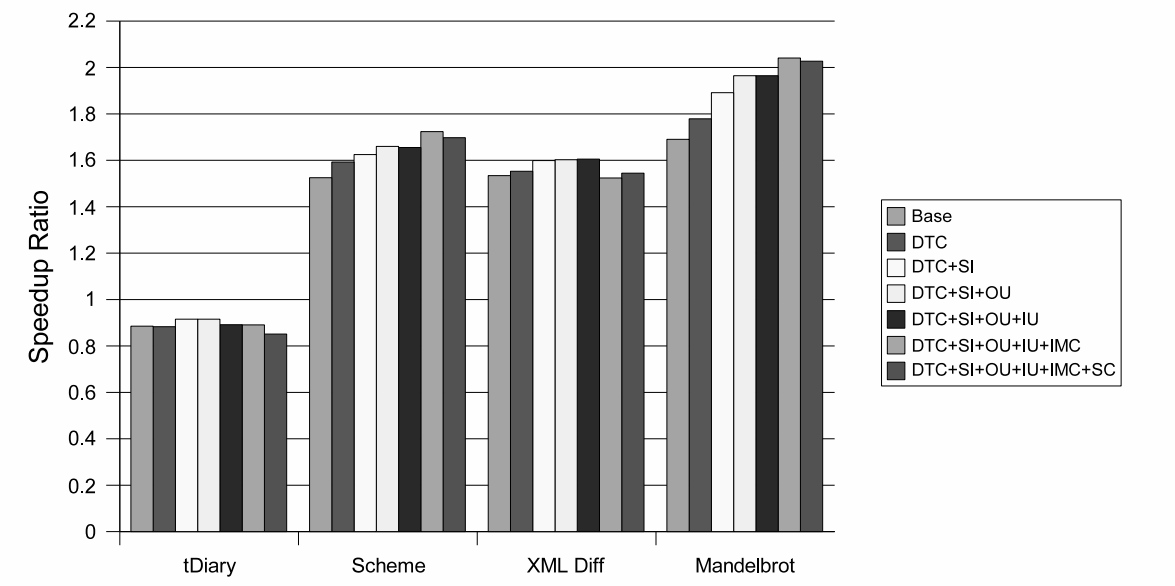

図 4.12 マクロベンチマークの評価結果

繰り返し，その時間を計測した．

Scheme は Ruby で実装した，リストをたどり実行するナイーブな Scheme 処理系である．評価ではここで Scheme で記述したフィボナッチ数を求めるプログラムを実行した．これは記号処理を行う．

XML Diff は Ruby で書かれた XML 処理系を利用して XML 文書を木構造のデータに変換し，トラバースして二つの XML ファイルの差分を求めるプログラムで，主にテキスト処理を行う．

Mandelbrot は Ruby で記述された複素数ライブラリを利用してマンデルブロ集合を求めるプログラムであり，主に複素数計算（浮動小数点計算）を行う．

結果を見ると，tDiary の実行結果は遅くなってしまった．これは，そもそもテキスト

処理，ファイル入出力などのプログラムは VM による高速化が性能向上に寄与しない分野であり，もともと高速化が望めないことと，現状の YARV はまだリフレクション機能などを実装したばかりであり，パフォーマンスチューニングが出来ていないためである．Ruby は tDiary のようなウェブアプリケーションの実行環境としてよく利用されるので，この分野のアプリケーションが高速に実行されるような工夫は急務である．

Scheme，XML Diff はともに 1.5 倍程度の性能向上を実現できた．Ruby が従来速度的に苦手とされてきた記号処理に対し，VM の高速化が効果的であることが確認できた．また，Mandelbrot のような数値計算も 2 倍程度高速化することができた．

大規模なアプリケーション，とくに Ruby のいろいろな機能を利用するアプリケーションではスタックキャッシングが若干性能上不利に働いているが，これは次章で述べるコード量増加による悪影響のためである．スタックキャッシングの適用については今後もプログラムの挙動をさらに詳細に検討し，適用方法を考察していきたい．

4.5.4　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 4.13 にまとめた．Y 軸は実行時間（秒）を示す．

Perl[36] 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 と呼ばれる命令実行型仮想レジスタマシンを用いた実装となる予定である．今回の評価は Perl 5.8.6 を利用した．

Python[42] 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して，最適化はあまり行われていない．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 0.8.5[27] の実行も比較の対象とした．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方式の処理系である．仮想マシンのアーキテクチャは，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なっている．

図 4.13 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 CPU 環境で行いそれぞれ Cygwin 上で動作する各処理系を利用した．

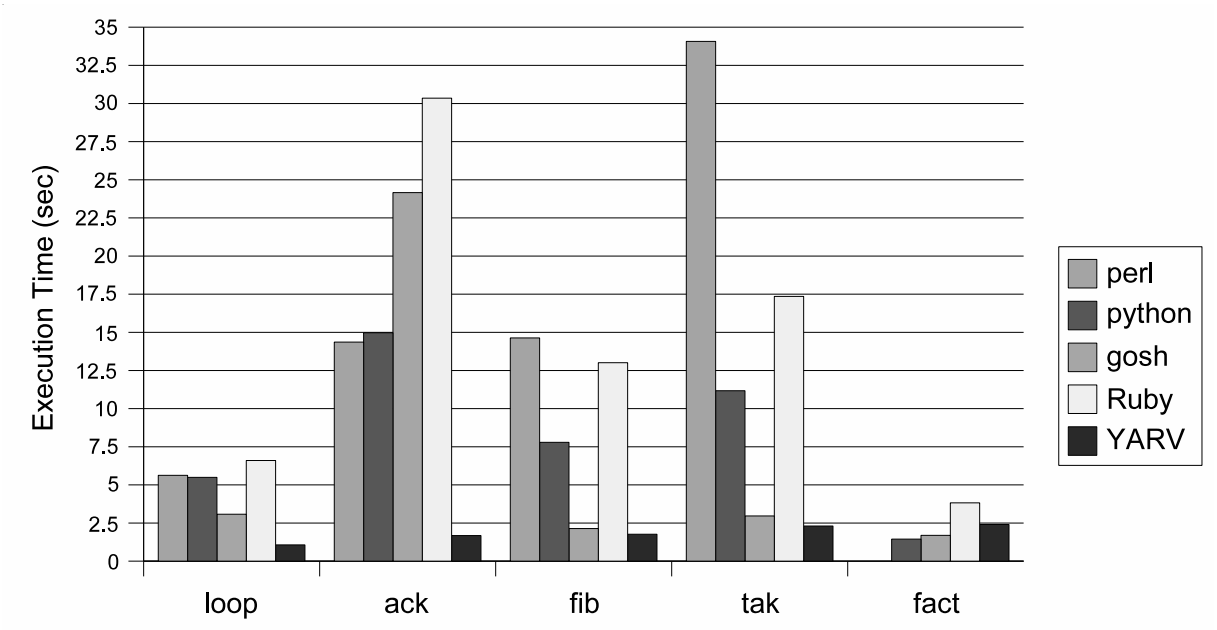

図 4.13 他言語との速度比較

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．

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計算時間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．

4.6　関連研究

Ruby 向け仮想マシンはいくつか提案されている [44] が，JRuby[35] や IronRuby[29] は旧 Ruby 処理系の実装を継承しているため実行速度は速くない．具体的には，構文木をたどる単純なインタプリタとして実装している．また，xruby[60] や Rubinius[39] などのその他の Ruby 用仮想マシンは実装が不十分で Ruby プログラムのほんの一部しか評価できない，という状態である．

ruby2c[7] 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．また，このプロジェクト自体は Ruby のサブセットを対象にしている．

rubydium[28] 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．

文献 [78] 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 では次章で述べる VM 生成系を利用することにより，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，この手法の適用を容易にした．

4.7　まとめ

3 章および 4 章では Ruby プログラムを高速に実行するための Ruby 用仮想マシンである YARV について，その設計と実装方法，高速化手法，そして実際に性能について評価した結果を述べた．

高速化手法については，YARV における各種最適化について述べた．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，旧 Ruby 処理系と比較して高速に実行できることを示した．とくに，VM の性能がボトルネックになっていた記号処理分野での性能向上を確認した．そして，各種最適化を段階的に適用した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 よりもプログラムが高速に実行できることを確認した．

今後の課題として，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．

さらなる性能向上 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用いた最適化手法によって高速化を行っていきたい．
VM の命令ディスパッチにはダイレクトスレッデッドコードを利用しているが，プロセッサの分岐予測には悪影響を与えてしまう．スタックキャッシングを利用することで分岐する箇所を分散する効果を得ることが出来るが，分岐予測をより有効利用するサブルーチンスレッディング [4] や，これを利用してネイティブコードと効率的に融合させる手法 [61, 62] を適用したい．
命令の融合操作も特定のプログラムに関してしか行っていないため，より一般的なプログラムで評価し，Ruby プログラム全般に適した命令セットの追求を行いたい．
また，ガーベージコレクタについては現在の Ruby 処理系の mark & sweep 型の機構をそのまま流用しているが，世代別 GC にするなどの改善が必要である．

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 [23] 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，より高速な Ruby 処理系を目指していきたい．

第 5 章

仮想マシン生成系

5.1 はじめに

YARV の開発では，仮想マシン開発を容易にし，ソフトウェア自体の保守性を高めるために VM（仮想マシン）記述言語および VM 生成系を作成した．作成した VM 生成系は，VM 記述言語で記述された VM 記述を読み込み，コンパイラやアセンブラ，仮想マシン本体のプログラム片を自動的に生成する．

VM 生成系は仮想マシンの最適化命令を半自動生成するためにも利用した．命令の融合操作，静的スタックキャッシングといった機械的に適用可能な最適化に関しては，VM 生成系を用いて必要な命令を半自動生成することで仮想マシンの高速化手法の適用を容易にした．

プログラムの自動生成を利用することで，手作業でコードを追加する際に問題となるバグの混入を防ぎ，YARV 自体の保守性を高めた．また，生成系を利用することで新しい命令の追加が容易となったため，開発初期段階での命令セットの試行錯誤が容易となった．

本章では VM 生成系と，VM 記述言語について詳細を述べる．そして，VM 生成系を利用した最適化命令の生成，およびその他の VM 構成要素の生成について示す．そして，仮想マシン生成系についての関連研究を述べ，最後にまとめる．

5.2 VM 生成系

本節では仮想マシンの構築，および次章で述べる最適化手法を容易に実現するための仮想マシンの生成系について詳細を示す．

仮想マシン，とくに評価器での命令実行はプログラムカウンタ，スタックポインタの増

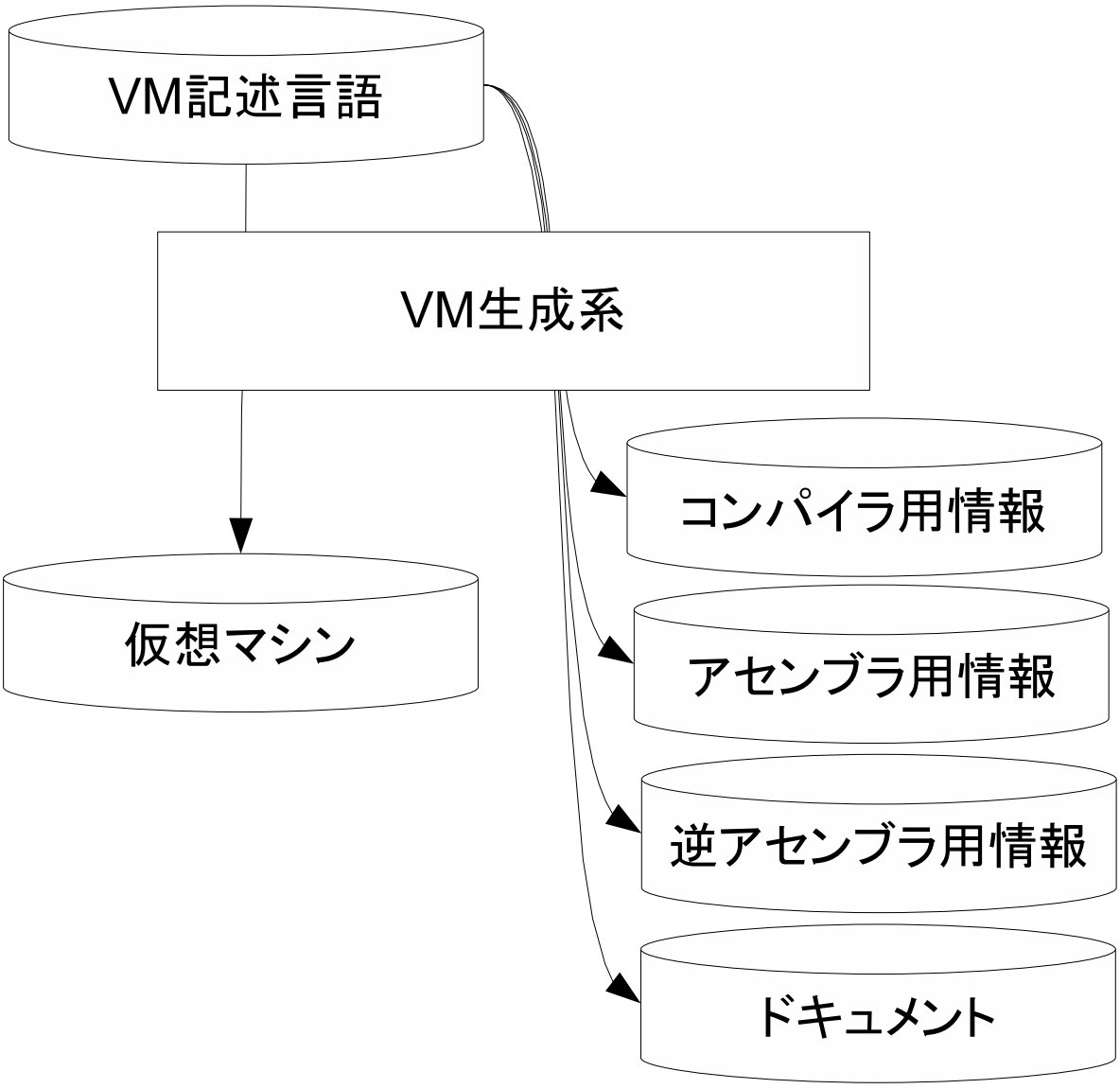

図 5.1　VM 生成系

減など，決まりきった処理を行うことが多い．そのような処理を人間がそれぞれ記述するのはミスが入りこむ余地を生じるだけであり無駄である．また，後述する機械的な変換で可能となる最適化を実現するには，機械可読可能な形で変換のベースとなる命令の処理を記述しなければならない．

そこで，VM 記述言語を設計し，簡単な VM 生成系を開発した．VM 生成系は VM 記述言語で記述された命令の内容を，命令実行を行う評価器，コンパイラのコード片，アセンブラ，逆アセンブラなどのプログラム，ドキュメントなどに変換する（図 5.1）．

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ないため単純である．

5.3 VM 記述言語と生成された VM コード片

YARV の命令は VM 記述言語によってそれぞれ定義する．図 5.2 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[*1]．`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．

VM 記述言語の文法は次の通りである．まず，`DEFINE_INSN` というキーワードで文法定義の開始を宣言する．その次の行で命令の名前を宣言する．この後ろに 3 行，パラメータを記述する括弧が並び，それぞれ命令オペランド，スタックオペランド，命令の返り値（スタックにプッシュする値）を示す変数の定義を行う．変数の定義は C の関数パラメータの定義と同等の記述が可能である．その後，中括弧でくくった部分に，定義した変数を利用して C 言語でその命令が何をするのか，その挙動を記述する．

図 5.2 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はない．

命令オペランド，スタックオペランド，命令の返り値のための変数は，命令の種類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．この可変長引数にアクセスする場合は，VM の値スタックを直接参照することで行う．

これらの情報から VM 生成系では図 5.2 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

[*1] `mult_plusConst` 命令は，本稿での説明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 言語プログラム片

```
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 5.2　VM 記述言語の記述例および生成された C 言語のプログラム片

スタックに格納する処理，および対応する環境では次章で述べる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．

5.4 VM 生成系を利用した最適化命令の生成

本節では，VM 生成系を利用して最適化のための命令が生成できることを示す．具体的には，融合操作した命令，および静的スタックキャッシングを行うための命令を生成する．ただし，不用意に命令を生成すると仮想マシンの規模が大きくなるためプロセッサの命令キャッシュに載らないという問題がある．本節ではこの問題についても考察する．

5.4.1 融合命令の生成

融合命令の作成は，前章で述べた 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 5.3 では，前章図 5.2 で例に出した `mult_plusConst` 命令に命令オペランド `c` 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片を示している．

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 5.5 は，図 5.4 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片を示している．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 5.5 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

```
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 5.3　生成されたオペランド融合命令

```
DEFINE_INSN
dup
()
(int v)
(int v1, int v2)
{
  v1 = v2 = v;
}
```

図 5.4　dup 命令の定義

```
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 5.5 dup 命令と mult_plusConst の融合命令

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 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．

5.4.2　静的スタックキャッシュ命令の生成

YARV では 4.3.5 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 S を開始状態とした I 相当の処理を行う命令とすると，$SC(I, S_{xx})$，$SC(I, S_{ax})$，$SC(I, S_{bx})$，$SC(I, S_{ab})$，$SC(I, S_{ba})$ の 5 命令を新たに用意する必要がある．VM 生成系は I の命令記述に基づき，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 $SC_t(I, S)$ を定義する．これは，命令 I について $SC(I, S_s)$ の次の命令の開始状態が S_e であることを示す．

$SC_t(I, S_s)$ 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 4.7 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．

VM のための C プログラム片の生成時は，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 5.6 に示す C プログラム片に変換される．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冗長なアクセスを含むように見えるが，多くの場合 C コンパイラがこのような冗長なアクセスを除去するため，最終的には効率良い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 $S_i = SC_t(I_i, S_{i-1})$ $(i \geq$

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

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．

5.4.3 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，3×3 で 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を増やすことができるため，実際にどの命令を生成するべきかという点で議論が必要である．

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，合計 875 個の命令がある．現在は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命令数削減の工夫はしていないが，今後は文献 [78] で提案されているようなシステマティックな命令選択あるいは文献 [13] で述べられている，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．

5.5　VM 以外のコード生成

VM 生成系は仮想マシン以外のコード片を生成する．本節では，それら仮想マシン以外で生成するコードについて述べる．

5.5.1　コンパイラ用コードの生成

VM 記述および VM 生成系により，コンパイラはある命令が命令オペランドを何個取るか，値スタックをいくつ消費（ポップ）し，命令終了後にいくつ積む（プッシュ）するかを知ることができる．この情報はコンパイラが命令列を変換するために必要になる．

これらの情報は命令テーブルの情報として出力し，コンパイラはラベルの計算やスタック消費量の計算のために利用する．VM 生成系を用いない場合，これらの情報を命令の追加，変更ごとに別途メンテナンスしなければならなかったが，VM 生成系が自動でこれらの処理を行うことにより，間違いを混入することなくコンパイラと連携することができる．

5.5.2　アセンブラ，逆アセンブラ用コードの生成

VM の命令は VM 生成系によってある数値に割り当てられる．逆アセンブラは，VM 生成系が作成するこの対応表，および上述した命令オペランドの情報を参照して YARV 命令列の情報を逆アセンブルした情報を表示する．

アセンブラは，この対応表を利用して命令の文字列から命令番号を算出し，文字列情報

から YARV 命令列へ変換する．

アセンブラ，逆アセンブラについても他の生成コード片と同様に自動で生成することで実際の情報との一貫性を取ることが容易になる．

5.6 関連研究

vmgen[14] 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[79] 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．

5.7 まとめ

本章では VM 生成系を用いた仮想マシンの構成手法について述べた．

基本的には文字の置き換え操作しか行わない，簡単なプログラム生成系でも仮想マシンの開発を容易にすることができた．とくに，機械的に導出できるプログラムを自動生成することで，人手によるバグの混入機会を削減し，ソフトウェアの保守性を高めることができた．

ただし，最適化命令の半自動生成を利用すると，生成される命令数が爆発し，仮想マシンのコンパイル時間，そしてプロセッサの命令キャッシュミスによる実行時間の増加につながることがわかった．この問題を回避するためには，生成する最適化命令の適切な削減手法が必要であり，VM 生成系にこのアルゴリズムを追加することが今後の課題である．

第 6 章

ネイティブスレッドを用いた Ruby スレッド処理機構

6.1 はじめに

近年，マルチコアプロセッサのコモディティ化に代表される，並列計算機の普及が著しい．しかし，並列計算を行うためのソフトウェア開発は困難であることが知られており，並列計算を行うプログラムを手軽に開発できる手法が求められている．手軽にプログラムを書くことを目標としている Ruby では，プロセスを分散させる並列計算は対応しているが，データの共有についてはプロセス間通信を別途行う必要があるなど，利用が難しい．

Ruby の特長の一つに，言語レベルでマルチスレッド実行に対応しているということが挙げられる．このマルチスレッド機能を用いることで，ネットワークに対応したサーバプログラムなどが容易に記述可能である．旧 Ruby 処理系でのスレッドは並列実行には対応していないが，これを並列に実行させるモデルへ拡張させるのは並列計算を活用させる仕組みとして自然である．そこで，本章と次章にて，Ruby スレッドを並列計算機上で並列実行させるための仕組みについて議論し，その上で開発を行い評価を行う．

プログラミング言語におけるマルチスレッド機能とは，複数のスレッド（命令流）を並行，もしくは並列に実行する機能である．最近の多くのプログラミング言語，たとえば Java[52] や C#[30] において標準でサポートされている．Ruby と比較されるプログラミング言語である Python[42] や Perl[36] においても同様にサポートされている．

Ruby において，スレッドを用いるプログラムは図 6.1 のように記述される．Ruby プログラムにおいて生成されたスレッドを，以降 Ruby スレッドという．図 6.1 では，新しい Ruby スレッドを `Thread.new` によって生成する．`do ... end` で囲まれたブロック

```
m = Mutex.new
th = Thread.new do
  # (A) ここに記述した処理を新しいスレッドとして実行
  m.synchronize do
    # (Am) ここに記述した処理は (Bm) と排他に実行される
  end
end
# (B) ここに記述した処理は (A) と並行に実行される
m.synchronize do
  # (Bm) ここに記述した処理は (Am) と排他に実行される
end
# ...
th.join # ここでスレッド th と合流する
```

図 6.1　Ruby におけるスレッド生成と合流・排他制御の例

の中身 (A) とその後ろに書かれた処理 (B) は別スレッドとして並行に実行される．Ruby スレッドを Ruby プログラム中ではスレッドオブジェクトとして扱うことができ，図 6.1 では生成したスレッドオブジェクトを変数 th に代入している．スレッドオブジェクトに対して Thread#join メソッドを呼ぶことで，スレッドオブジェクトに対応する Ruby スレッドが合流するのを待つ．(Am)，(Bm) の処理は，プログラム冒頭で生成した Mutex オブジェクトにより排他制御される．

Ruby スレッドを利用した実用的な例として，Ruby スレッドを利用した ECHO サーバの例を図 6.2 に示す．親スレッド（プログラム起動時にスタートするスレッド）は TCPServer オブジェクトを利用して，クライアントからの接続を待つ．クライアントからの接続があれば，新しいスレッドを生成し，コネクションを新しいスレッドへ渡す．親スレッドは，コネクションからの接続を待つ．生成されたスレッドは，クライアントからの送信を待機し，受信したテキストをクライアントへ送り返す．親スレッドと子スレッドは並行に実行するので子スレッドが処理を行っている間も親スレッドは新たなクライアントからの要求を処理することができる．

言語処理系にマルチスレッド機能を実現する方法として，ネイティブスレッド処理機構

```
require "socket"

gs = TCPServer.open(0)
addr = gs.addr
addr.shift
printf("server is on %s\n", addr.join(":"))

while true
  Thread.start(gs.accept) do |s|
    print(s, " is accepted\n")
    while s.gets
      s.write($_)
    end
    print(s, " is gone\n")
    s.close
  end
end
```

図 6.2 Ruby スレッドを利用したエコーサーバプログラム (Ruby リファレンスマニュアルより引用)

を利用する方式がある．ネイティブスレッド処理機構は OS など，システムが提供するスレッド管理機能であり，POSIX Thread（以降 Pthread）[25] などの仕様として提供されている．ネイティブスレッド処理機構が対応していれば，スレッドの論理的な並行実行だけではなく，並列計算機上での物理的な同時並列実行を行うことが可能である．なお，この機構により管理されるスレッドをネイティブスレッドという．

現在広く利用されている，Ruby 実装である CRuby では，Ruby スレッドをユーザレベルスレッド[*1]として実現している [63, 75]．これは，複数の Ruby スレッドを一つのネイティブスレッド上で交互に実行させることで，論理的に並行実行するスレッドを実現する方式である．この方式は，スレッド生成などのスレッド制御が軽量に行えることや，ス

[*1] ユーザレベルスレッドをグリーンスレッドとも呼ぶこともある．

レッドスケジューリングを柔軟に行うことが可能であること，移植性の高さなどの利点がある．

しかし，ユーザレベルスレッドでは，常に一つのネイティブスレッドしか利用しないため，Ruby スレッドを並列計算機上で同時並列に実行することができない．近年のマルチコアプロセッサに代表されるメモリ共有型並列計算機のコモディティ化にともない，ソフトウェアの並列実行による高速化は今後ますます重要となる課題であるが，Ruby スレッドが並列に実行できないのは大きな問題である．

また，旧 Ruby 処理系によるユーザレベルスレッド実装では真の並行実行を実現出来ない．たとえば，ある Ruby スレッドが UNIX のシステムコールである select のような，実行をブロックしてしまう処理を実行すると，その他の Ruby スレッドへスケジューリングされずに全 Ruby スレッドがブロックしてしまう．旧 Ruby 処理系では実行をブロックする処理を行わず，処理系内部でポーリングを行いこの問題を回避しているが，いつもポーリングなどの代替手段を用いることが出来るとは限らない．

そこで，YARV に，ネイティブスレッド処理機構を利用してメモリ共有型並列計算機上で並列実行が可能な Ruby スレッドを実現した．

並列実行する Ruby スレッドを実現するにはいくつか課題があるが，特に過去のプログラム資産を利用可能にすることは重要である．Ruby の特長の一つに，世界中で開発されている多くの Ruby 用拡張ライブラリが利用可能という点がある．拡張ライブラリは C 言語などで実装されるネイティブメソッドによって構成されているが，旧 Ruby 処理系ではネイティブメソッド実行中に Ruby スレッド切り替えを起こさないことを保障していた．つまり，既存のネイティブメソッドは逐次実行を前提としているため，並列実行を行うために必要な排他制御などを含まず，スレッドセーフではない．開発する処理系でこれらのプログラム資産を利用できないのは大きな問題である．

そこで，VM 中の全 Ruby スレッドが共有する単一のロック（ジャイアントロック, 以降 GL）を用いて，複数の Ruby スレッドが同時にネイティブメソッドを実行することのないように排他制御を行なう．GL を用いることで旧 Ruby 処理系のために開発されてきたネイティブメソッドをそのまま利用することができる．また，並列度を向上させるために，スレッドセーフに書き換えたネイティブメソッドを GL なしで実行可能にした．スレッドセーフ化を進めることで，段階的に並列度を向上することができる．

YARV の並列化に関する本研究の主眼は，言語処理系の並列化に関する新規手法の提案ではなく，既存の処理系の資源を活用しながら並列実行を行うための機構の提案である．従来の並列処理をサポートする言語処理系の研究開発とは違い，逐次実行を前提として開発されたネイティブメソッドなど，過去のプログラム資産を利用可能としたまま並列

実行による性能向上を実現するという点に着目し，実際に開発を行った．

本章では，まずネイティブスレッドを利用した Ruby スレッド処理機構について述べ，次章で具体的な並列化手法について述べる．

以下，2 節で旧 Ruby 処理系における Ruby スレッド処理機構問題点を述べた後，適切な Ruby スレッドの実現手法および並列化手法について検討する．3 節でネイティブスレッドに対応する Ruby スレッド処理機構の実現手法を述べ，4 節でまとめる．並列実行を行うために必要な機構および Ruby スレッド・並列実行処理機構の評価については次章で述べる．

なお，本研究ではメモリ共有型並列計算機以外は対象としないため，並列計算機は以降メモリ共有型並列計算機を指すこととする．メモリ分散型並列計算機上で仮想的にメモリ共有型並列計算機を実現しているシステムもあるが，それはメモリ共有型並列計算機として扱う．

6.2 Ruby スレッドの並列実行手法の検討

本節では Ruby スレッド処理機構について，旧 Ruby 処理系での実現方式を述べ，その問題点を述べる．そして，その問題点を解決するネイティブスレッドを利用した並列実行可能な Ruby スレッド処理機構の実現方式を検討する．

6.2.1 旧 Ruby 処理系用スレッド処理機構と問題点

旧 Ruby 処理系での Ruby スレッド処理機構の実装方式とその問題点について述べる．

旧 Ruby 処理系でのスレッド実現手法

旧 Ruby 処理系での Ruby スレッド処理機構は，すべてユーザレベルで実現されており，OS やシステムソフトウェアが提供するネイティブスレッド処理機構は一切利用しない．これは主に移植性を高めるためであり，たとえばネイティブスレッド処理機構を用意していないシステム上でも Ruby を利用すればマルチスレッド並行プログラミングが可能になるという利点がある．

Ruby スレッドの生成は処理系の Ruby スレッドスケジューラへ新しいスケジューリング対象の登録という形で行われる．これはメモリアクセスのみで行われるため，生成などの操作に OS へのシステムコールを介すことが多いネイティブスレッドよりも高速である．

Ruby スレッドの切り換えはアラームシグナル（SIGVTALRM）などを利用してプリエンプティブに行われる．一定間隔で起動するアラームシグナルハンドラはインタプリタに共通の割り込みフラグをセットする．処理系は定期的に割り込みフラグをポーリングし，Ruby スレッド切り換えのタイミングを検知することができる．ネイティブメソッドの実行中にはポーリングを行わないので切り換えは起こらない．

スレッドコンテキストの切り換えは次のような退避・復帰処理により行われている．まず，setjmp によって取得する実行コンテキスト，およびマシンスタックをヒープに複製して退避する．復帰時はヒープ上に退避していたマシンスタックをスタック領域へ書き戻し，longjmp によって実行コンテキストを回復する．マシンスタックのコピーを行わなくてもマシンスタックの切り換えは可能であるが [69, 68]，そのためにはシステムや CPU に依存したスレッド生成，コンテキスト切り換え処理が必要となり，移植性が低下する．

Ruby スレッドスケジューラは，ある Ruby スレッドの入出力待ちによるブロックによって全スレッドがブロックすることを防ぐため，select システムコールによってポーリングする [76]．具体的には，スレッドスケジューラが起動すると存在するすべての Ruby スレッドをチェックし，入出力待ちであるスレッドと，入出力待ちをしているファイル記述子を集め，その集合に対して timeout を 0 秒として select システムコールによるポーリングを行う．もし，あるファイル記述子に対してブロックせずに入出力が可能であることがわかれば，そのファイル記述子を待っていた Ruby スレッドを実行可能にする．

また，一時停止中の Ruby スレッドについても Ruby スレッドスケジューラが実行を開始するかどうかを管理している．

解決すべき問題点

旧 Ruby 処理系による Ruby スレッド処理機構が解決するべき問題点は，性能上の問題点と機能的な問題点に分けられる．

性能上の問題点としては，すべての Ruby スレッドが一つのネイティブスレッド上で切り換えられながら実行しているため，並列計算機上での並列実行による性能向上を行うことができないという点が挙げられる．また，コンテキスト切り換えのたびにマシンスタックのコピーを行うのは性能上問題である．そして，スレッドスケジューリングを行うたびにすべての Ruby スレッドをチェックするのは，Ruby スレッド数が増えれば増えるほど実行時間が長くなり問題である．

機能的な問題点は，Ruby スレッドの真の並行実行に対応できないという点である．これは，ネイティブメソッド実行中にブロックしてしまうような処理や計算時間のかかる処理をある Ruby スレッドが実行した場合，Ruby スレッドスケジューラを起動できなくな

るため，全 Ruby スレッドがブロックしてしまうからである．入出力待ちによるブロックは Ruby スレッドスケジューラにおいてポーリングすることで全 Ruby スレッドのブロックを防いでいる．しかし，その他の要因によるブロックが発生した場合は正しく並行実行することが出来ない．

まとめると，解決すべき問題点は次のようになる．

1. 並列計算機による性能向上が困難
2. 高いオーバヘッドのコンテキスト切り換え
3. スケジューラのスケーラビリティの低さ
4. ブロックするようなネイティブメソッドの記述が不可能

6.2.2 ネイティブスレッドを利用した Ruby スレッド処理機構の実現手法の検討

前節で述べた Ruby スレッド処理機構の問題点を解決するため，OS などシステムが提供するネイティブスレッド処理機構を利用する手法を検討する．

ネイティブスレッドの利用に関する議論

前節で述べた問題点を解決するために，特に並列計算機の利点を生かすためにはネイティブスレッドを積極的に利用するのが自然である．並列計算機環境で提供されるネイティブスレッドはそれぞれ並列に実行可能であることが多いからである．

しかし，ネイティブスレッド処理機構を利用することを前提とした場合，それをサポートしていないシステムでの Ruby スレッドの利用が出来なくなるという移植性の問題が考えられる．これに関しては，最近のほとんどのシステムではネイティブスレッド処理機構を提供しているため，大きな問題はないと判断した．たとえば，UNIX 系 OS での Pthread，Microsoft の Windows での独自のスレッドシステムがある．また，現在広く利用されているネイティブスレッド処理系は事実上，上記の二つなので，移植性の点で大きな問題ではないと判断した．

ネイティブスレッドを採用する欠点は，他にもユーザレベルでの独自スレッド制御に比べて，(a) スレッド制御コストが増大する可能性，および (b) 生成可能なスレッド数が一般的に少ない，という点がある．

(a) に関してはネイティブスレッドの生成や終了などの制御にシステムコールを発行する場合が多いため，一般的に制御コストは高くなる．しかし，ネイティブスレッドを利用

すれば環境に依存した高速な Ruby スレッド切り換えを実現できる．Ruby スレッドの生成や終了は，プログラムの最初と最後にまとめて行うようにすることができるが，スレッド切り換えは常に発生する．そのため，ユーザレベルスレッドと比べ，スレッド切り換えの高速なネイティブスレッドのほうが性能的に有利であると判断した．

(b) の生成可能スレッド数は，ユーザレベルで独自にスレッドを実装する場合，生成可能 Ruby スレッド数の制限はメモリサイズのみとなる．しかし，ネイティブスレッドの生成可能スレッド数にはシステム固有の制限があるため，より少数のスレッドしか生成できない．実際に実験したところ，数百から数千スレッドの生成に制限された．しかし，ネットワークプログラムのような Ruby スレッドを利用するアプリケーションにおいて，数千を超える Ruby スレッドを利用することは稀であるため，問題はないと判断した．

Ruby スレッドとネイティブスレッドのマッピング

Ruby スレッドとネイティブスレッドをどのように写像するか，という点では，Ruby スレッド一つにつきネイティブスレッドを一つ用意するのが直観的であり，簡単である (1 対 1 モデル)．コンテキスト切り換えやスレッドスケジューラは，ネイティブスレッド処理機構が提供する効率の良いものを利用することができる．また，あるネイティブスレッドがブロックしても，他のネイティブスレッドは実行を続けることができる．並列実行に対応しているネイティブスレッド処理機構であれば，Ruby スレッドの並列実行による性能向上を得ることができる．つまり，1 対 1 モデルは 2.1.2 で述べた問題点を全て解決する．

このモデル以外にも，M 個の Ruby スレッドにつき，N 個のネイティブスレッドを用意して（$M > N$），M 個の Ruby スレッドを適切なネイティブスレッドに Ruby 処理系側でスケジューリングする（M 対 N モデル）という方式が考えられる．この方式では，スレッド制御をユーザレベルで軽量に行うことができ，並列計算機による性能向上も可能である．しかし，旧 Ruby 処理系と同様に Ruby スレッドをネイティブスレッドに割り当てる処理を独自に実装しなければならず，移植性の問題が発生する．結局，このモデルでは 2.1.2 で述べた問題のうち (2)〜(4) が解決できない．

以上の検討から，1 対 1 モデルを採用することとする．

ただし，Ruby スレッドとネイティブスレッドの対応が 1 対 1 モデルでも，ネイティブスレッド処理機構が Scheduler Activations[2] などの技術を用いた M 対 N モデルのスレッドライブラリであった場合，ユーザレベルでスレッド制御を行うことになり性能が改善する．つまり，ネイティブスレッド処理機構の性能を向上することが Ruby スレッド処理機構の性能向上に直結する．ネイティブスレッド処理機構の性能改善は，システムソフ

トウェアレベル [59, 72]，およびプロセッサレベル [56, 34, 46] でも継続して行われているため，今後の技術向上を期待するのは十分妥当である．

同期粒度の検討

ネイティブスレッドを並行・並列に実行するためには，共有資源に対する一貫性保障のための同期や排他制御が必要となる．並列度をあげ，並列計算機上による性能向上を図るためには，これらが必要な箇所を特定し，出来る限り短い同期区間とするべきである．

しかし，すでに存在する膨大なネイティブメソッドの実装は逐次実行を前提としているため，並列実行のために必要な排他制御を行っていない．排他制御を行わないまま Ruby スレッドを並列実行すると，すでに他の Ruby スレッドによって解放したメモリ領域へアクセスしてメモリ保護例外を発生させるなどの，Ruby プロセスを異常終了させるような処理を行う可能性がある．本論文ではこのような異常終了を引き起こす可能性がある処理を危険な処理とし，そうでない処理を安全な処理と定義する．

排他制御は，細粒度ロック（以降 FL）かジャイアントロック（GL）を利用する方法が考えられる．FL は排他制御を必要とする共有資源ごとに用意するロックである．GL は全 Ruby スレッドが共有する単一のロックであり，複数の共有資源をまとめて排他制御する．GL ではなく FL を利用するほうが排他制御の対象が限定的であるために並列度は向上するが，対象ごとにロックを用意する必要があり，実装は困難になる．

ネイティブスレッドを利用するにあたって考えられる，処理系による同期粒度の選択肢は (a) 処理系が FL により排他制御，(b) 処理系が GL により排他制御，(c) 処理系が GL により常に並列実行を抑止，(d) 処理系側では排他制御などを行わずにユーザプログラマが Ruby プログラム上で適切な排他制御を行うことを必須とする，などが考えられる．これらの方法をまとめたものを図 6.3 に示す．また，参考までに，旧 Ruby 処理系での Ruby スレッド処理機構を (e) として示す．

図 6.3 の水平方向の矢印は右方向を正とする時間軸を表し，点線は Ruby スレッドがブロックしていることを表す．RT は Ruby スレッド，NT はネイティブスレッドを示す．NT1, 2 はそれぞれ CPU1, 2 上で実行しているとする．以下，各方式について説明する．

(a) は各共有資源に対して細粒度ロックを用いて処理系による排他制御を行う方式である．図 6.3(a) では，共有資源 Resource1 と Resource2 に対して RT1，RT2 がそれぞれ並列にアクセスしているが，それぞれ別の細粒度ロック FL1，FL2 を用いて排他制御を行っているため，RT1 と RT2 は並列に実行していることを示している．この方式は性能上有利であるが，既存のネイティブメソッドのソースコードを精査し，適切な細粒度ロックの挿入が必要である．この作業は膨大であり，作業も一般的に困難であるため，既存の

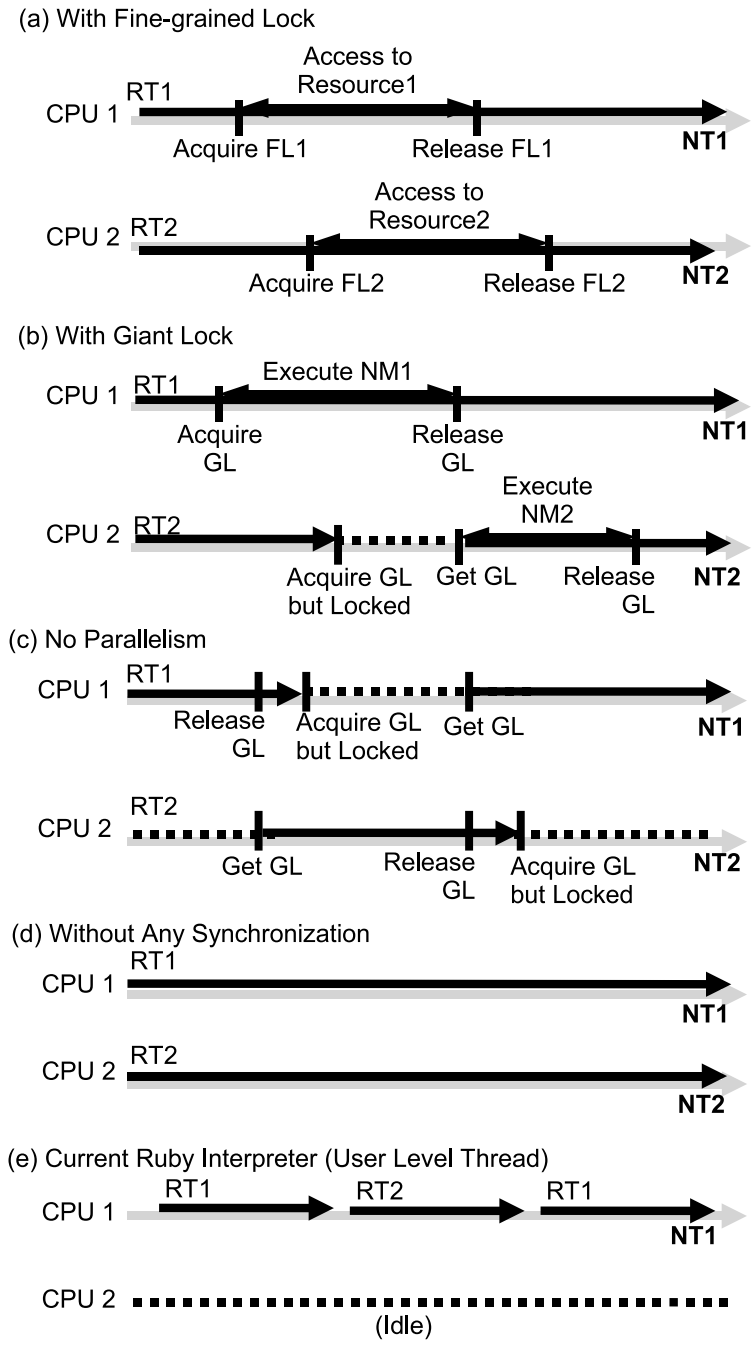

図 6.3　同期粒度の方式検討

ソフトウェアの再利用という点から，この方式を採用するのは現実的ではない．

(b) は，危険な処理を行う可能性のあるネイティブメソッドを，処理系で GL を獲得してから実行する方式である．図 6.3(b) は，RT1 がネイティブメソッド NM1 を実行中に RT2 がネイティブメソッド NM2 を実行しようとするが，RT1 が GL を獲得しているため，RT2 が RT1 の GL 解放まで待機してから MN2 を実行することを示している．この方式は，(a) よりも処理系の実装が容易であり，ネイティブメソッド以外は並列実行可能であるという利点があるが，GL による並列度の抑制，GL 制御のオーバヘッドが問題である．

(c) は GL を実行権として扱い，GL を獲得している Ruby スレッドのみしか実行しないよう，処理系が制限する方式である[*2]．図 6.3(c) は，GL を保持する Ruby スレッドのみが実行していることを示している．この方式では同時に単一の Ruby スレッドしか実行されないため，排他制御などが不要であり，旧 Ruby 処理系用に開発された多くのプログラム資源が利用可能であるという利点がある．しかし，並列計算機上で並列実行できないという従来の Ruby スレッド処理機構と同じ問題を有する．

(d) は排他制御などを処理系側では行わず，必要な排他制御は Ruby プログラム記述者が行うとする方式である．この方式は実装が容易であり，性能の点でも最も有利である．なぜなら，(a) ではプログラム上排他制御が不要な共有資源でも必ず排他制御を行うが，(d) では排他制御を行うかどうかは Ruby プログラム作成者が選択することができるためである．しかし，作成者が排他制御についてすべて責任を負うため，誤って危険な処理を行う可能性がある[*3]．

プログラミング言語 Ruby，および処理系の設計思想として，高速な実行よりも，ユーザに対してなるべく容易な手法を提供するという方針がある．ここでいうユーザとは，Ruby プログラム作成者はもとより，ネイティブメソッド記述者も含む．方式 (d) は前者，方式 (a) は後者に対して大きな負担を強いることになる．

そこで，方式 (b) を取り，スレッドセーフであると明示しないネイティブメソッドの実行は GL により排他制御することにした．これより，ネイティブメソッドの開発において，並列実行のための特別な処理は記述しなくても済む．つまり，すでに開発されてきた多くのネイティブメソッドがそのまま利用できる．そして，性能に影響する処理を，細粒

[*2] この方式を採用しているスクリプト言語 Python[42] の処理系では，ここでいう実行権をジャイアントインタプリタロックと言うが，(b) における GL とは意味が違う．

[*3] (a)〜(c) についても，トランザクション処理などにおいては Ruby プログラム記述者による排他制御の挿入が必要である．しかし，これを怠ってもプログラムの結果が意図したものにならないだけで，処理系が異常終了するような危険な処理を行うことは無い．

度ロックを用いた排他制御の追加などを行いスレッドセーフな実装へ書き換えるていくことで，段階的に方式 (a) へ近づけ，並列度を向上させていくことを可能にする．とくに，Ruby のようなオープンソースソフトウェアとして開発が進められているプロジェクトでは，細粒度ロックを利用した方式に書き換えるのではなく，本方式のような段階的な変更を許容する方式が適している．

問題となる GL 獲得のためのオーバヘッドはロック獲得の機会を減少することによって回避する．

検討のまとめ

最後に，本節で述べた検討の結果をまとめる．

開発する Ruby スレッド処理機構はネイティブスレッド処理機構を用いて Ruby スレッドを並列実行可能にする．Ruby スレッド一つにネイティブスレッド一つを割り付ける 1 対 1 モデルとする．スレッドセーフではないネイティブメソッドを実行する際には GL を用いて排他制御し，既存のネイティブメソッドの実装を用いた Ruby プログラムを安全に実行できるようにする．ただし，スレッドセーフに書き換えたネイティブメソッドは並列に実行させる．これにより段階的な並列化による性能向上を可能にする．

次節以降，この検討を踏まえたネイティブスレッドを用いた Ruby スレッド処理機構について述べる．また，GL を用いた並列実行の実現，および評価については次章で述べる．

6.3　ネイティブスレッドを利用した Ruby スレッドの実現

本節では，Ruby スレッドにネイティブスレッドを割り当てる方法について，Ruby スレッド特有の機能を実現する方法とあわせて具体的に述べる．

6.3.1　Ruby スレッドシステムの全体像

前節の検討の結果，YARV では Ruby スレッドとネイティブスレッドを 1 対 1 対応させるモデルを採用することにした．図 6.4 に YARV の Ruby スレッドシステム，ネイティブスレッド処理機構，並列計算機を含めた全体像を示す．

図 6.4 の RT は Ruby スレッド，NT はネイティブスレッドに対応する．PE は Processor Element の略で，計算機環境の並列実行単位を表し，PE の数だけ並列度があることを示す．Ruby スレッドに対応するネイティブスレッドはネイティブスレッドスケジューラによって，並列実行単位に割り付けられ，並列実行する．

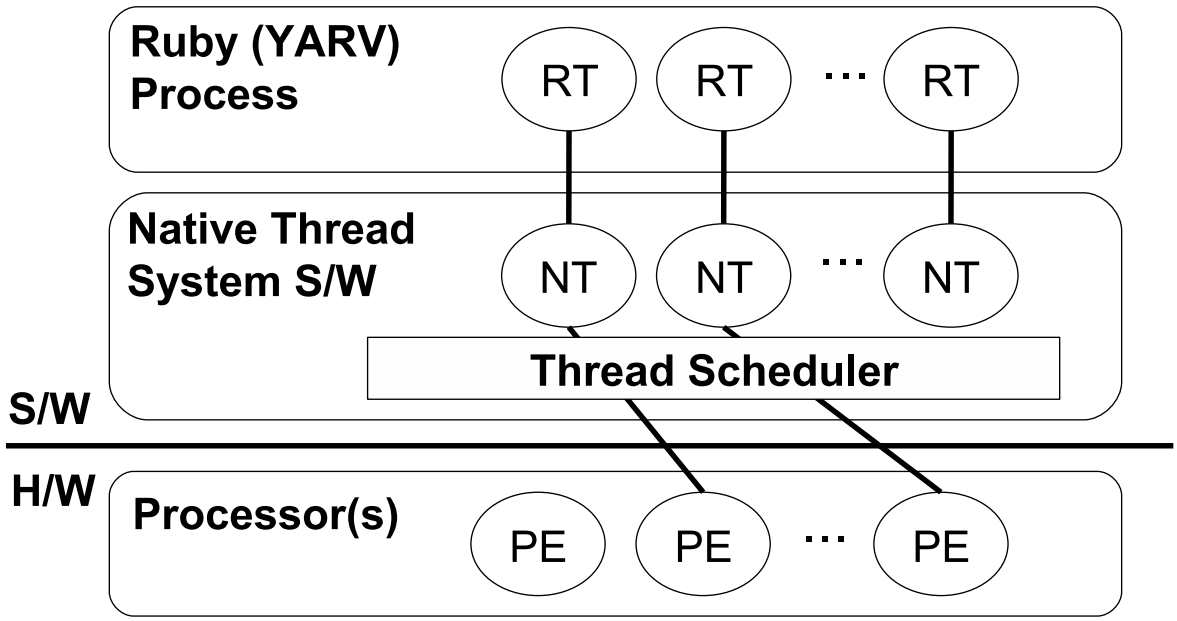

図 6.4 Ruby スレッドシステムの全体像

6.3.2 Ruby スレッド管理データ

YARV では Ruby スレッドを管理するために，スレッド管理データを各 Ruby スレッドごとに保持している．スレッド管理データについて，簡略化したものを図 6.5 に C 言語風の擬似コードで示す．

`thread_id` は対応するネイティブスレッドを示す識別子である．Pthread 環境では，型 `thread_id_t` は `pthread_t` の意味である．`status` は Ruby スレッドの実行状況であり，実行中やブロック中，終了後などの情報を保持する．`wait_next_id` は次節で述べるスレッドの合流にて利用される．`lock`，`unblock_func` は後述するブロック解除関数，`sleep_cond` は同様に後述するスレッドの一時停止に利用される．

その他にスレッド管理データは VM 実行コンテキストを保持している．これは，その Ruby スレッドを実行するために必要なスタックやプログラムカウンタ，スタックポインタなどを意味する．仮想マシンはこの VM 実行コンテキストに随時アクセスしながら Ruby プログラムを実行する．

```
struct thread_data {
  thread_id_t thread_id;
  int status;
  struct thread_data *wait_next_id;
  lock_t lock; void (*unblock_func)();
  cond_t sleep_cond;
  // その他，VM 実行コンテキスト
  // （スタック，仮想レジスタなど）
};
```

図 6.5　スレッド管理データ

6.3.3　Ruby スレッドの制御手法

図 6.1 で述べた生成，合流，排他制御を実現する Ruby スレッド制御について，ネイティブスレッド処理機構を用いる方法を述べる．

Ruby スレッド制御は仮想マシンの命令ではなく，対応するネイティブメソッドにより実現している．以下，Ruby スレッド制御について，対応するネイティブメソッドとその実現手法を述べる．

生成 (Thread.new)　Ruby スレッドの生成時に，対応するネイティブスレッドの生成を行う．Pthread では `pthread_create`，Windows スレッドでは`_beginthreadex` API を利用した．生成されたネイティブスレッドは Ruby スレッドとして実行を開始する．

終了 (Thread#exit)　Ruby スレッドの実行が終了すると，後処理を終わらせてネイティブスレッド自体も終了するように実装した．ネイティブスレッドが終了・消滅しても，Ruby レベルでアクセスできるスレッドオブジェクト自体はオブジェクトへの参照がある限り生き続ける．

合流 (Thread#join)　スレッドの合流はネイティブスレッド処理機構が提供する手段（たとえば，Pthread の `pthread_join`）は用いず，スレッド管理データに待ちスレッドとして登録（`wait_next_id` を利用）するようにして独自に実装した．これは，

システムの資源であるネイティブスレッドを，Ruby スレッドの実行終了後，可能な限り早く解放するためである．すでにネイティブスレッドが終了している場合，すぐに合流処理は成功する．

排他制御 (Mutex#synchronize) 排他制御は，ネイティブスレッドが提供する排他制御機能を利用して実装した．具体的には，Pthread 環境では `pthread_mutex_lock / unlock`，Windows スレッドでは `Enter / LeaveCriticalSection` を利用した．Windows スレッドにおいて，Mutex オブジェクトを利用しなかったのは，CriticalSection のほうが軽量であること，そして Ruby のスレッド制御では Mutex オブジェクトが提供するプロセス間での同期機能などが必要なかったためである．

6.3.4 Ruby スレッドのスケジューリング

Ruby スレッドの切り換え，スケジューリング処理はネイティブスレッド処理機構の該当機能をそのまま利用する．コンテキスト切り換えは，旧 Ruby 処理系が行っていた非効率な方式ではなく，ネイティブスレッド処理機構によるシステムに依存した高速なコンテキスト切り換えを行うことが期待できる．

Ruby スレッドのスケジューリングの公平性は，ネイティブスレッド処理機構の行うスレッドスケジューリングに依存する．Ruby スレッドに設定された優先度は，ネイティブスレッド処理機構が提供する優先度設定に適切にマッピングする．

6.3.5 Ruby スレッドへの割り込み

Ruby スレッドの特長として，任意のタイミングで特定の Ruby スレッドに対して割り込みをかけ，特定の処理を行わせるという機能がある．たとえば，任意の Ruby スレッドに対して，実行中の処理に割り込み強制的に例外を引き起こすことができる（図 6.6）[*4]．

この割り込み機能は，ポーリングポイントを VM 内に適切に設けることで実現する．割り込みを行う際には，対象の Ruby スレッドに対して割り込み要因を記述してから割り込みフラグをセットする．対象 Ruby スレッドはポーリングタイミングで割り込み処理が必要であることを検知することができる．このポーリング間隔が割り込みへの反応速度を決定するが，YARV では VM のジャンプ命令やメソッド起動・終了時にポーリングを行うようにしている．Ruby プログラムでは特にメソッドの起動が頻繁に行われるので，反

[*4] Ruby の標準添付ライブラリの一つである `timeout` は強制的に例外を引き起こす機能を利用して実装されている．

```
th = Thread.new do
  ... # (A)
end
# (A) の処理を実行中に例外を発生させる
th.raise SomeException
```

図 6.6　他の Ruby スレッドに対し例外を発生するプログラム

応時間は十分短い．

ここで問題になるのが，割り込み対象の Ruby スレッドがなんらかの理由により一定時間以上ブロックしていた場合である（以降，ブロック状態という）．ブロック状態では割り込みフラグに対するポーリングは行わないため，割り込みフラグのみでは即座に割り込み処理を起こすことが出来ない．

このような問題は，たとえば，入出力待ちのためにブロック状態となった Ruby スレッドの処理を，他の Ruby スレッドからタイムアウトのために中断したいという場合に発生する．ネットワークプログラミングにおいて，タイムアウトのために例外を発生させ，入出力待ちを中断させるというのは，Ruby ではよく行われるイディオムである．

この問題に対処するために，ブロック解除関数を利用することにした．ブロック解除関数とは，ブロックする要因に応じて，そのブロック状態をキャンセルするための処理を行う関数である．ある Ruby スレッド RT がブロック状態になる可能性のある処理を実行する前に，ブロック解除関数のポインタを RT のスレッド管理データ（3.2 節で述べた `unblock_func` エントリ）に設定する．RT はブロック状態から抜けるときにブロック解除関数の登録を削除する．ブロック状態の RT へ割り込みを行うには，RT に登録されたブロック解除関数を実行する．ブロック状態を脱した RT は割り込みフラグをチェックすることで割り込みを検知する．ブロック解除関数の設定と実行は排他的に行う必要があるが，並行処理の性質上，正当性の検証が容易ではないため，現在も検討中である．

例として，select システムコールを実行してブロック状態となった Ruby スレッド RT をどのように中断させるか説明する．Pthread 環境において select システムコールの実行を中断させるには，`pthread_kill` 関数で中断させたいネイティブスレッドに対してシグナルを送るという方式がある．そこで，ネイティブスレッドへシグナルを送るというブロック解除関数 `UBF_select` を作成する．RT が select システムコールを実行する前に，

```
// select システムコールのためのブロック解除関数
UBF_select(thread_data *th){
  while (th はブロック状態？) {
    pthread_kill(th->thread_id, SIGVTALRM);
  }
}
// select システムコールを実行するネイティブメソッド
ruby_select(thread_data *th){
  set_unblock_function(th, UBF_select);
  // select システムコールを発行・ブロック状態へ
  select(...);
  // select システムコール・ブロック状態から抜ける
  clear_unblock_function(th);
  CHECK_INTERRUPT(th);
}
// 割り込みをかける処理
interrupt(struct thread_data *th, intr_t *intr){
  SET_INTERRUPT(th, intr); // 割り込み要因の設定
  if (th->unblock_function)
    (th->unblock_function)(th);
}
```

図 6.7 ブロック解除関数とその利用例

RT に UBF_select を登録する．割り込みを行う場合，RT に登録された UBF_select を，RT を引数として実行し，select システムコールを中断させる．これらの処理を C 言語風の擬似コードにまとめたものを図 6.7 に示す．なお，図では排他制御に関する処理は省略してある．

本節冒頭で述べた，他 Ruby スレッドへの例外発生は，YARV によって次のように行われる．まず例外を発生させるという情報を対象 Ruby スレッドへ登録し，対象 Ruby スレッドのブロック解除関数が登録されていればそれを実行する．対象 Ruby スレッドは

例外が配送されたことを検知して例外処理を発生する．

6.3.6 シグナルの扱い

シグナルは UNIX でのプロセス間通信などに用いられるが，Ruby レベルでもシグナルはサポートしている．つまり，Ruby プロセスが受信したシグナルに応じて実行するべき処理を Ruby プログラムで記述することができる．シグナルに関してはシステムによって扱いが異なるため，ここでは Pthread 環境についてのみ説明する．

Pthread 環境ではネイティブスレッドごとにシグナルマスクを持つ．プロセスに送られたシグナルは，そのシグナルがマスクされていないネイティブスレッドのうちどれか一つに配送される．任意の Ruby スレッドがシグナルを受信可能とすると，任意のタイミングでシグナルを受信することを考慮しなければならず，実装が複雑になる．また，Pthread の条件変数による一時停止（`pthread_cond_wait`）などは，シグナルハンドラの実行によって解除できないため，シグナルに対応する処理を正しく実行することが出来ない可能性がある．

そこで，Ruby スレッドに割り当てるネイティブスレッド以外に，管理スレッドというネイティブスレッドを用意した．管理スレッド以外のネイティブスレッドのシグナルマスクを設定し，管理スレッドにのみシグナルが配送されるようにした．管理スレッドがシグナルを受信すると，即座にメインスレッド（Ruby プログラム開始と同時に生成される Ruby スレッド）へ前節で述べた割り込みをかけ，受信したシグナルに対応する処理を実行させる．

6.3.7 Ruby スレッドの一時停止

YARV では `sleep` メソッドなどで Ruby スレッドを一時停止させるために，Pthread 環境下では条件変数と `pthread_cond_timedwait`，もしくは `pthread_cond_wait` 関数を利用して一時停止を実現した．前者は時間制限のある一時停止（いわゆる sleep），後者は他 Ruby スレッドからの割り込みなどがあるまで中断する場合に利用する．ブロック解除関数は `pthread_cond_signal` 関数を呼び出す．図 6.8 に Pthread 環境での一時停止処理を C 言語風擬似コードで示す．

Windows の場合は，`WaitForMultipleObjects` でイベントオブジェクトを待ち合わせることで実現した．ブロック解除関数は `SetEvent` 関数でイベントオブジェクトをシグナル状態にすることで実現した．

```
// 一時停止を解除するブロック解除関数
UBF_sleep(thread_data *th){
  pthread_cond_signal(th->sleep_cond, ...);
}
// 一時停止するネイティブメソッド
ruby_sleep(thread_data *th, timeval *t){
  set_unblock_function(th, UBF_sleep);
  if (t == 0) // 割り込みがあるまでブロック
    pthread_cond_wait(th->sleep_cond, ...);
  else        // 指定時間までブロック
    pthread_cond_timedwait(th->sleep_cond, ...);
  clear_unblock_function(th);
  CHECK_INTERRUPT_FLAG(th);
}
```

図 6.8 Pthread 環境での一時停止処理

Thread#join での Ruby スレッドの合流にもこの一時停止の機構を利用した.

6.4 まとめ

本節では，プログラミング言語 Ruby の並列実行を可能にするために，Ruby スレッドを並列に実行するための基盤である Ruby スレッドのネイティブスレッド対応について述べた.

マルチコアに代表されるメモリ共有型並列計算機がコモディティ化している現在，プログラミング言語で並列性を活かす工夫は必須である．とくに，いわゆるスクリプト言語が目指している，手軽にプログラミング，という文脈で並列計算を利用することは，今後の計算機環境の促進のためにとくに重要である.

プログラミング言語 Ruby では，言語レベルでスレッドによる並行性のサポート，つまり論理的な命令列の多重化をサポートしているので，このスレッドを複数同時に実行するように並列化を行うのは，モデル的にも自然である.

並列化にあたっては，旧 Ruby 処理系でスレッドを実現するために利用していた独自のユーザレベル実装について述べ，本手法がスレッドの実現手法として，効率的に問題であり，また並列実行が不可能であることを述べた．そして，これを解決するために OS など，システムが提供するネイティブスレッドを用いた Ruby スレッド処理機構について検討した．ネイティブスレッドを用いる利点は，並列実行が可能であること，ほぼ仕様が POSIX Thread，もしくは Windows スレッドに 2 極化されており，移植性が保てることを確認し採用した．

ネイティブスレッドと Ruby スレッドをどのようにマッピングするかについては，検討の結果 1:1 型として実装した．1:1 対応にすることで，実装が単純になるという大きな利点がある．Ruby スレッドを並列実行するにあたり，必要になる同期は，スレッドセーフでないネイティブメソッドを VM ごとに一つ用意するジャイアントロックを用いて排他制御することで，既存の多くのスレッドセーフでない Ruby 用拡張ライブラリを利用可能とする方針とした．ただし，仮想マシン内部で利用するデータ構造については細粒度排他制御を行うことで，処理系自体は並列実行可能とすることにした．

Ruby スレッドをネイティブスレッドに割り当てるために，Ruby スレッドの機能，具体的には生成，同期，スケジューリングをネイティブスレッドの機能をそのまま利用することで実現した．ただし，合流に関してはネイティブスレッドの機能をそのまま利用することは出来なかったため，ネイティブスレッドの機能を利用して実装した．

また，Ruby スレッドの割り込み機能も同様にネイティブスレッドに該当する機能が無かったため，そのための機構を用意した．Ruby スレッドの割り込み機能とは，ある Ruby スレッド A から，他の Ruby スレッド B に強制的に例外を発生させる機能であり，タイムアウト処理等に利用されている．この機能では，Ruby レベルでは割り込みフラグをポーリングルすることで実現できるが，C レベルでブロック状態となった場合，例えば `select` システムコールでネイティブスレッドがブロックしてしまった状態で割り込みがかけられた場合，この `select` システムコールを解除する手法が必要となる．ブロックする処理ごとにブロックを解除方法は異なるため，ブロック状態になりえる処理を行う前には，ブロック状態を解除する処理を行うブロック解除関数を登録し，ブロックとなりえる処理を行う，という枠組みを用意することでこの問題を解決した．OS のシグナル対応や，スレッドの一時停止などの機能も，Ruby スレッドに対する割り込みと言えるため，この枠組みを利用して実現した．

本節では，並列処理を行うに必要な，Ruby スレッドのネイティブスレッド対応について述べた．このネイティブスレッド対応は，並列計算機以外の環境での，並列実行を許さない処理系でも必要となる検討課題であった．なぜなら，旧 Ruby 処理系が行っていた独

自のユーザレベルスレッド実装では，ネイティブスレッドを利用する外部ライブラリとの併用が困難であるという問題があったためである．YARV では，本章で述べた手法を用いて Ruby スレッドをネイティブスレッド対応とすることで，この問題を解決した．

次章では，実際に並列処理を行うにあたって必要な作業，とくに同期手法とその軽量化について述べる．また，実際にメモリ共有型並列計算機上で実行し，ネイティブスレッドを用いた Ruby スレッドによって Ruby プログラムを並列実行した結果について述べる．

第 7 章

Ruby スレッドの並列化

7.1 はじめに

本章では，Ruby プログラムを並列に実行するために必要な Ruby スレッド処理機構に必要な仕組み，とくに並列実行に必要となる排他制御の Ruby 処理系への導入手法について述べ，実際に Ruby スレッドを並列実行し評価した結果を述べる．

並列・並行実行に必要な，ネイティブスレッドを利用した Ruby スレッド処理機構の構成を前章で述べた．しかし，このまま Ruby スレッドを並列に実行させると，Ruby スレッド間で共有するデータの一貫性が取れなくなり，またスレッドセーフではない C 言語で記述した Ruby 用拡張ライブラリを実行することによってメモリ保護違反などの危険な処理を行ってしまう．

そこで，Ruby 処理系に適切な排他制御を導入し，並列実行によって引き起こされる危険な処理を回避するようにした．具体的には Ruby スレッド間で共有するデータ構造について検討し，メモリ管理やガーベージコレクション（GC）について再検討した．さらに，仮想マシンごとにひとつ用意するジャイアントロック（GL）を用いてスレッドセーフでない処理を適切に同期し，保護するという手法により，Ruby スレッドを並列に実行させることにした．C によって実装された機能を徐々にスレッドセーフにすることで，段階的な並列化が可能になる．また，これらの排他制御の導入に伴う性能低下を避けるため，可能ならば排他制御不要なデータ構造にするなどの手法を適用した．

そして，実際に並列計算機上で実行した評価結果を述べる．評価は，いくつか特徴の違う Ruby スレッドを利用するプログラムをメモリ共有型並列計算機上で実行した結果を述べる．評価の結果，GC や GL の獲得・解放が必要ない，フィボナッチ数を求めるプログラムでは台数効果が確認できた．しかし，GC や GL が発生するプログラムでは性能向上

の限界や性能低下を確認した.

7.2 並列にアクセス可能なスレッド管理データ

Ruby スレッドを並列実行するためには, VM コンテキストを参照するために, 各 Ruby スレッド管理データへ同時並列にアクセスする必要がある.

これを実現するために, ネイティブスレッド処理機構がサポートするスレッドローカルストレージ（以降, TLS）を利用した. TLS はネイティブスレッドごとにそれぞれ別々にもつ空間である. 各 Ruby スレッドはそれぞれのスレッド管理データへのポインタを, 対応するネイティブスレッドの TLS に格納する.

TLS の利用方法はネイティブスレッド処理機構によって様々であるが, たとえば Pthread の規格のみに沿うのであれば `pthread_getspecific` 関数や, これに関する API が利用できる. また, OS と C コンパイラの組み合わせによっては通常のグローバル変数定義に `thread`, もしくは`__thread` 指示子を付加することで, その変数をスレッドローカル変数として利用できる.

7.3 並列実行に対応したメモリ管理

Ruby はガーベージコレクション（以降, GC）によるメモリ管理を行うが, 並列実行を行うために排他制御や同期を行わなければならない. そこで, 本節では YARV が行う並列実行に対応したメモリ管理について述べる. なお, 今回は既存のメモリ管理を拡張する形で実装を行ったため, 旧 Ruby 処理系でのメモリ管理とあわせて説明する.

7.3.1 旧 Ruby 処理系のメモリ管理

ここで, 説明のために旧 Ruby 処理系のメモリ管理を簡単に紹介する. Ruby でのメモリ管理はオブジェクト単位で行うので, オブジェクトの管理方式について説明する.

図 7.1 に示すとおり, オブジェクトスペースには生きているオブジェクト（図中白丸）と解放されたオブジェクト（図中灰色の丸）がある. 解放されたオブジェクトは FreeList という連結リストによって管理されている. 生きているオブジェクト, 解放されたオブジェクト, および FreeList はどの Ruby スレッドからも参照可能である.

オブジェクト割り当て時には FreeList の先頭のオブジェクトを新しいオブジェクトとして割り当てる（図中 (a)）. この処理は連結リストの操作として行われる.

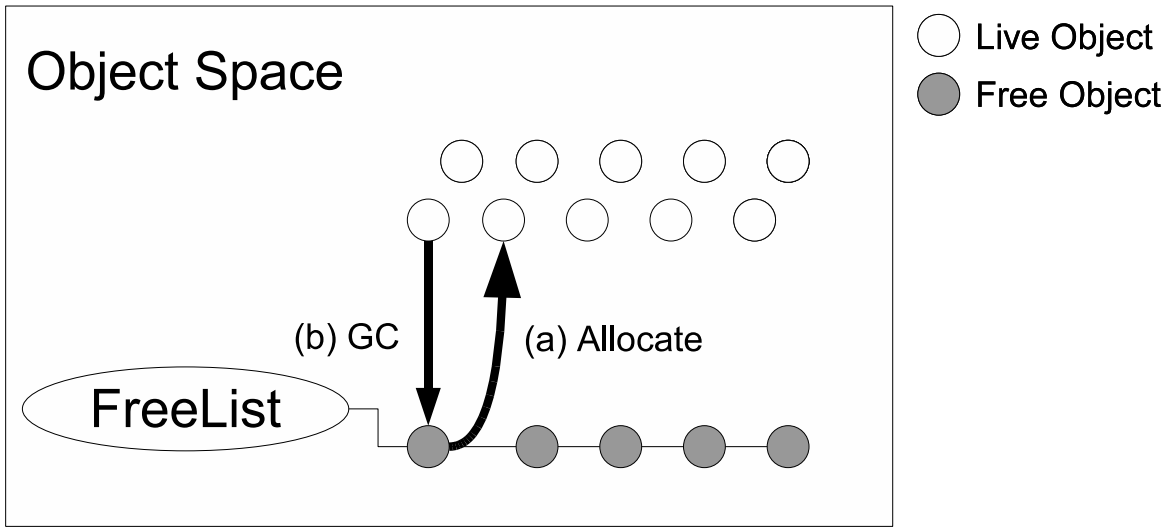

図 7.1 旧 Ruby 処理系でのメモリ管理

GC アルゴリズムは停止型の保守的マークアンドスイープである [75]．これは，ネイティブメソッド実装時にリードバリアやライトバリアが必要なく，実装の単純さを優先するための選択である．

GC はマシンスタックなどを根としてマークする．マークされなかったオブジェクトが回収され FreeList につながれる（図中 (b)）．

旧 Ruby 処理系ではメモリ管理の実行中に他の Ruby スレッドへ切り換わることが無いように実装されているため排他制御は一切不要である．

7.3.2 オブジェクトアロケーション

Ruby スレッドが並列実行する場合，FreeList はどの Ruby スレッドからも同時にアクセスされる可能性があるため，FreeList に対するリスト操作には排他制御が必要である．しかし，Ruby プログラムにおいて頻発するオブジェクトアロケーションのたびに排他制御を行うのは性能上問題である．そこで，スレッドローカルな FreeList（以降，TLFL）を用いることでこの問題を解決した．この方式を図 7.2 に示す．

まず，Ruby スレッドは FreeList から，解放されたオブジェクトを N 個（現在の実装では $N = 4096$）確保し，TLFL へ連結する（図 7.2 の (a)）．このとき，FreeList のリスト操作は排他制御を伴う．

オブジェクトの割り当て時には，TLFL の先頭から一つオブジェクトを取り出して割り当て処理を行う．TLFL は他の Ruby スレッドからはアクセスされないため排他制御は

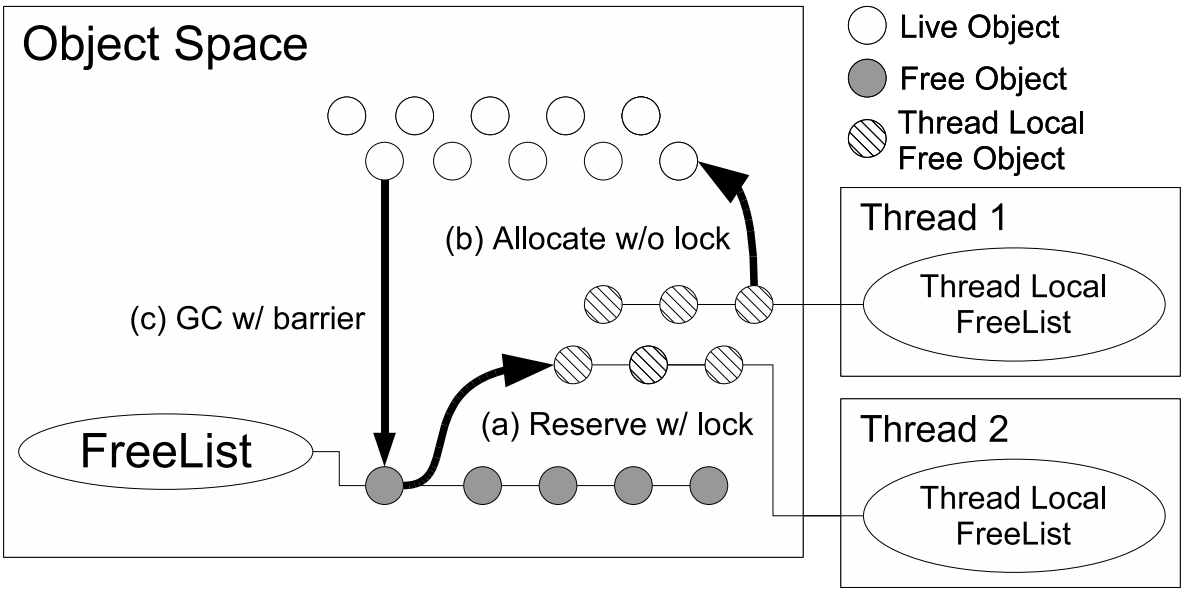

図 7.2 並列実行に対応したメモリ管理

不要である．もし TLFL が空だった場合，再度 FreeList から一定数確保する．

この工夫により，オブジェクト確保の際に排他制御が必要になる機会を $1/N$ 回に削減している．

7.3.3 ガーベージコレクション

YARV では旧 Ruby 処理系と同様の理由で停止型の保守的マークアンドスイープ GC を採用している．

停止型 GC を実現するため，GC 時には全 Ruby スレッドを停止する．Ruby スレッド数 4 で，GC 実行 Ruby スレッド RT1 が GC を行う様子を図 7.3 に示す．RT1 は他の Ruby スレッド全てが一時停止するまで，つまり (全実行スレッド数 $rt\#$ − 待機スレッド数 $wt\#$) が 1 になるまで待機する．RT2〜RT4 は同期のために wt#を 1 ずつ増加させてから一時停止する．RT4 が一時停止する際，$rt\# - wt\# = 1$ となるため RT1 の実行を再開させる．RT1 は GC を実行（図 7.2(c)）し，GC 終了後に GC のために一時停止していた全 Ruby スレッドの実行を再開する．

Ruby スレッドは，前節で述べた割り込みフラグポーリングポイントにおいて GC 処理が発生中であるかをチェックし，同期を行う．同期は Pthread 環境での条件変数，Windows 環境でのイベントハンドルを利用して実現した．

ブロック状態の Ruby スレッドは GC のための待ちを行うことが出来ないので，ブロッ

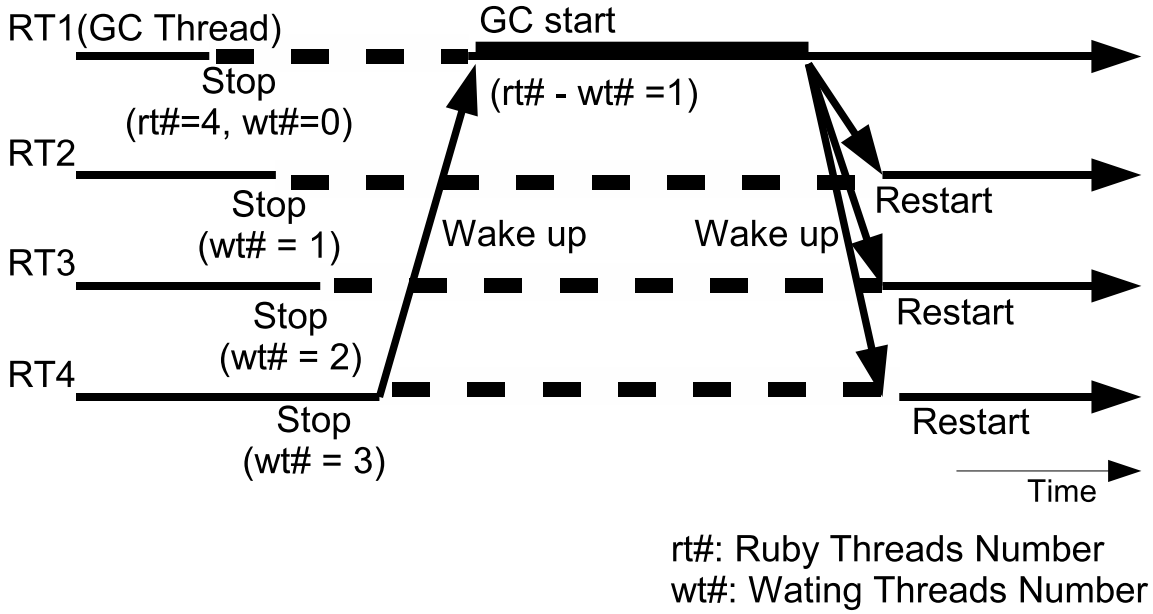

図 7.3 GC のための同期

ク状態になる前に待機スレッド数を増加させておく．そのとき，マークに必要となるマシンスタックなどの情報も同時に保存しておく．GC 実行中にブロックが解除されたときには GC が終了するまで待機する．GC のための同期中に Ruby スレッドが終了した場合は，全実行スレッド数を 1 減らして GC スレッドに通知する．

7.4 排他制御の導入

本節では並列実行にあたって必要となった排他制御の導入について，特にハッシュ表の排他制御，およびジャイアントロックの導入について述べる．

7.4.1 ハッシュ表の排他制御

旧 Ruby 処理系，および YARV が利用しているハッシュ表の実装では，参照と設定処理はアトミックに行わなければならない．そこで，ハッシュ表ごとに細粒度ロックを設けて，ハッシュ表に対する参照・設定時には細粒度ロックを用いて排他制御を行うようにした．

YARV において，グローバル変数，インスタンス変数，定数，メソッドテーブルなどの並列に利用される可能性がある機能はハッシュ表を用いて実現している．ハッシュ表への

アクセスをスレッドセーフにしたことにより，これらの機能もスレッドセーフとなった．

7.4.2　ジャイアントロックの導入

既存のネイティブメソッドは逐次実行を前提としておりスレッドセーフではないので，複数のネイティブメソッドを同時並列に実行すると Ruby プロセスの異常終了を引き起こすような危険な処理を行う可能性がある．そこで，6.2.2 で検討したとおり，そのような恐れのあるネイティブメソッドは，VM につき一つ用意するジャイアントロック（GL）を獲得してから実行するようにした．

GL 獲得のタイミングはスレッドセーフと明示的に示されていないネイティブメソッドの起動時に行う．なお，明示する方法については次項で述べる．獲得した GL は，ネイティブメソッド終了時に解放する．この方式により，すでに多く存在する Ruby 用拡張ライブラリを安全に利用することができる．

ただし，ネイティブメソッド実行中に Ruby プログラムで定義されたメソッドを呼び出すことができるので，その際は GL を解除してメソッドを実行する．メソッドが処理が戻った時，GL を再獲得する．

また，YARV では例外処理のために `longjmp` 関数によって大域ジャンプを行う場合があることを [45] で論じた．このとき，GL を獲得していない状態で大域ジャンプしたが，ジャンプ先を記録した時点（`setjmp` した時点）では GL を獲得していた場合，GL 獲得状態に不整合が生じる．そのため，`setjmp` 関数によって実行コンテキストを保存する際には同時に GL 獲得状態も保存しておき，大域ジャンプによって GL 獲得状態の不整合が起こることを防ぐ．

7.4.3　スレッドセーフな処理を宣言するための API

ネイティブメソッドをスレッドセーフであると明示するための Ruby C API として `rb_define_method_ts` 関数（ts: Thread Safe）を用意した．この API で定義されたネイティブメソッドは，呼び出し時に GL 獲得操作を行わない．

スレッドセーフなネイティブメソッドを段階的に増やしていくことで，YARV の並列度を順次向上することができる．

7.5 排他制御オーバヘッドを削減するための工夫

排他制御を行うと，ロック獲得，解除のためのオーバヘッド，および競合状態におけるネイティブスレッドの待ち状態への遷移にかかるコストが問題になる．また，現在のYARV の実装ではスレッドセーフなネイティブメソッドが少ないため，GL 獲得操作は比較的多く行われる．したがって，排他制御のオーバヘッド削減の工夫は重要である．

そこで，本節では排他制御オーバヘッドを削減するための工夫について述べる．

7.5.1 ロック不要なメソッドキャッシュの参照

4 章で述べたとおり，YARV ではグローバルメソッドキャッシュ [75]，およびインラインメソッドキャッシュを利用している [73]．キャッシュ表の参照はキャッシュエントリとしてメソッド ID やクラスをキーとし，メソッド本体を値としている．このエントリの更新，および参照はアトミックに行わなければ不整合を生じる．図 7.4 の (A) にロックを用いたメソッド検索処理を C 言語風の擬似コードで示す．このプログラムではメソッド探索を行う間，メソッドキャッシュ全体をロックする．しかし，Ruby プログラムではメソッド起動は頻繁に行われるため，メソッドキャッシュ参照のたびに排他制御を行うのは性能上問題である．

そこで，キャッシュミス時にはメソッドキャッシュエントリ自体をアトミックに切り替えることで排他制御を不要とした．この処理を図 7.4 の (B) に示す．キャッシュミス時にエントリの内容を書き換える際，キャッシュエントリを GC 対象オブジェクトとして新しく生成し，キャッシュテーブルに登録する．キャッシュエントリの参照とキャッシュ表への登録はポインタアクセスであるためアトミックに行うことができる．したがって参照時に排他制御を行う必要はない．

なお，図 7.4(B) の処理順序は意味があるので，コンパイラによる最適化等で入れ替わらないように工夫する必要がある他，プロセッサによってはメモリオーダリングに応じた適切なメモリバリア命令の挿入などが必要となるので注意されたい．

キャッシュミス時は新たに GC 対象オブジェクトの割り当てが起こりコストがかかるが，ヒット時は排他制御が必要ない．メソッドキャッシュのヒット率は十分高い [73] ため，採用した手法のほうが有利である．

ただし，プログラムによってはインラインメソッドキャッシュのヒット率が低くなる場合があるため，ミスを繰り返したら今後そのメソッド呼び出しではインラインメソッド

```
/* (A) 参照時に排他制御を用いる手法 */
cache_entry method_cache[CACHE_SIZE];

method_search_with_cache(class, id){
  LOCK(method_cache_lock); // ロック獲得
  cache_entry *e = &method_cache[HASH(class, id)];
  if (!(e->class == class && e->id == id)) {
    /* キャッシュミスのため，メソッド探索を行い
       メソッドキャッシュエントリを更新 */
    e->class = class; e->id = id;
    e->body = method_search(class, id);
  }
  UNLOCK(method_cache_lock); // ロック解放
  return e->body;
}

/* (B) 参照時に排他制御が不要な手法 */
cache_entry *method_cache[CACHE_SIZE];

method_search_with_cache(class, id){
  cache_entry *e = method_cache[HASH(class, id)];
  if (!(e->class == class && e->id == id)) {
    /* キャッシュミスのため，メソッド探索を行い
       新しいメソッドキャッシュエントリを作成 */
    method_cache[HASH(class, id)] = e = new_entry(
      class, id, method_search(class, id));
  }
  return e->body;
}
```

図 7.4 メソッドキャッシュ実現手法

キャッシュを利用しないようにした．

7.5.2 1 スレッド実行時のジャイアントロック

Ruby スレッドが一つしかない場合，GL による排他制御を行う必要はないので行わない．

Ruby スレッドが一つであるかどうかの判断は，共有資源である同時実行スレッド数カウンタを参照する必要がある．しかし，そのカウンタが 1 であれば，ほかの Ruby スレッドがその値を変更する可能性はないため，排他制御無しにこの確認を行うことができる．厳密には，タイミングによって Ruby スレッド数を 2 以上と誤判定する可能性があるが，通常の GL 獲得処理を行うだけなので問題ない．

また，単一 Ruby スレッド実行時に新たな Ruby スレッド生成時，GL を獲得してから生成処理を行うことで，GL 獲得状態の一貫性を保つことができる．

7.5.3 スピンロックの利用

GL 獲得時，他の Ruby スレッドによって GL がすでに獲得されていた場合，競合が発生する．事前評価によると，Pthread 環境にて，ロックの競合によりネイティブスレッドを待ち状態へ移行する場合，大きなオーバヘッドがかかることがわかった．そこでスピンロックを積極的に利用するようにした．本項では Pthread 環境での実装例を紹介する．

まず，`pthread_mutex_trylock` 関数を用いて一定回数（現在の実装では 100 回）スピンロックすることにした．各繰り返しでは `sched_yield` 関数によってプロセッサの実行権を手放し，他の Ruby スレッドに制御を移すことで GL を獲得している Ruby スレッドの実行を先に行うようにした．

また，`pthread_mutex_trylock` の実行回数も削減するために GL が解放されているかどうかのチェックも行った．これをまとめ，Pthread 環境での GL 獲得処理を C 言語風の擬似コードで記述したものを図 7.5 に示す．なお，GL 解放時には `gl_locked` 変数を 0 クリアする．

7.5.4 利用 CPU の制限

GL 獲得のための競合が多発し，前述したネイティブスレッドの待ち状態への遷移が頻発すると大きく性能が低下する．これは，複数の Ruby スレッドが頻繁に GL 獲得を行うことで生じる．

```
pthread_mutex_t giant_lock;
int gl_locked;

acquire_giantlock(){
  for (int i=0; i<100; i++){
    if (gl_locked == 0 &&
        pthread_mutex_trylock(&giant_lock) == 0) {
        gl_locked = 1; return;
    } sched_yield();
  } // 競合発生
  pthread_mutex_lock(&giant_lock);
  gl_locked = 1;
}
```

図 7.5 ジャイアントロック獲得処理（Pthread 環境）

このような状況では，競合を起こしている Ruby スレッドがその後も競合を発生すると予測できる．そこで，競合を頻発させる Ruby スレッド群は並列実行しないようにした．具体的には，競合を繰り返す Ruby スレッド群が利用できる CPU を一つに制限することで実現した．現在の実装では，制限する Ruby スレッド群を 1 秒間に GL 競合回数が 3000 回を越える Ruby スレッドの集合としている．この数値は環境や動作させるプログラムによって最適な値が異なるため，YARV 起動時に指定出来るようにした．この機能は `pthread_setaffinity_np()` 関数（NPTL[8]），または `SetThreadAffinityMask()` 関数（Windows）を用いて実装した．ただし，これらの関数をサポートしていないシステムではこの機能は無効となる．

なお，利用プロセッサを制限した後，プログラムの挙動が変化し，競合が起こらないようになっている可能性があるので，この制限は一定の間隔（現在の実装では 1 秒ごと）で解除するようにした．

7.5.5 単一実行権による逐次実行

並列計算機ではない計算機システムや，本質的に並列計算できないプログラムの場合，並列実行による性能向上が得られないため，並列実行は排他制御のオーバヘッドの分だけ無駄である．

そのため，GL を単一実行権として扱い，Ruby スレッド切り換え時に GL を解放，つまり実行権を手放し，他の Ruby スレッドへ遷移するというモード（コンパイルオプション）を設けた．これは，6.2.2 において検討した方式 (c) を実装したものである．

このモードでは並列実行を行うことができないが，実行時に処理系による排他制御をほぼ行わない．並列実行による性能向上が見込めない場合，このモードが性能的に有利である．

7.6 評価

本節では，ネイティブスレッドに対応した Ruby スレッドの性能と，並列性の評価について述べる．

評価環境は，Intel Xeon CPU E5335 2.00GHz のプロセッサ（Quad Core CPU）を 2 個利用した，合計 8 コア構成の共有メモリ型並列計算機上で行った．利用したソフトウェアは次の通りである．OS は GNU/Linux 2.6.18 x86_64，コンパイラは gcc version 4.1.2 20061115 を利用した．比較対象とする Ruby 処理系は ruby 1.8.6 (2007-11-02 patchlevel 5000) [x86_64-linux] を用いた．Linux 環境下であるため，利用するネイティブスレッド処理機構は Pthread 環境（NPTL）である．YARV の最適化オプションは 4 章で述べた最適化オプションで融合操作，スタックキャッシング最適化以外を適用した．

評価は複数回実行し，もっとも速いものを計測結果とした．

7.6.1 スレッド制御プリミティブの評価

本節では，ネイティブスレッドを用いた Ruby スレッド処理機構の，それぞれの機能の性能評価を行う．

スレッド生成，合流，同期

Ruby スレッド制御プリミティブの実行時間を表 7.1 に示す．

Ruby スレッドの生成はネイティブスレッドの生成コスト（この場合，`pthread_create`

表 7.1 スレッド制御プリミティブの性能評価

	Ruby	YARV	NT Primitive
生成（10 万回生成に要した秒数）	0.89	1.95	0.59
合流（10 万回合流に要した秒数）	0.04	0.99	0.52
排他制御（100 万回行った秒数）	0.67	0.38	-

のコスト）がそのままかかるので，性能が悪い．また，ネイティブスレッド生成時間以外に，仮想マシンのための VM スタックの割り当て時間も生成オーバヘッドも大きいことがわかった．合流についても，ネイティブスレッドによる同期処理（この場合 `pthread_cond_wait`）が必要になるため，旧 Ruby 処理系に比べて性能は低いことを確認した．

排他制御は `Mutex` オブジェクトを利用したロックの獲得と解除を繰り返し行ったものである．YARV のほうが若干性能が高い．これは YARV での Mutex に一部機能が不足しているためである．具体的には，Mutex をロックした Ruby スレッドが，Mutex をロックしようとしてブロックした Ruby スレッドを，他の Ruby スレッドからの割り込みによりロック獲得処理をキャンセルするための処理である．

一般的に，ある程度長期間生きる Ruby スレッドを利用するプログラムでは，スレッド生成コストよりも排他制御のコストのほうが問題になる．そのため，スレッド生成および合流の遅さは大きな問題にならないと言える．

なお，旧 Ruby 処理系においては，同期，排他制御のプリミティブとして `Thread.critical` の設定がある．これは，セットすると他の Ruby スレッドへのスイッチを禁止するという機能である．しかし，この機能にはいくつか問題があり，並列実行に対応するにはコストが大きすぎるため廃止することが決定している．そのため，本論文では `Thread.critical` による排他制御は評価に含めなかった．

スレッド切り換え

Ruby スレッドを二つ用意し，それぞれスレッド切り換え処理を 10 万回行い，その速度を計測した．旧 Ruby 処理系のスレッド切り換えにはマシンスタックのコピーを伴うため，スタックの深さに処理時間が比例するという問題があった．そこで，図 7.6 で示すように，スレッド切り換え時におけるスタックの深さをパラメータとしてスレッド切り換え

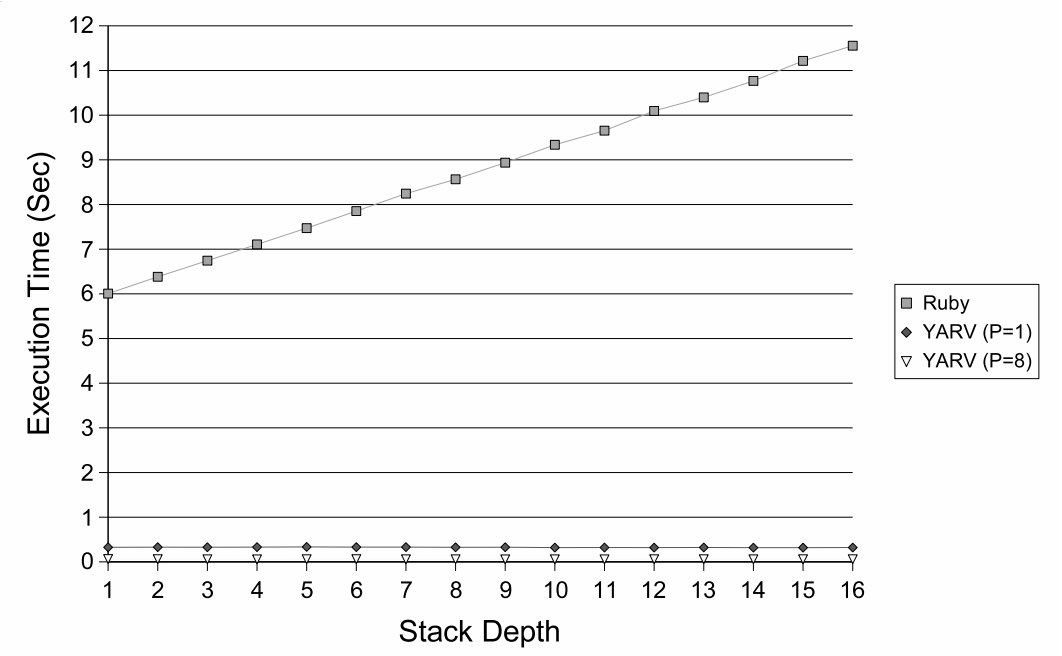

図 7.6 コンテキストスイッチの実行時間

の処理時間を計測した．評価には旧 Ruby 処理系 (Ruby) と，利用するプロセッサコア数が 1 の場合である YARV (P=1) と 8 の場合である YARV (P=8) を用いた．

評価の結果，旧 Ruby 処理系はスレッド切り換えのコストがスタックの深さに比例するが，YARV のコストは一定であり，旧 Ruby 処理系よりも軽量であることがわかった．また，YARV (P=1) と YARV (P=8) を比較した場合，YARV(P=8) が 2 倍以上高速であり，並列実行の効果が出ていることが確認できた．

7.6.2 マイクロベンチマーク

図 7.7 および図 7.8 にマイクロベンチマークの結果を掲載する．この評価では，比較対象として旧 Ruby 処理系 (Ruby)，CPU プロセッサコア数を 1, 2, 4, 6, 8 個上で実行する YARV (P=1)〜YARV (P=8) と，YARV (NoPara) を用意した．YARV (NoPara) は 7.5.5 で述べた並列実行を行わない代わりに排他制御，ジャイアントロックの獲得，解放のオーバヘッドを不要とする動作モードである．各グラフともに，縦軸を単一 CPU で実行する YARV (P=1) の実行時間を基準とした速度向上比としている．横軸は，図 7.7 ではベンチマークプログラム，図 7.8 ではコア数を示しており，各ベンチマークごとの YARV (P=1)〜YARV (P=8) の値をプロットした折れ線グラフである．

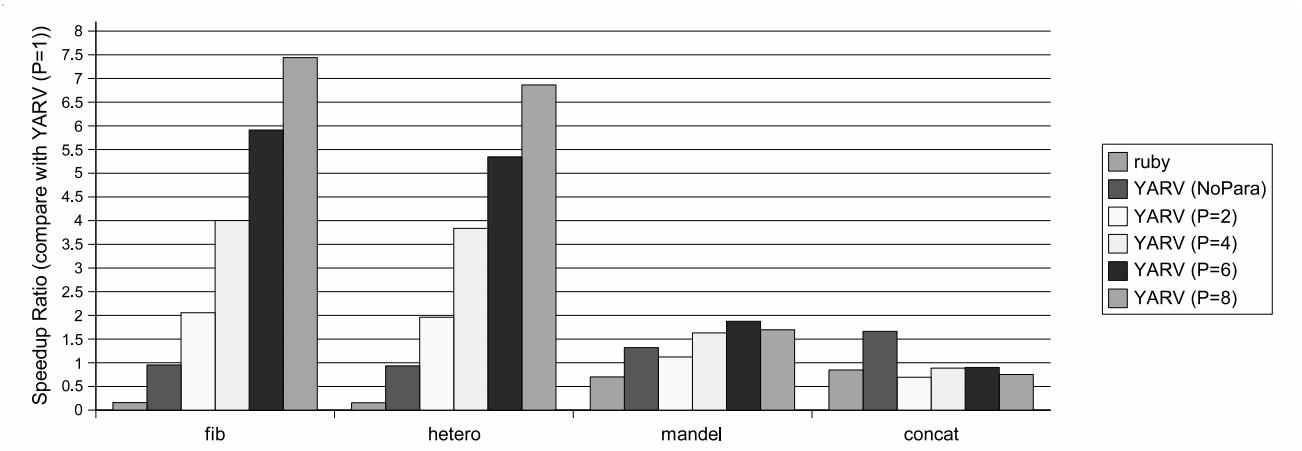

図 7.7 並列プログラムのマイクロベンチマーク結果（プログラムごと）

マイクロベンチマークに利用したプログラムは次の通りである．fib は 40 番目のフィボナッチ数を再帰処理で求めるプログラムである（図 7.9）．concat は 1 つの文字列オブジェクトに対して複数の Ruby スレッドから文字列を追加するプログラム，mandel はマンデルブロ集合を求めるプログラムで，32 個の Ruby スレッドを生成し，処理をそれぞれ実行した．concat は全 Ruby スレッドが同時に一つの文字列に追加するため競合が発生する．hetero は，40 番目のフィボナッチ数を並列に求めるプログラムと，concat プログラムと同様に文字列を追加していくプログラムを 1 スレッド動かしたもので，フィボナッチ数を求めるプログラム，および文字列追加を行うプログラムがそれぞれ並列に実行される．

評価結果を見ると，fib ではほぼ台数効果が確認できた．これは，GL や GC が不要であり，各 Ruby スレッドが互いに独立に実行出来るためである．

しかし，concat のような文字列処理で GL による排他制御が必要な場合[*1]，並列計算による性能向上は見ることが出来なかった．また，ロック獲得のオーバヘッドが不要であるという点から，YARV (NoPara) がもっともよい性能を示した．

mandel は結果を集める処理で GL が必要な処理があったため，またガーベージコレクションのオーバヘッドがプログラム全体の処理時間の多くを占めるため，台数効果を得ることが出来なかった．ガーベージコレクションは合流のオーバヘッドがあるため，Ruby スレッドの数が大きいと性能が悪くなる．そのため，YARV (P=8) では性能が落ちてい

[*1] 文字列処理に関しては，文字列ごとに細粒度ロックを用意し，これを用いて排他制御することで GL の利用を不要とすることが出来る．しかし，現在の実装では文字列処理のすべてをスレッドセーフな処理に置き換えていないため，ここでは GL による排他制御を行っている．

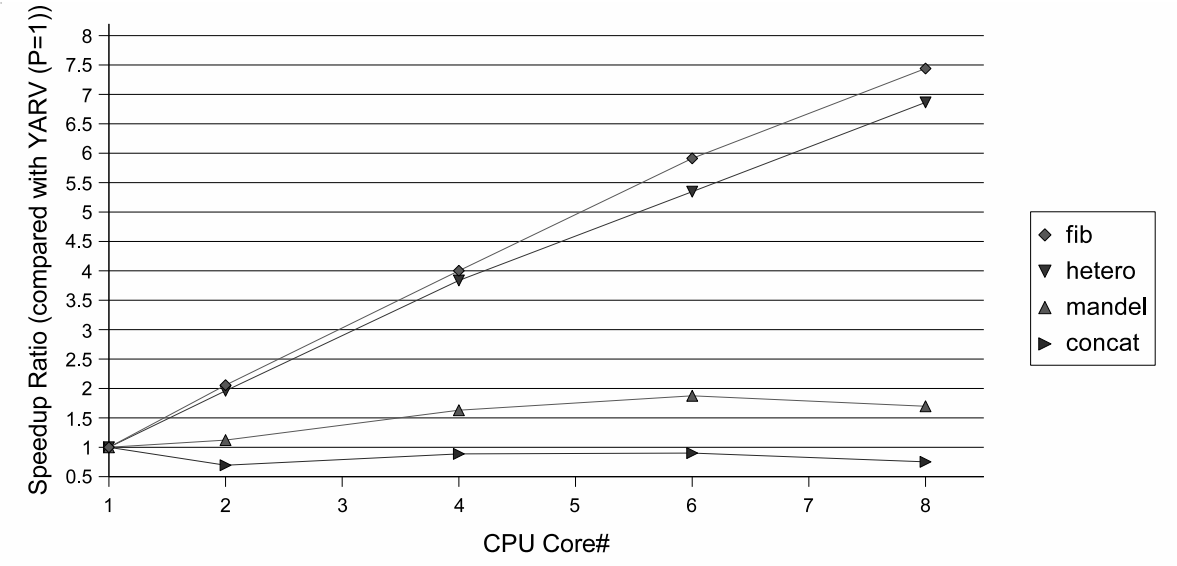

図 7.8 並列プログラムのマイクロベンチマーク結果（コア数での性能推移）

ることがわかる．

hetero の結果から，ある一つの Ruby スレッドが GL を必要とする処理を実行しても，他の Ruby スレッドを並列実行することで，台数効果を確認することができた．

ここで, GL による排他制御が性能にどのような影響を与えるかを確認するため, YARV (P=8) の実行における各プログラムでの GL の獲得頻度，および競合頻度を調査した．この結果を表 7.2 に示す．それぞれ GL 獲得回数，GL 競合回数を YARV (P=8) の実行時間（秒）で割ったものである．

fib は GL 獲得回数が少ないため，GL 制御のオーバヘッドは無いことがわかる．concat は GL 獲得・競合頻度が高いため，7.5.4 で示した利用 CPU 制限機能が働き，ほぼ単一 CPU で実行することとなった．そのため，並列実行による性能向上が得られなかった．hetero では，fib の処理と concat の処理には依存関係がないため，concat 処理での GL 獲得頻度は高くても GL 競合頻度が少なくなり，利用 CPU 制限が起こらなかった．このため，並列実行による性能向上を得ることができた．mandel は GL 獲得回数は比較的少ないが，結果を集める処理の一部に長時間 GL を獲得したまま解放しない場合があったため，競合が多く発生した．これが並列実行における性能向上のボトルネックの一要因となった．

この結果から，GL を必要とする処理が多い現在の実装では，hetero のような，GL を利用する Ruby スレッドのほかに GL 不要の計算を行う Ruby スレッドを作り並列度を

```
# 逐次実行版
def fib n
  if n < 2
    1
  else
    fib(n-1) + fib(n-2)
  end
end

# 並列実行版
def pfib n
  if n < 2
    1
  else
    if n < 32
      fib n
    else
      t1 = Thread.new{pfib(n-1)}
      t2 = Thread.new{pfib(n-2)}
      t1.value + t2.value
    end
  end
end
```

図 7.9　フィボナッチ数を求める Ruby プログラム

向上するのが現実的である．

7.6.3　利用 CPU の制限

7.5.4 で述べた利用 CPU 制限がどの程度効果があったかについて，ジャイアントロックの競合が頻発した concat プログラムで評価を行った．評価は利用 CPU 制限を行う場

表 7.2 GL 獲得・競合頻度

ベンチマーク名	GL 獲得頻度	GL 競合頻度
fib	0.45×10^3	25.03
hetero	$1,083.78 \times 10^3$	36.39
mandel	395.25×10^3	643.23
concat	$2,393.635 \times 10^3$	9.95

(times/sec)

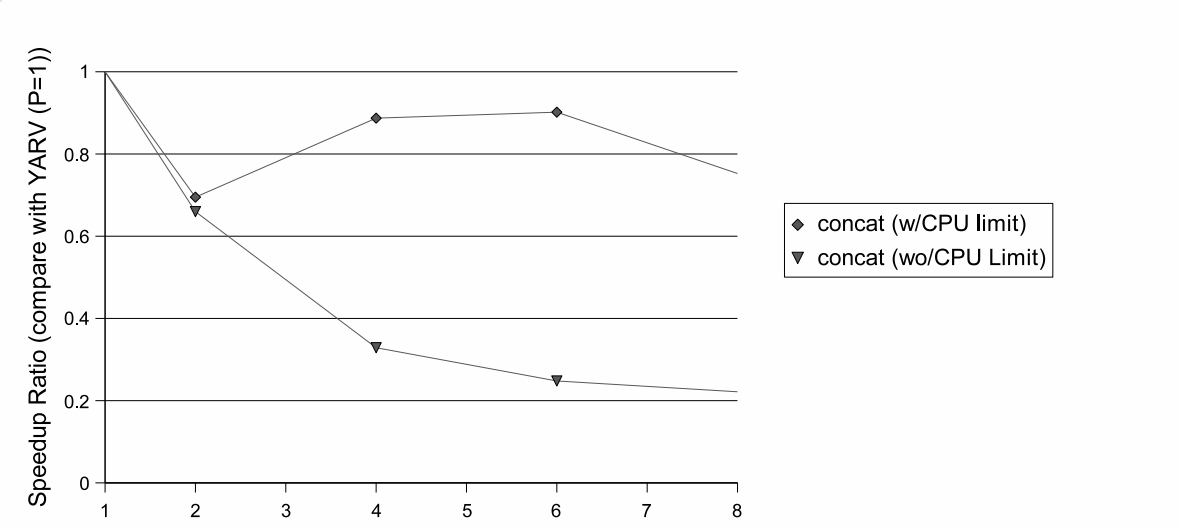

図 7.10 利用 CPU 制限の効果の確認

合と行わない場合で，それぞれ 1〜8 コアで並列実行した．評価結果を図 7.10 に示す．

評価の結果，利用 CPU 制限を行わない場合は，8 CPU コア利用時に 1 CPU コア利用時に比べ速度向上比が 0.22 となり，約 4.5 倍の性能低下となったが，ジャイアントロックの競合を検知し，競合が発生している Ruby スレッドの利用可能 CPU コアを制限することによって，速度向上比が 0.75，約 1.3 倍の性能低下に抑えることができた．これは，対処を行わない場合に比べて 3.4 倍の性能向上を実現したことになる．

7.7 関連研究

すでに多くの並列実行をサポートするプログラミング言語とその処理系が研究，開発されている [77]．しかし，それらの言語の処理系は開発当初より並列化を意識した構造としているため，本論文で議論したような，既存のコードを活用するなどの問題の多くはそもそも存在しない．

プログラミング言語 Java[52] ではマルチスレッド，および並列実行に関する研究が多く行われてきており，特に Java 仮想マシン [66] における同期コストの削減のための研究が行われている [67]．Java で実装する Java 仮想マシンである Jalapeno[1] では，本論文では利用しなかった M 対 N モデルを採用しており，プロセッサの命令を利用して同期，排他制御などを実装し，高い性能を得ている．しかし，その方式では移植性に問題があるため，YARV では採用しなかった．[21, 40] などの研究では Java 仮想マシンでのネイティブスレッドの利用について，移植性や性能についての議論しているが，それらの知見は本論文で述べた実装にもある程度あてはまる．

Ruby と比較されるその他のプログラミング言語のマルチスレッド対応については，Perl[36] では Parrot[54] にマルチスレッド機能を取り込む作業を行っているが，仕様を検討中の段階である．現在の Perl 処理系（Perl 5.8）がもつ ithread[51] は，スレッド間で共有するオブジェクトを明示的に指定せねばならず，Ruby のスレッドと等価なものではない．Python[42] では本論文 7.5.5 で述べた単一実行権を用いて同時に一つの Python スレッドしか動作させないというネイティブスレッドを用いたスレッドの実装を行っているが，並列計算機による性能向上を得ることができない．また，Python のガーベージコレクタはリファレンスカウンタで実装しているため，並列実行可能な実装にするにはオーバヘッドがかかる．Ruby，および YARV では停止型の保守的マークアンドスイープ GC を採用しているのでその問題はない．

7.8 まとめ

本章では，Ruby を並列に実行するにあたり必要になる，Ruby スレッドのための同期手法について検討した．そして，実際にメモリ共有型並列計算機上で評価を行い，Ruby スレッドの並列実行による性能向上について調査した．

メモリ保護違反などを起こさない，Ruby プログラムの安全な実行を実現するためには，言語処理系に適切な同期が必要である．

本章ではまず，言語処理系内部で利用するスレッド管理データをネイティブスレッド処理機構が提供するスレッドローカルストレージによって，各 Ruby スレッドが並列に Ruby スレッドを管理する領域へアクセスできるようにした．そして，全 Ruby スレッドを合流させて実行するガーベージコレクションについて解説した．そして，各 Ruby スレッドがスレッドローカルフリーリストを持つことで，オブジェクト割り当て時の排他制御を不要とした．

仮想マシン内で利用する共有データは，ほぼハッシュ表で管理しているため，共有データの排他制御はハッシュ表へのアクセス時に集中して行うことで解決した．そして，スレッドセーフでないネイティブメソッドを実行する際にはジャイアントロックを獲得し実行するようにすることで，過去のプログラム資産を利用可能とした．ジャイアントロックの管理は，例外発生時の大域ジャンプ発生時に整合性を取る必要があるが，これを解決するために例外処理にジャイアントロック獲得状況を記録する必要があることを述べた．

これらの排他制御は大きなオーバヘッドが伴う．とくに，現在の YARV ではスレッドセーフでないネイティブメソッドが多く存在するため，ジャイアントロックの獲得，解放，競合のオーバヘッドが大きな問題となる．そこで，いくつかの同期，排他制御コストを削減する手法について論じた．

まず，4 章で述べたインラインメソッドキャッシュを排他制御を排他制御不要なデータ構造にする手法について提案した．Ruby スレッドが 1 つしか存在しない場合，ジャイアントロックの獲得，解放は無駄なので省略することを述べた．排他制御にあたっては，完全にブロックするような同期プリミティブの利用をさけ，スピンロックを利用したほうが効率がよいことを述べた．そして，ジャイアントロックの獲得が競合している場合，競合のオーバヘッドが非常に大きくなるため，競合している Ruby スレッドを一つの CPU コア上でしか実行させなくする，利用 CPU 制限手法を提案した．

メモリ共有型並列計算機上での評価の結果，Ruby スレッド生成と合流にはネイティブスレッドを利用するオーバヘッドのために性能低下が起きたが，同期オーバヘッドが小さいことを確認した．生成オーバヘッドでは，ネイティブスレッド生成のオーバヘッドとともに，VM 用スタックの生成オーバヘッドが大きいことがわかった．スレッド切り替えについては，旧 Ruby 処理系で問題となった，スタックの深さに比例したスレッド切り替えオーバヘッドについては，ネイティブスレッドを利用した Ruby スレッドでは存在しないことを確認した．

並列実行を行った結果，8 CPU コアを利用した場合，最大 7.4 倍程度の性能向上を達成し，台数効果を確認することができた．とくに，ジャイアントロックを必要とする処理と，ジャイアントロックを必要としない処理を行う Ruby スレッドを混合したときに台数

効果による性能向上を観測することができた．しかし，多くのオブジェクトを生成するプログラムでは，ガーベージコレクションを行う際の同期のオーバヘッド，および並列化していないガーベージコレクションのため，性能が頭打ちとなった．ジャイアントロック獲得が頻発するプログラムにおいて，ジャイアントロックの競合による大きな性能低下が観測されたが，競合発生時の動的な利用 CPU 制限によって，性能低下を抑え，最悪値に比べて 3.4 倍の性能向上を達成することができた．

Ruby の並列化に関して今後の課題は多いが，特に標準ライブラリにスレッドセーフなネイティブメソッドが少ないので見直しが必要である．中でも Ruby プログラムで頻出する文字列，および配列操作についてスレッドセーフ化を行い処理並列性を向上させることが必要である．

また，現在 GC の処理は逐次実行しているが，評価の結果問題であることがわかったため，並列 GC の導入を検討したい．その他，Ruby スレッドの生成が高速であるという前提で Ruby スレッドを多く生成するような Ruby プログラムが多くあるので，たとえばスレッドプールを利用するような方式に変更していく必要がある．

プログラミング言語，とくに Ruby のような気軽にプログラミングを行うことを目標とするスクリプト言語による並列化の支援は，今後ますます重要になっていくため，この問題にはこれからも積極的に取り組んでいきたい．

第 8 章

結論

8.1 本研究の成果と得られた知見

本研究では，プログラミング言語 Ruby 用の高速な仮想マシン構成法について述べた．実際に開発した仮想マシンには，動的特性を持つ Ruby に適用可能な高速化手法を搭載した．また並列計算機での並列実行を可能にするため，まず Ruby スレッド処理機構を計算機システムが提供するネイティブスレッドを用いて構築し，適切な同期処理を用いて並列実行することを可能とした．実際にいくつかのプログラムで評価を行い，これらの最適化，並列化手法に効果があることを確認した．また，旧 Ruby 処理系との互換性や移植性の維持，保守性の向上を行うための工夫により，実用的な言語処理系を実現した．

ここで開発した新しい Ruby の処理系は，Ruby 処理系の公式の次期バージョンとして採用されることが決まっている．本研究によって言語処理系の高速化に関する種々の知見のみならず，実際に国際的に広く公開・利用される高性能・高品質のソフトウェアが生み出されたことを強調しておきたい．

以下に本研究の成果と得られた知見をまとめる．

8.1.1 仮想マシン YARV の構築

高速な Ruby 用仮想マシンを実現するにあたり，2 章で Ruby の言語仕様，および現状の Ruby 処理系（旧 Ruby 処理系/CRuby）の特徴を述べ，高速化にあたっての問題点を述べ，高速な Ruby 処理系開発の課題をまとめた．とくに，旧 Ruby 処理系の抽象構文木をそのまま辿る構造では，既知の言語処理系の高速化技術が適用できず問題であり，そして Ruby の動的な仕様が静的な解析と最適化を阻害することを述べた．つまり，Ruby 処

理系の仮想マシンモデルへの転換と，静的な解析に寄らない，Ruby に適した実行時最適化技術の適用が課題となる．

この課題を達成するために，Ruby 用仮想マシン YARV: Yet Another RubyVM を設計した．インタプリタとしての利便性を保つために，実行時に Ruby プログラムをコンパイルし YARV 命令列へ変換し，実行する構成とした．YARV の計算モデルは，Ruby のメソッド呼び出しを多用するためスタックへ値を格納することが多いという性質と，コンパイル時間が短いというインタプリタ向けの性質を考慮してスタックマシンモデルを採用した．Ruby プログラムを正しく表現するためのスタックマシンモデルの YARV 命令を設計し，全部で 55 命令となった．

仮想マシンは仮想レジスタ，値スタック，メソッド情報を管理するためのコントロールフレームスタック（CF スタック）を利用して計算をすすめるように設計した．値スタックと CF スタックを別途用意したのは，クロージャ生成における複雑なポインタ操作を単純化するためである．

例外処理は例外表を用いて，例外が発生しない限り余計なオーバヘッドを生じない方式を用いることで大幅な高速化を実現した．しかし，C で記述した Ruby 拡張用のネイティブメソッド中でも例外処理を行うという，Ruby C API に対応するために，setjmp/longjmp を用いる例外処理方式も同時に利用するようにした．つまり，例外表が利用できる部分は例外表を，ネイティブメソッドレベルでは setjmp/longjmp を用いる手法を利用するというハイブリッドな方式とした．

仮想マシンの高速化には，静的解析を必要としない実行時高速化技術を用いることにした．また，C 言語レベルで実現できる移植性の高い最適化に取り組み，どのような環境でも高速に実行できる処理系の開発を目指した．コンパイル時には，のぞき穴最適化により，不要な命令の除去を行う様にした．また，後述する最適化用命令への，命令列のパターンマッチによる変換も実現した．

スタックマシンモデルの仮想マシンの主なオーバヘッドとしては，主に (1) 命令ディスパッチ，(2) 命令オペランドのフェッチ，(3) スタック操作 (4) 命令本体の実行に分類される．(1) は，各命令の分岐をどのように行うかが問題となるが，知られているもっとも容易で高速な手法であるダイレクトスレッデッドコードを適用した．また，頻出する命令パターンから新規命令を生成するオペランド融合，命令融合を行うことで，(1)，(2) のオーバヘッド削減を行った．そして，(3) のスタック操作オーバヘッドを削減するために，スタックトップを特別なレジスタに格納する静的スタックキャッシングを実装した．

(4) は，Ruby ではとくにメソッド呼び出しが頻出するので，このオーバヘッドの削減が重要である．そこで，(a) インラインメソッドキャッシュや (b) 特化命令を利用した．

(a) では，メソッド検索結果を命令列中にキャッシュする手法である．ある命令番地において呼び出すメソッドの実体はほぼ同一であるため，高いキャッシュ率が期待できる．ただし，メソッド再定義など，キャッシュを無効にする必要があるため，VM 状態カウンタを設けてキャッシュが有効かを判断するようにした．この手法は，キャッシュ参照のたびに VM 状態カウンタをチェックする必要があるが，メソッドの再定義時には VM 状態カウンタを変更するだけで済むため効率的である．(b) は，メソッドフレームの生成などの，メソッド起動のオーバヘッドに比べてメソッド本体の処理時間が十分に短い場合，たとえば整数演算などを，特別な命令に置き換える．ただし，Ruby では静的な解析ができないため，どのメソッド呼び出しが整数演算かわからないため，`a+b` のような，顕著な例について `opt_plus` というような特化命令へ変換する．`opt_plus` では，引数を検査し，整数同士の演算であれば，そして整数同士の演算が再定義されていなければ，その場で計算を行い実行結果を返し，そうでなければ通常のメソッド呼び出し処理を行う．現在，特化命令は 15 のメソッド呼び出しに対応している．

仮想マシンの構築は，文字の置き換えだけで行う簡単な VM 生成系を利用して行った．生成系利用者は VM 記述言語により命令オペランド，スタックオペランド，スタックへ命令終了時に格納する値，そして命令の本体を記述する．VM 生成系はこの記述から仮想マシン本体のプログラム片やコンパイラ，アセンブラ，逆アセンブラのプログラム片を自動的に生成する．また，融合命令，静的スタックキャッシングで利用する最適化用の命令を自動的に生成する．この仕組みを利用することで仮想マシンという複雑な処理系の保守性を向上し，高速化への試みを容易にすることができた．ただし，無制限に自動生成を行うとプログラム片が膨大な量になり，プロセッサの命令キャッシュミスを起こすなどの影響があることがわかった．

実際に仮想マシン YARV を実装して評価を行ったところ，旧 Ruby 処理系と比べてマイクロベンチマークで顕著な性能向上を確認することができた．とくに，整数演算を多用するプログラムでは特化命令による性能向上により，最大 25 倍ほどの性能向上を得ることができた．しかし，文字列演算や多倍長整数演算などのライブラリの実行時間が大部分を占めるプログラムでは，仮想マシンの導入による高速化はあまり得られなかった．そして，他のスクリプト言語の処理系と比較しても，仮想マシンの性能が勝っていることがわかった．

各最適化の効果はベンチマークプログラムの性質によってまちまちだが，多くのベンチマークにおいて，Ruby の仮想マシン化だけでも 2 倍程度の性能向上を達成することができた．また，仮想マシン化によって，従来は適用することが難しかった最適化技術を適用することが出来た．つまり，Ruby 処理系の仮想マシン化が Ruby の高速化の方針として

正しかったといえる．

8.1.2　Ruby の並列化

マルチコアプロセッサに代表される，コモディティ化した並列計算機上でプログラムを並列実行し，処理時間を短縮するという需要は大きい．とくに，スクリプト言語のような記述力の高い言語で，並列化により性能向上を目指す意義は大きい．

そこで，Ruby を並列実行により高速化することとした．Ruby には，言語レベルで並行実行，つまり論理的な実行単位の多重化をサポートするためのスレッド処理機構を有している．この Ruby スレッドを物理的，時間的に並列に実行するように拡張するのは自然な拡張であり，既存の Ruby のマルチスレッドプログラムも並列実行によって性能向上が実現できる．

旧 Ruby 処理系では，Ruby スレッドを独自実装したユーザレベルスレッドを利用して実現していた．この方式は，移植性が高いという利点があるが，Ruby スレッドを並列実行することができず，スレッド切り替えも低速であるという問題点があった．この問題を解決するために，OS などが提供するネイティブスレッドを利用して Ruby スレッド処理機構を実現することとした．

ネイティブスレッドを利用するにあたっては，移植性や実装の単純さを考慮して Ruby スレッド 1 つに対してネイティブスレッド 1 つを対応させる方式を取ることにした．この方式では，ネイティブスレッドの生成など，制御オーバヘッドが問題になるが，一般的なマルチスレッドプログラミングにおいて，プログラムの実行中にはスレッドの生成を頻発することはないため，性能に影響がないと判断した．ネイティブスレッド制御機能には含まれていない Ruby スレッドの制御である合流と割り込みについては，ネイティブメソッドの機能を組み合わせて対応する必要があった．

Ruby スレッドを並列実行するにあたり，仮想マシンがメモリ保護違反などを起こさない安全な実行を実現するには適切な排他制御が必要になる．YARV において排他制御が必要になるのは (1) スレッド間で共有する管理データ，(2) ガーベージコレクション（GC）を含んだオブジェクトの管理，(3) インラインキャッシュ，(4) スレッドセーフでないネイティブメソッドの実行である．

(1) は，スレッド間で共有するデータは，たとえばメソッド名とメソッドの実体のように名前と値の組で構成された表で管理されているので，この表へのアクセスを排他制御することで適切な排他制御が実現できた．(2) では，GC 実行時にはすべてのスレッドを合流して逐次 GC を行うようにした．オブジェクトの生成にはスレッドローカルフリーリス

トを用意して，生成時に排他制御しなくても良い構造にした．(3) は，インラインキャッシュアクセスのたびに排他制御するのは速度的に問題なので，キャッシュを更新する際には毎回新しいキャッシュエントリを生成するようにした．更新時には生成オーバヘッドがかかるが，アクセス時には排他制御が必要ない．一般的にキャッシュヒット率は高いので，本方式が有利と判断した．

(4) は，過去の膨大なプログラム資産の活用に関する互換性の問題である．実行効率を考えると，スレッドセーフでないネイティブメソッドをすべて細粒度排他制御を利用したスレッドセーフな実装に書き換えることで並列性を向上させることが最良であるが，一挙にこれを行うのは現実的ではない．ライブラリの多くが利用不可能となれば，Ruby 処理系としての実用性を著しく損ねることになり問題である．そこで，スレッドセーフでないネイティブメソッドの実行時は VM に一つ用意するジャイアントロック（GL）を獲得し，終了後には解放するような実装とした．ネイティブメソッドからは Ruby のメソッドを起動することが出来るため，その際には GL を解放するようにした．GL の競合によるスレッド切り替えが多発すると，逐次実行に比べ性能が格段に落ちることがわかった．そこで，GL 競合を監視する仕組みを設け，競合回数が閾値をこえると利用するプロセッサエレメントを制限し，競合による性能低下を抑える仕組みを設けた．

実際に並列に実行する Ruby スレッド処理機構を実現し，合計 8 プロセッサコアを持つメモリ共有型並列計算機上で評価した．まず，ネイティブスレッドを利用した Ruby スレッド処理機構の性能評価を行った．Ruby スレッドの生成，合流に関してネイティブスレッド制御のオーバヘッドによる性能低下が起きた．また，生成時には VM 化による，VM 用データ構造の割り当てオーバヘッドも性能低下の原因であることがわかった．スレッド切り替えは旧 Ruby 処理系に比べ格段に性能が向上した．

次に，並列実行による性能向上を計測したところ，8 プロセッサコア上で実行して，最大 7.4 倍の性能向上を達成することができ，台数効果を確認することができた．これは，(a) GL を必要としない処理，および (b) GL を必要とする Ruby スレッドを 1 つと必要としない処理を混合した処理において確認できた．多くの Ruby オブジェクトを生成するプログラムでは，逐次実行型のガーベージコレクションがボトルネックになり，台数効果が制限される結果となった．GL の競合が頻発するプログラムの場合，大きな性能低下が発生したが，動的な利用 CPU 制限により，性能低下を抑え，最悪値に比べて 3.4 倍の性能向上を達成することができた．

GC，および GL が不要な処理，たとえば記号処理や整数演算については台数効果を確認できた．今後，並列 GC の実装，ネイティブメソッドのスレッドセーフ化が進むことで，一般的で実用的な Ruby プログラムの，並列実行による高速化が可能になると思われ

る．また，十分なスレッドセーフ化が行われていなくても，処理時間の多くを占めているネイティブメソッドのみをスレッドセーフ化することで，並列実行による高速化を実現可能とした．

8.2 今後の課題

本研究ではさまざまな Ruby プログラムを高速に実行するための様々な手法を検討し実際に開発・評価したが，Ruby 処理系開発についての課題はまだ多い．

YARV には他にも高速化を行う余地がある．たとえば，メソッド呼び出しや，Ruby で多用されるブロック呼び出しのオーバヘッドを削減するためのインライン化は今後の高速化の上で重要な課題である．ただし，Ruby の動的な仕様から，コンパイル時にインライン化可能か完全に解析することはできないので，コンパイル時に解析した条件が変わった時点で，この最適化を無効にするなどの仕組みが必要がある．これは，オンスタックリプレースメント [16] などの技術を利用すれば可能となるが，Ruby ではこの処理が頻出する可能性があるので，出来る限り実行コストが少ない手法を検討する必要がある．

また，他の言語処理系で多く行われている実行時コンパイラの開発を進めることも一つの高速化の手段である．しかし，他の言語処理系と違い YARV 命令は 1 命令の粒度が大きいため実行時コンパイラによる最適化の効果が小さい可能性がある．今後，実行時コンパイラの開発コストと得られる高速化のトレードオフを勘案しながら検討する．実行時コンパイラ以外にも，実行前に先にコンパイルする AOT コンパイル技術がある．AOT コンパイラを利用して先にコンパイルしておけば，インタプリタ型言語処理系で問題となる実行時のプログラムの読み込み，およびコンパイルする時間になることが期待できる．また，C 言語などにコンパイルすることで，C 言語用コンパイラに搭載された最適化機構を利用することができるので，より高速に実行することができると期待できる．

並列化に関しても，現状ではジャイアントロックが必要となる処理が多いため，並列計算機を十分に利用できているとは言い難い．この対策は，地道にスレッドセーフコードを追加していくことで対応ができるため，今後行うべき課題である．とくに，文字列，配列などの基本的なデータ構造を扱うネイティブメソッドがスレッドセーフとなっていないのは問題であるため，対応が必要である．また，オブジェクト指向並列プログラミング言語である Ruby には，並列 GC の対応も大きな課題である．

その他の機能としては，たとえば VM のマルチ VM 化機能（MVM）[6, 48] の追加が期待される．現在はその CRuby の特性から 1 プロセスにつき 1 VM しか生成することができなかった．しかし，今回の VM 化に伴いインタプリタを構成するためのデータ構造

が整理されたことから，複数の VM を 1 プロセス中に実行することが可能になる．MVM は Ruby 処理系を他のアプリケーションに組み込み，いくつもの Ruby 環境を使い分けたい場合に特に有用である．

また，Ruby プログラムを並列化する単位が Ruby スレッドのみというのも問題である．他のプログラミング言語より容易に操作することができるとはいえ，マルチスレッドプログラミングは本質的に難しい．とくに，一つの命令流を並列化のために複数に分割して性能を向上させるプログラムを記述するのは難しい．そこで，スレッド以外にプログラムを並列化する手法を検討しなければならない．たとえば，MVM の各インスタンスを並列に実行させるという手法が考えられる．MVM の各インスタンス間では，依存関係をなくすことが可能であるため，同期処理の問題が発生しないというメリットがある．今後，このようなスレッド以外の並行実行単位，およびその並列化手法を検討していく必要がある．

8.3 まとめ

計算機システムの需要の増大とともに，ソフトウェアに対する要求も大きくなり，迅速な開発が求められている．そこで，Ruby のような記述力の高いプログラミング言語が重要である．しかし，Ruby のようなスクリプト言語処理系の高速化はあまり行われていなかった．

本研究では，高速な Ruby 処理系を構築するために，Ruby 処理系の仮想マシン化，および Ruby スレッドの並列実行による Ruby の並列化を行った．仮想マシンには，既知の最適化を Ruby 向けにアレンジして適用した．Ruby スレッドの並列化はネイティブスレッドを利用し，適切な排他制御を仮想マシン側で行うことで実現した．

評価を行い，仮想マシン化により逐次実行を高速に実行できることを確認し，仮想マシンを用いた Ruby 処理系が Ruby の高速化に十分寄与することを確認した．また，並列計算機上での評価の結果，台数効果を得ることができた．並列計算機がコモディティ化されている現在，Ruby のような記述力の高い言語による並列プログラミングは今後ますます重要になるため，この成果の意義は大きい．

本研究で述べた手法はすべて移植性や過去の Ruby 処理系との互換性を考慮してあるため実用的な処理系の実現手法の提案でもある．VM 生成系を利用した仮想マシン開発を行い，保守性の向上も実現した．

本研究で開発した仮想マシン YARV は次期 CRuby の公式リリースである Ruby 1.9.1 に搭載される予定である．このバージョンがオープンソースソフトウェアとして公開された暁には，本研究を進める上で開発したソフトウェアが世界中の Ruby プログラマに利用

されることなる．つまり，これまで Ruby プログラムの実行は遅いために Ruby を利用することが出来なかったユーザに対し，速度の問題を緩和し，利用可能領域を広めるという本研究の目的を達成することが出来た．このように，本研究は研究的意義だけでなく，実用的意義の貢献も行うことが出来たのは大きな成果である．

付録 A

YARV 命令一覧

本付録では YARV の命令一覧を記載する．命令はカテゴリごとに表としてまとめている．表には各命令ごとに命令名，オペランド，スタックの状態遷移，およびその命令の解説を記載した．オペランドは命令オペランドを意味している．

スタックの状態遷移は，→ の左に記載した変数の数だけスタックからポップし，その値を利用して命令を実行し，命令終了後に → の右に記載した変数の数だけスタックへプッシュする，という意味である．変数名が ... となっている場合，スタックへいくつプッシュされるかは状況によって異なることを意味している．

なお，本付録の命令一覧は VM 生成系を利用してほぼ自動的に作成した．

A.1 基本命令

A.1.1 nop カテゴリ

命令名	オペランド	スタックの状態遷移	解説
nop		→	何もしない．命令数の調節に利用する．

A.1.2 variable カテゴリ

命令名	オペランド	スタックの状態遷移	解説
getlocal	idx	$\longrightarrow$ val	idx で指定されたローカル変数を得る.
setlocal	idx	val $\longrightarrow$	idx で指定されたローカル変数を val に設定する.
getspecial	$key, type$	$\longrightarrow$ val	特殊なローカル変数（$, $_, ...）の値を得る.
setspecial	key	obj $\longrightarrow$	特別なローカル変数（$, $_, ...）の値を設定する.
getdynamic	$idx, level$	$\longrightarrow$ val	level, idx で指定されたブロックローカル変数の値を得る.
setdynamic	$idx, level$	val $\longrightarrow$	level, idx で指定されたブロックローカル変数の値を val にする.
getinstancevariable	id	$\longrightarrow$ val	self のインスタンス変数 id の値を得る.
setinstancevariable	id	val $\longrightarrow$	self のインスタンス変数 id を val にする.
getclassvariable	id	$\longrightarrow$ val	現在のスコープのクラス変数 id の値を得る.
setclassvariable	id	val $\longrightarrow$	klass のクラス変数 id を val にする.
getconstant	id	$klass$ $\longrightarrow$ val	定数 id の値を得る.
setconstant	id	$val, klass$ $\longrightarrow$	定数 id の値を val にする.
getglobal	$entry$	$\longrightarrow$ val	グローバル変数 id の値を得る.
setglobal	$entry$	val $\longrightarrow$	グローバル変数 id の値を設定する.

A.1.3 put カテゴリ

命令名	オペランド	スタックの状態遷移			解説
putnil			→	val	スタックに nil をプッシュする.
putself			→	val	スタックに self をプッシュする.
putobject	val		→	val	オブジェクト val をスタックにプッシュする.
putstring	str		→	val	文字列ををコピーしてスタックにプッシュする.
concatstrings	num	$...$	→	val	スタックトップの文字列を num 個連結し，結果をスタックにプッシュする.
tostring		val	→	val	to_str の結果をスタックにプッシュする.
toregexp	opt	str	→	val	文字列 str を正規表現にコンパイルしてスタックにプッシュする.
newarray	num	$...$	→	val	新しい配列をスタック上の num 個の値で初期化して生成しプッシュする.
duparray	ary		→	val	配列 ary を dup してスタックにプッシュする.
expandarray	$num, flag$	$..., ary$	→	$...$	スタックトップのオブジェクトが配列であれば，それを num 個に展開する.
concatarray		$ary1, ary2st$	→	ary	二つの配列 ary1, ary2 を連結しスタックへプッシュする.
splatarray	$flag$	ary	→	obj	配列 ary に対して to_splat を呼び出す.
checkincludearray	$flag$	obj, ary	→	$obj, result$	配列 ary に要素 obj が含まれているかチェック. case/when で利用する.
newhash	num	$...$	→	val	新しいハッシュをスタックトップの num 個を初期値として生成する.
newrange	$flag$	$low, high$	→	val	Range.new(low, high, flag) のようなオブジェクトを生成しスタックにプッシュする.
putnot		obj	→	val	!obj の結果をスタックにプッシュする.

A.1.4 stack カテゴリ

命令名	オペランド	スタックの状態遷移	解説
pop		$val \longrightarrow$	スタックから一つポップする.
dup		$val \longrightarrow val, val$	スタックトップをコピーしてスタックにプッシュする.
dupn	n	$... \longrightarrow ...$	スタックトップの n 個をコピーしてスタックにプッシュする.
swap		$val, obj \longrightarrow obj, val$	スタックトップの 2 つの値を交換する.
reput		$..., val \longrightarrow val$	スタックキャッシングの状態を調整するために必要な命令.
topn	n	$... \longrightarrow val$	スタックトップから n 個目をスタックにプッシュする.
setn	n	$..., val \longrightarrow val$	スタックトップの値を n 個目のスタックにコピー
emptstack		$... \longrightarrow ...$	current stack を空にする.

A.1.5 setting カテゴリ

命令名	オペランド	スタックの状態遷移	解説
definemethod	$id, body, is\_singleton$	$obj \longrightarrow$	（特異）メソッド id を body として定義する.
alias	$v\_p, id1, id2$	$\longrightarrow$	alias を作る．もし v_p が Qtrue なら, valias (global variable) を作る.
undef	id	$\longrightarrow$	undef を行う.
defined	$type, obj, needstr$	$v \longrightarrow val$	defined? を行う.
postexe	$blockiseq$	$\longrightarrow$	END に対応するためにブロックを登録する.
trace	nf	$\longrightarrow$	trace 用の命令.

A.1.6 class/module カテゴリ

命令名	オペランド	スタックの状態遷移	解説
defineclass	$id, klass\_iseq, define\_type$	$cbase, super \longrightarrow val$	クラス定義スコープへ移行する.

A.1.7 method/iterator カテゴリ

命令名	オペランド	スタックの状態遷移	解説
send	$op\_id, op\_argc, blockiseq, op\_flag, ic$	... → val	メソッド呼び出しを行う.
invokesuper	$op\_argc, blockiseq, op\_flag$	... → val	super を実行する.
invokeblock	$num, flag$	... → val	yield を実行する.
leave		val → val	このスコープから抜ける.
finish		val → val	VM loop から抜ける.

A.1.8 exception カテゴリ

命令名	オペランド	スタックの状態遷移	解説
throw	$throw\_state$	$throwobj$ → val	大域ジャンプを行う.

A.1.9 jump カテゴリ

命令名	オペランド	スタックの状態遷移	解説
jump	dst	→	PC を (PC + dst) にする.
branchif	dst	val →	もし val が false か nil でなければ, PC を (PC + dst) にする.
branchunless	dst	val →	もし val が false か nil ならば, PC を (PC + dst) にする.

A.2 最適化用命令

命令名	オペランド	スタックの状態遷移	解説
getinlinecache	ic, dst	$\rightarrow$ val	インラインキャッシュが有効なら，値をスタックにプッシュして dst へジャンプする．
onceinlinecache	ic, dst	$\rightarrow$ val	once を実現する．
setinlinecache	dst	val $\rightarrow$ val	インラインキャッシュの値を設定する．
opt_case_dispatch	$hash, else\_offset$	$..., key$ $\rightarrow$	case 文で，可能なら表引きでジャンプする．
opt_checkenv		$\rightarrow$	将来の拡張用．
opt_plus		$recv, obj$ $\rightarrow$ val	特化命令： X+Y．
opt_minus		$recv, obj$ $\rightarrow$ val	特化命令： X-Y．
opt_mult		$recv, obj$ $\rightarrow$ val	特化命令： X*Y．
opt_div		$recv, obj$ $\rightarrow$ val	特化命令： X/Y．
opt_mod		$recv, obj$ $\rightarrow$ val	特化命令： X%Y．
opt_eq		$recv, obj$ $\rightarrow$ val	特化命令： X==Y．
opt_lt		$recv, obj$ $\rightarrow$ val	特化命令： X<Y．
opt_le		$recv, obj$ $\rightarrow$ val	特化命令： X<=Y．
opt_gt		$recv, obj$ $\rightarrow$ val	特化命令： X>Y．
opt_ge		$recv, obj$ $\rightarrow$ val	特化命令： X>=Y．
opt_ltlt		$recv, obj$ $\rightarrow$ val	特化命令： X<<Y．
opt_aref		$recv, obj$ $\rightarrow$ val	特化命令： recv[obj]．
opt_aset		$recv, obj, set$ $\rightarrow$ val	特化命令: recv[obj] = set．
opt_length		$recv$ $\rightarrow$ val	特化命令： recv.length()．
opt_succ		$recv$ $\rightarrow$ val	特化命令： recv.succ()．
opt_regexpmatch1	r	obj $\rightarrow$ val	最適化された正規表現マッチ．
opt_regexpmatch2		$obj2, obj1$ $\rightarrow$ val	最適化された正規表現マッチ 2
opt_call_c_function	$funcptr$	$\rightarrow$	ネイティブコンパイルしたメソッドを起動．

謝辞

本研究は，筆者が東京農工大学大学院知能・情報工学専修博士後期課程，および東京大学大学院情報理工学系研究科助手（助教）に在籍中にまとめたものです．本研究を進めるにあたり，本当に多くの方々にご指導，ご助言を頂きましたことをここに感謝いたします．

本論文の主査を引き受けてくださった東京大学大学院の竹内郁雄教授には，筆者を東京大学の教員へと誘ってくださり，教員としてのイロハをご教示頂きました．深く感謝いたします．

東京大学大学院の平木敬教授，萩谷昌己教授，稲葉真理准教授には本論文の副査をお引き受けいただきました．また，研究や教育に関して大変有意義なのご指導いただきました．深く感謝いたします．

東京工業大学の千葉滋准教授には本論文の副査をお引き受けいただいたほか，2 度の IPA 未踏ソフトウェア創造事業のプロジェクトの採択およびプロジェクトマネージャとして面倒を見ていただきました．研究を社会へ還元する，その姿勢を学ばせていただきました．深く感謝いたします．

東京農工大学大学院の並木美太郎教授には，筆者が研究室配属された 2002 年より，学士，修士，博士課程における学生時代の指導教員として，そして筆者が就職した現在まで，長い間大変お世話になりました．研究内容はもとより，研究の基礎から応用まで幅広くご指導いただきました．筆者の研究に対する姿勢はすべて並木先生から教えていただきました．深く感謝いたします．

早稲田大学筧捷彦教授には IPA 未踏ソフトウェア創造事業未踏ユースに採択して頂き，プロジェクトマネージャとして面倒を見て頂きました．本研究を進めることができたのも，まず最初にこのご支援を頂くことが出来たためです．深く感謝いたします．

筑波大学の前田敦司准教授には言語処理系に関する多くの知識をご教授いただきました．また，論文執筆にあたっては共著をお引き受けいただきました．本研究にも強く影響を受けております．深く感謝いたします．

東京農工大学工学部情報学部の先生方には学部時代から多くのご指導を頂きました．深く感謝いたします．また，先輩，同輩，後輩諸氏にはたくさんの助言や励ましを頂きました．改めて感謝いたします．

東京大学大学院情報理工学系研究科創造情報学専攻の先生方，および学生の皆様には研究および教育活動にあたり多くの助言を頂きました．深く感謝いたします．

株式会社ネットワーク応用通信研究所のまつもとゆきひろ氏には YARV 開発にあたりたくさんの協力や助言を頂き，また方式の議論をして頂きました．まつもと氏が開発した Ruby 処理系の評価器を YARV に置き換えることについても快く許可頂きました．まつもと氏が Ruby という素晴らしいプログラミング言語を生んでいなければ，また処理系をオープンソースソフトウェアとして公開していなければ，本研究は成り立たなかっただろうと思います．深く感謝いたします．

YARV 開発コミュニティ，Ruby 開発コミュニティ，および Ruby 利用者コミュニティの方々には本研究を進めるにあたっての助言を多数いただきました．深く感謝いたします．とくに，『Ruby ソースコード完全解説』を執筆し，筆者に Ruby 処理系についての正しい理解をもたらしてくれた青木峰郎氏には深く感謝いたします．

本研究は独立行政法人情報処理推進機構未踏ソフトウェア創造事業「2004 年度未踏ユース」，および「2005 年度上期未踏」，「2006 年度下期未踏」に採択され支援を受けました．ここに深く感謝いたします．

参考文献

[1] Bowen Alpern, et al. The Jalapeno virtual machine. *IBM Systems Journal, Java Performance Issue*, Vol. 39, No. 1, 2000.

[2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, Vol. 10, No. 1, pp. 53–79, 1992.

[3] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. A survey of adaptive optimization in virtual machines. In *Proceedings of the IEEE*, Vol. 93, pp. 449 – 466, February 2005.

[4] Marc Berndl, Benjamin Vitale, Mathew Zaleski, and Angela Demke Brown. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pp. 15–26, Washington, DC, USA, 2005. IEEE Computer Society.

[5] Jr. Brent W. Benson. Javascript. *SIGPLAN Not.*, Vol. 34, No. 4, pp. 25–27, 1999.

[6] Grzegorz Czajkowski and Laurent Daynés. Multitasking without comprimise: a virtual machine evolution. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pp. 125–138, New York, NY, USA, 2001. ACM.

[7] Ryan Davis and Eric Hodel. ruby2c automatic translartion of ruby code to c. http://zenspider.com/ ryand/Ruby2C.pdf.

[8] Ulrich Drepper and Ingo Molnar. The new native posix thread library for linux : Nptl. http://people.redhat.com/drepper/nptl-design.pdf : Draft.

[9] Anton Ertl. Threaded code. http://www.complang. tuwien.ac.at /forth/threaded-code.html.

[10] M. Anton Ertl. Stack caching for interpreters. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pp. 315–327. ACM Press, 1995.

[11] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pp. 278–288. ACM Press, 2003.

[12] M. Anton Ertl and David Gregg. The structure and performance of *Efficient* interpreters. *The Journal of Instruction-Level Parallelism*, Vol. 5, , November 2003. http://www.jilp.org/vol5/.

[13] M. Anton Ertl and David Gregg. Combining stack caching with dynamic superinstructions. In *Interpreters, Virtual Machines and Emulators (IVME '04)*, pp. 7–14, 2004.

[14] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. vmgen: a generator of efficient virtual machine interpreters. *Softw. Pract. Exper.*, Vol. 32, No. 3, pp. 265–294, 2002.

[15] M. Ertl and D. Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, Vol. 5, pp. 1–25, 2003.

[16] Stephen J. Fink and Feng Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pp. 241–252, Washington, DC, USA, 2003. IEEE Computer Society.

[17] Free Software Foundation (FSF). Gcc home page. http://gcc.gnu.org/.

[18] Jeffrey E. F. Friedl, 田和勝 (訳). 詳説 正規表現 第 2 版. O'Reilly Japan, Inc, 2003.

[19] Brent Fulgham. The computer language shootout benchmarks. http://shootout. alioth. debian.org/.

[20] Rob Gordon, 林秀幸 (訳). JNI:Java Native Interface プログラミング. ソフトバンククリエイティブ, 10 1998.

[21] Yan Gu, B. S. Lee, and Wentong Cai. Evaluation of java thread performance on two different multithhreaded kernels. *Operating Systems Review*, Vol. 33, No. 1, pp. 34–46, 1999.

[22] David Heinemeier Hansson. Ruby on rails: Web development that doesn't hurt. http://www.rubyonrails.org.

[23] Janice J. Heiss. The multi-tasking virtual machine: Building a highly scalable jvm. http://java.sun.com/developer/ technicalArticles/Programming/mvm/, 2005.

[24] Jim Hugunin. A dynamic language runtime (dlr). http://blogs.msdn.com/hugunin/archive/2007/04/30/a-dynamic-language-runtime-dlr.aspx, 4 2007.

[25] IEEE. *ISO/IEC 9945-1 ANSI/IEEE Std 1003.1*, 1996.

[26] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of lua. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pp. 2–1–2–26, New York, NY, USA, 2007. ACM.

[27] Shiro Kawai. Gauche - a scheme interpreter. http://www.shiro.dreamhost.com/scheme/gauche/index-j.html.

[28] Alexander Kellett. rubydium. http://rubyforge.org/ projects/rubydium/.

[29] John Lam. Rubyforge: Ironruby: Project info. http://rubyforge.org/projects/ironruby.

[30] Jesse Liberty, 鈴木幸敏 (訳), 首藤一幸 (訳), 情報技研 (訳). プログラミング C# 第 4 版. オライリージャパン, 2 2006.

[31] Yukihiro Matsumoto and et.al. Ruby application archive. http://raa.ruby-lang.org/.

[32] Microsoft. Microsoft .net information. http:// www.microsoft.com/net/.

[33] Microsoft. 共通言語ランタイム (clr). http://www.microsoft.com/japan/msdn/netframework/ programming/clr/.

[34] Nicholas Mitchell, Larry Carter, Jeanne Ferrante, and Dean Tullsen. Ilp versus tlp on smt. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, p. 37. ACM Press, 1999.

[35] Charles Oliver Nutter and Thomas E Enebo. Jruby java powered ruby implementation. http://jruby.codehaus.org/.

[36] Inc. O'Reilly Media. Perl.com: The source for perl – perl development, perl conferences. http://www.perl.com/.

[37] John K. Ousterhout. Scripting: Higher-level programming for the 21st century.

Computer, Vol. 31, No. 3, pp. 23–30, 1998.

[38] Steven Pemberton and Martin Daniels. *Pascal Implementation: The P4 Compiler and Interpreter*. Ellis Horwood, 1982.

[39] Evan Phoenix. The rubinius project. http://rubini.us/.

[40] Ruben Pinilla and Marisa Gil. Jvm: platform independent vs. performance dependent. *Operating Systems Review*, Vol. 37, No. 2, pp. 44–56, 2003.

[41] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pp. 291–300. ACM Press, 1998.

[42] Python Software Foundation. Python programming language. http://www.python.org/.

[43] RubyCentral. Rubyforge. http://rubyforge.org/.

[44] RubyGarden. Ruby: Virtualmachineoptions. http://www.rubygarden.org/ruby?VirtualMachineOptions.

[45] Koichi Sasada. Yarv: Yet another ruby vm. http://www.atdot.net/yarv/.

[46] KOICHI SASADA, MIKIKO SATO, KANAME UCHIKURA, YOSHIYASU OGASAWARA, HIRONORI NAKAJO, and MITARO NAMIKI. A lightweight synchronization mechanism for an smt processor architecture(processor architectures). 情報処理学会論文誌. コンピューティングシステム, Vol. 46, No. 16, pp. 14–27, 20051215.

[47] Yunhe Shi, David Gregg, Andrew Beatty, and M. Anton Ertl. Virtual machine showdown: stack versus registers. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pp. 153–163, New York, NY, USA, 2005. ACM.

[48] Sunil Soman, Laurent Daynès, and Chandra Krintz. Task-aware garbage collection in a multi-tasking virtual machine. In *ISMM '06: Proceedings of the 5th international symposium on Memory management*, pp. 64–73, New York, NY, USA, 2006. ACM.

[49] Douglas Stockwell and et.al. Gardens point ruby.net compiler. http://www.plas.fit.qut.edu.au/Ruby.NET/.

[50] Bjarne Stroustrup, 長尾高弘 (訳). プログラミング言語 C++ 第 3 版. アジソンウェスレイパブリッシャーズジャパン, 12 1998.

[51] Dan Sugalski. perlthrtut - tutorial on threads in perl.

http://search.cpan.org/ jhi/perl-5.8.1/pod/perlthrtut.pod.

[52] Sun Microsystems. Java テクノロジ. http:// jp.sun.com/java/.

[53] Satoru Takabayashi. pdumpfs: Plan9 もどきのバックアップシステム. http://0xcc.net/pdumpfs/.

[54] The Perl Foundation. Parrot - parrotcode. http://www.parrotcode.org/.

[55] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby*. The Pragmatic Programmers, 2004.

[56] Dean M. Tullsen, Susan Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the22th Annual International Symposium on Computer Architecture*, pp. 392–403, 1995.

[57] David Ungar and Randall B. Smith. Self. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pp. 9–1–9–50, New York, NY, USA, 2007. ACM.

[58] Alfred V.Aho, Brian W.Kernighan, and Peter J.Weinberger. プログラミング言語 AWK. シイエム・シイ, 3 2001.

[59] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: scalable threads for internet services. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pp. 268–281, New York, NY, USA, 2003. ACM Press.

[60] xue.yong.zhi. Xruby. http://xruby.com/default.aspx.

[61] Mathew Zaleski, Marc Berndl, and Angela Demke Brown. Mixed mode execution with context threading. In *CASCON '05: Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, pp. 305–319. IBM Press, 2005.

[62] Mathew Zaleski, Angela Demke Brown, and Kevin Stoodley. Yeti: a gradually extensible trace interpreter. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pp. 83–93, New York, NY, USA, 2007. ACM.

[63] 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.

[64] 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/.

[65] B.W. カーニハン, D.M. リッチー, 石田晴久 (訳). プログラミング言語 C 第 2 版. 共

立出版, 6 1989.

[66] 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.

[67] 河内谷, 古関, 小野寺. スレッド局所性を利用した java ロックの高速化. 情報処理学会論文誌 (PRO), Vol. 44, No. SIG 15 (PRO19), pp. 13–24, 11 2003.

[68] 多田好克, 寺田実. 移植性・拡張性に優れた c のコルーチンライブラリー実現法. 電子情報通信学会論文誌, Vol. J73D-I, No. 12, pp. 961–970, 12 1990.

[69] 安倍広多, 松浦敏雄, 谷口健一. Bsd unix 上での移植性に優れた軽量プロセス機構の実現. 情報処理学会論文誌, Vol. 36, No. 2, pp. 296–303, 2 1995.

[70] 松本行弘. Ruby の真実. 情報処理, Vol. 44, No. 5, pp. 515–521, 2003.

[71] 哲高林, 俊之増井. Quickml:手軽なグループコミュニケーションツール. 情報処理学会論文誌, Vol. 44, No. 11, pp. 2608–2616, 2003.

[72] 笹田, 佐藤, 河原, 加藤, 大和, 中條, 並木. マルチスレッドアーキテクチャにおけるスレッドライブラリの実現と評価. 情報処理学会論文誌：ACS, Vol. 44, No. SIG11(ACS3), pp. 215–225, Sep 2003.

[73] 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, 第 1 巻, pp. 305–306, 3 2005.

[74] 小野寺民也. オブジェクト指向言語におけるメッセージ送信の高速化技法. 情報処理, Vol. 38, No. 4, pp. 301–310, 1997.

[75] 青木峰郎. Ruby ソースコード完全解説. インプレス, 2002.

[76] 田中哲. Ruby I/O 機構の改善 – stdio considered harmful –. Linux Conference 抄録集, pp. 1–10, 2005.

[77] 田浦健次朗. 細粒度マルチスレッディングのための言語処理系技術. コンピュータソフトウェア, Vol. 16, No. 2, 3, pp. 1–19, 9–28, 1999.

[78] 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, 10 2003.

[79] 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, 4 2005.

[80] 真人木山. オブジェクト指向スクリプト言語 ruby への世代別ごみ集めの実装と評価. 情報処理学会論文誌. プログラミング, Vol. 42, No. 3, pp. 40–48, 2001.

[81] 真人木山, 大輔佐原, 孝夫津田. オブジェクト指向スクリプト言語 ruby への世代別ごみ集め実装手法の改良とその評価. 情報処理学会論文誌. プログラミング, Vol. 43, No. 1, pp. 25–35, 2002.