情報処理学会論文誌　プログラミング　Vol.8　No.1　1–10 (June 2015)

Ruby向けトレース方式Just-in-timeコンパイラの設計と実装

井出 真広[1,a)]　倉光 君郎[1,b)]

受付日 2014年10月11日, 採録日 2015年2月9日

概要：我々はオブジェクト指向スクリプト言語 Ruby を高速に実行するための処理系であるトレース方式 JIT コンパイラ，RuJIT の開発した．RuJIT は，Ruby 言語の C 言語による処理系 CRuby を対象に，CRuby が生成したバイトコードをトレースし，頻繁に実行された箇所やループを機械語へ変換する．本論文ではまず我々のトレース方式 JIT コンパイラの設計と実装について述べる．その後，Ruby 言語の持つ性質や機能に起因するトレースの構成や最適化が困難な点を紹介し，それを解決するための最適化手法について述べる．本論文では，RuJIT の設計と実装，および開発によって得られた知見について詳しく解説する．そして，RuJIT の性能について評価する．

キーワード：Just-in-time コンパイラ，スクリプト言語，Ruby

A Trace-based Just-in-time Compiler for Ruby

Masahiro Ide[1,a)]　Kimio Kuramitsu[1,b)]

Received: October 11, 2014, Accepted: February 9, 2015

Abstract: This paper describes RuJIT, a trace-based JIT compiler for Ruby which traces program code to determine frequently executed traces (hot paths and loops) in running programs and emits optimized machine code specialized to these traces. We first describe the design and implementation of the RuJIT compiler. Then we show that two Ruby's functions and characteristics make difficult to improve the performance of JIT compiled code. To produce better quality code, we describe solutions for these issues as well as implementation of optimization techniques.

Keywords: Just-in-time compiler, Scripting language, Ruby

1. はじめに

Ruby 言語 [1] は，オブジェクト指向スクリプト言語であり，高い生産性やプログラミングのしやすさから Web アプリケーション開発を中心に広く利用されている．Ruby 言語は，Smalltalk，Lisp などから着想を得た多くの機能を持ち，JavaScript や Python，Perl などの他のスクリプト言語と同様に動的型付けやメタプログラミングの機能，外部関数呼出機能による他の言語との高い相互運用性，クロージャなどの機能を備えている．

これらの言語機能はプログラムの書きやすさなど生産性を向上させる．その実装としてスクリプト言語の評価器はインタプリタとして実装されていることが多い．そのため Java の HotSpotVM [2] などの高速な処理系と比較すると，いまだに大きな性能差が存在する．これに対しスクリプト言語の開発者，研究者はスクリプト言語を高速に実行するための評価器や Just-In-Time（JIT）コンパイラを様々な言語に対して新たに実装している [3], [4], [5], [6], [7], [8], [9]．Ruby 言語でも Ishii ら [10]，MacRuby [11] や Rubinius [12] は，メソッド単位でコンパイルを行うメソッド方式 JIT コンパイラを導入し，それぞれの処理系で Ruby 言語の実行速度の改善を行ってきた．

本論文では Ruby 言語向けトレース方式 JIT コンパイラ RuJIT を提案する．トレース方式 JIT コンパイラとはプログラム上で頻繁に実行されるパス（たとえばループなど）をコンパイル対象とする JIT コンパイラである．

[1] 横浜国立大学大学院
Graduate School of Yokohama National University, Yokohama, Kanagawa 240–8501, Japan
[a)] imasahiro9@gmail.com
[b)] kimio@ynu.ac.jp

Ruby 言語では他の言語と異なりブロックと呼ばれる関数オブジェクトをメソッド内で繰り返し呼び出すイディオムを多用する．そのため単純なトレース選択の方法では関数オブジェクトの呼び出しを含むトレースを複数の異なる関数オブジェクトの呼び出しで利用することとなり，これはトレース作成時に構築した仮定条件の正しさの検査など実行オーバヘッドが生じる．本研究では関数オブジェクトが不変であると仮定をおき，不変式の巻き上げにより実行時オーバヘッドを減らす工夫をコード生成時に行い，この問題に取り組んだ．

本論文における，我々の貢献は以下のとおりである．

- 我々は，Ruby 向けトレース方式 JIT コンパイラを実装し，ベンチマークによる評価を行った．
- 我々は，Ruby 特有の問題に対して 2 つのコード生成の工夫を行い，実行速度への寄与について評価を行った．

本論文の構成は，次のとおりである．

本論文で対象とする Ruby 言語と，その処理系の概要を 2 章で述べる．3 章では，本論文で提案する JIT コンパイラについて概要を述べる．4 章では，RuJIT が生成したトレースの実行性能を最大化するため工夫した点について述べる．5 章で性能について評価を行い，6 章で関連研究を紹介した後，7 章でまとめと今後の課題を述べる．

2. Ruby 言語と CRuby 処理系

本章では, Ruby 言語と Ruby の C 言語実装である CRuby 処理系を紹介する．本論文の記述は CRuby 2.2.0-preview1 に基づく．

2.1 Ruby 言語

Ruby は動的型付けやガベージコレクタ，クロージャなどの機能を持つ動的型付きオブジェクト指向プログラミング言語である．Ruby ではブロックと呼ばれるステートメントの列から構成される手続きを関数オブジェクトとしてメソッド呼び出し時，引数として渡すことができる．

Ruby では，このブロックを引数にとるメソッド呼び出しを利用するイディオムが広く利用されている．とくに繰返し処理の記述において Ruby 利用者は while 式，until 式を用いた場合に比べ，ブロックを引数にとるメソッド呼び出しを用いることが多い[*1]．たとえば，1 から 20 までの和の計算はこのブロックを引数にとるメソッド呼び出しを用いると**図 1** のように記述する．ここで 12–15 行目は Ruby 利用者が記述するコード，1–11 行目までが Ruby 処理系が提供する Range クラスの each メソッドの実装となっており，each メソッドはそれぞれ引数として渡されたブロックを 10 回実行する．

[*1] なお，Ruby にはほかにも for 文も用意されているが for 文は each メソッドを呼び出す糖衣構文である．

```
1  # Ruby 処理系内部で定義される Range.each メソッド
2  class Range
3  def each(&block)
4    i = self.begin
5    last = self.end
6    while (i < last)
7      block.call(i)
8      i+=1
9    end
10 end
11 end
12 # 1から 20までの和の計算
13 sum = 0
14 (1..10).each {|i| sum += i }
15 (11..20).each {|j| sum += j }
```

図 1　ブロックを引数にとるメソッド呼び出しを利用したトイプログラムと Range.each メソッドの定義

Fig. 1　Definition of Range.each method and a toy program which uses blocks.

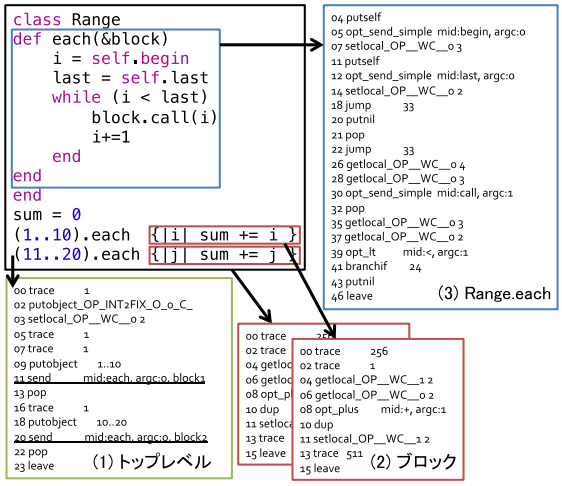

図 2　図 1 のバイトコード

Fig. 2　Bytecode for Fig.1.

2.2 CRuby 処理系

Ruby 言語の参照実装は CRuby と呼ばれ，CRuby ではバージョン 1.9 より YARV と呼ばれるスタックベースの仮想マシンを評価器として採用している．

YARV は実行時に Ruby プログラムをトップレベル式やメソッド，クラス定義文，ブロックをコンパイル単位として YARV バイトコードにコンパイルし，実行する．たとえば YARV は図 1 の Ruby プログラムを**図 2** に示すバイトコードに変換する．それぞれは (1) トップレベル式，(2) each メソッドの渡される 2 つのブロック，(3) each メソッドの合計 4 つのバイトコード列に対応している．生成されたバイトコードはトップレベルのバイトコードから順に実行され，send 命令で each メソッドのバイトコードが実行される．このとき，引数として，それぞれブロックのバイトコード列を保持したブロックオブジェクトを引数として

メソッド呼び出しが行われ，each メソッド内部でブロック内に保持されたバイトコード列がそれぞれ実行される．

3. RuJIT

本章では，Ruby 向けトレース方式 JIT コンパイル RuJIT について述べる．**図 3** に RuJIT の概要図を示す．RuJIT は YARV を拡張して構築しており，YARV 以外のものは変更をせず利用している．

3.1 RuJIT の全体像

RuJIT はトレース（プログラム中で頻繁に実行されるパス）を記録し，トレースのコンパイル，実行を行う JIT コンパイラである．その構成は (1) YARV バイトコードを入力とし，トレースの構築やコンパイル済みのトレースへのディスパッチを行うトレース選択器と (2) トレースした YARV バイトコードから RuJIT の中間表現に変換する IR 生成器，(3) 中間表現の最適化器，(4) 機械語への変換を行うコード生成器，(5) コンパイル済みトレースの管理を行うトレースキャッシュから構成される．

まず RuJIT は YARV がバイトコードを実行するごとにトレース選択器を起動する．トレース選択器は YARV が実行しているバイトコードの情報から 3.2 節で述べるアルゴリズムに従ってコンパイル対象となるバイトコード列の選択を行い，トレースの記録，もしくはコンパイル済みトレースの起動を行う．

トレースの記録は，実行された YARV バイトコードを実行時に行われた型検査，メソッドの検索結果とともに RuJIT 中間表現に変換したうえで記録する．

RuJIT はトレースの構築が完了すると，中間表現上での共通部分式除去，定数畳み込み，ループ不変式の巻き上げなどの最適化に加え，4.1 節で述べる最適化を施す．最後に，中間表現列を機械語に変換し，コンパイル済みトレースの構築が完了する．

3.2 トレース選択器

トレース選択器は，YARV が現在実行しているバイトコードをもとにトレースの起点の選択，トレースの記録，そしてトレース記録の停止を行う．

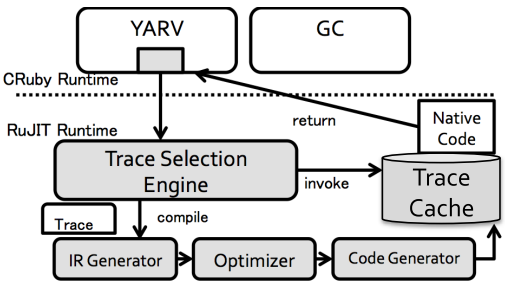

図 3　RuJIT の概要図
Fig. 3　An overview of RuJIT.

3.2.1 トレースの起点の選択

トレース選択器は，バイトコードが実行されるたびに以下のアルゴリズム（**図 4**）を適用し，コンパイル対象となるバイトコード列の選択を行う．なおトレースの選択には NET（Next Executing Tail）[13] の戦略を用いた．

トレース選択器はまずトレースの起点の候補を以下の 2 つの条件をもとに認識する（図 4 の 15 行目から 16 行目，11 行目から 12 行目）．

- 後方分岐（backward branch）命令
- 他のトレースの復帰点となる場所

1 つ目の条件はトレース選択器の入力が後方分岐命令のとき，このバイトコードをトレースの起点の候補とする．この条件によりトレース選択器は制御フローグラフを構築せずにプログラム中に存在するループ構造を近似的に検知する．2 つ目の条件は，コンパイル済みトレースの実行時にトレース構築に生成した仮定条件との不一致などによりトレースの実行を中断し，インタプリタへ実行を戻す際の復帰点（side exit）となるバイトコードの場所をトレースの起点の候補とする．この条件により，トレース選択器はすでに構築されたトレースに含まれていない実行パスもトレース対象にすることができる．

トレース選択器は，これらトレースの起点となりうる箇所に対して実行回数を記録するカウンタを割り当て，ト

```
1  void trace_selection(現在のプログラムカウンタ) {
2      if (mode == Record) {
3          if (トレース記録の終了条件を満たす) {
4              mode = Default;
5              トレースをコンパイルする
6          }
7          return;
8      }
9      if (code = FindCompiledCache(PC)) {
10         コンパイル済みトレースを実行
11         if (side exit が実行された)
12             現在のプログラムカウンタをトレース起点
                   候補とする
13         return;
14     }
15     if (isBackwardBranch(e))
16         現在のプログラムカウンタをトレース起点候補
               とする
17     if (counter = FindTraceCounter(PC)) {
18         counter += 1;
19         if (counter > Threshold)
20             mode = Record;
21     }
22     return;
23 }
```

図 4　トレース選択アルゴリズム
Fig. 4　Trace selection algorithm.

レースの起点の候補とする．トレースの起点の候補のうち，実行回数がしきい値を超えたものをトレースの起点とし，はじめてトレースの記録を開始する（図 4 の 17 行目から 21 行目）．

3.2.2　トレースの記録

トレースの記録は，YARV 上で実行されたバイトコードに加え，バイトコード実行時に行われた型検査やメソッド呼び出し時に行うメソッドの実体の検索，分岐命令の実行を記録する．このとき，RuJIT は同時にバイトコードから RuJIT 中間表現への変換を逐次行い，また型検査やメソッドの検索，分岐命令の実行はそれぞれのガード命令として変換される．例として**図 5** に加算を扱う YARV バイトコード opt_plus から RuJIT 中間表現に変換するコードを示す．

ここで，RuJIT はトレースの記録が行われていない場合のインタプリタの実行性能の劣化を抑えるため，YARV が持つバイトコード実行部分（図 5 の 1 行目から 11 行目）とは別に，トレース記録時に YARV の実行パスをガード命令として記録しながら中間表現の生成を行うバイトコード実行部分を持つ（図 5 の 12 行目から 30 行目）．RuJIT は，この 2 つのバイトコード実行部分を切り替えながらバイトコードの記録，実行を行う．ここで，**R_recv**，**R_obj** は変数 recv，obj の RuJIT 中間表現上での表現である．

```
1  case opt_plus:
2      VALUE recv = POP();
3      VALUE obj = POP();
4      if(recv は Fixnum && obj は Fixnum) {
5          if(Fixnum.+は再定義されていない)
6              val = recv + obj;
7          else
8              再定義されたFixnum.+の呼出
9      } else { ... }
10     PUSH(val);
11     次の命令のディスパッチ;
12 case opt_plus_on_record:
13     VALUE recv = POP();
14     VALUE obj = POP();
15     VALUE val;
16     if(recv は Fixnum && obj は Fixnum) {
17         EMIT(GuardFixnum, R_recv);
18         EMIT(GuardFixnum, R_obj);
19         if(Fixnum.+は再定義されていない) {
20             EMIT(GuardRedefine, Fixnum.+)
21             val = recv + obj;
22             EMIT(FixnumAddOverflow, R_recv, R_obj);
23         }
24         else
25             EMIT(GuardMethod, recv の型, R_recv)
26             EMIT(InvokeMethod, recv.+, R_recv, R_obj)
27             再定義されたFixnum.+の呼出
28     } else { ... }
29     PUSH(val);
30     次の命令のディスパッチ;
```

図 5　opt_plus の実行と RuJIT 中間表現への変換
Fig. 5　Bytecode handler for opt_plus in RuJIT.

3.3　トレース記録の終了条件

トレースの記録は以下に述べる条件が満たされるまで行われる．

- ループの検知
- トレースのバッファ溢れ
- 例外オブジェクトの throw
- C 言語で定義されたメソッドの実行

1 つ目の条件は，繰り返し実行されるバイトコード列のトレースの記録が完了したことを検知し，トレースの記録を終了する．本トレース選択器では，現在のプログラムカウンタが示すアドレスがトレースとして記録済みの場合にループであると検知する．

2 つ目の条件として，トレース記録器が持つトレース記録用バッファが溢れた場合，トレースの記録を終了する．これは，トレース選択器がパフォーマンスの観点から固定長のバッファを用いてトレースの記録を行うためである．

3 つ目，4 つ目の条件としてトレース選択器は例外オブジェクトの throw や C 言語で定義されたメソッドを実行した場合トレースの記録を終了する．ただし，コンパイル対象となるバイトコードの範囲を広げ，パフォーマンスを向上させるため，C 言語で定義されたメソッドの呼び出しのうち，4.2 節で後述する一部の場合は終了せずに記録を続ける．

トレース記録器は終了条件を満たすとバッファに記録されたトレースを機械語にコンパイルする（図 4 の 5 行目）．コンパイルされたトレースは，トレースキャッシュと呼ばれるトレース開始時のバイトコードをキーとするマップに保持される．トレース選択器はトレースキャッシュに現在のプログラムカウンタが含まれている場合，機械語にコンパイル済みのトレースを実行し処理を終了する（図 4 の 9 行目から 14 行目）．

4.　ランタイムオーバヘッドの削減

本論文では，RuJIT に対し，トレース方式 JIT コンパイラのランタイムオーバヘッドを削減する 2 つの工夫を行った．本章では，これらの工夫について述べる．

4.1　ブロックを考慮したトレースの構築

Ruby 言語では 2.1 節で述べたとおり，繰返しを引数にブロックをとるメソッドで実現することが多い．そのため繰返しを扱うメソッド内でブロックの呼び出しを行う処理はトレース選択器によりトレースの対象となりやすい．本節では図 1 で示した Ruby プログラムを簡略化したプログ

ラム（図 6）を例にトレースの構築を行う．

トレース選択器は図 4 で述べたアルゴリズムより図 6 の 8 行目から呼び出された each メソッド内に含まれるループとループ内で行われるブロック B1 をトレースとして構築する．構築されたトレースは，トレース実行時に渡されているブロックがトレース上にインライン展開されたブロックが持つ命令列が等しいかを検査し（以下，ブロックに対するガード命令と呼ぶ），同じ場合は展開されたブロックの実行を行い，異なる場合はインタプリタへ実行を遷移させる（図 7 Trace1A）．

ここで図 6 の 8 行目と 9 行目では，呼び出すメソッドは同じであるがメソッド呼び出し時に渡されるブロックが異なる．そのため 9 行目で呼び出される each メソッド内から呼ばれるトレース Trace1A はブロックに対するガード命令により実行が中断される．RuJIT は図 4 で述べたアルゴリズムより実行の中断した点を起点に新たにブロック B2 を呼び出すトレース（図 7 Trace1B）を構築し，Trace1B は Trace1A にリンクされる．

このようにブロックを引数にとるメソッドが複数の異なる呼び出しコンテキストから用いられる場合，ブロックに対するガード命令を起点にして構築されたトレースが生成され，トレース間の遷移は頻繁に行われる．そのため実行時に行うガード命令，トレースに展開されたブロックと呼び出すブロックが異なった場合の発生するトレースの遷移は大きな実行オーバヘッドとなる．

```
1 def each(&block)
2   while (A)
3     B
4     block.call(i)
5     C
6   end
7 end
8 each { B1; }
9 each { B2; }
```

図 6　図 1 を簡略化した Ruby プログラム
Fig. 6　Simplified version of Fig.1.

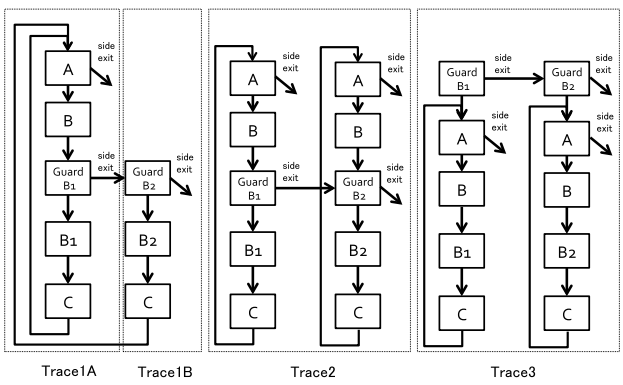

図 7　ブロックを考慮したトレースの構築例
Fig. 7　An example of traces for Fig.1.

CRuby ではブロックが参照する命令列はプログラム読み込み時に YARV バイトコードにコンパイルされ，プログラム実行時において不変である．ただし，この不変性は，ブロックに対する実行時書き換えを行った場合に失われる可能性がある．RuJIT では，あるトレース（以下，親トレースと呼ぶ）のブロックに対するガード命令をトレースの起点に構築されたトレース（以下，子トレースと呼ぶ）を構築した際，ブロックが参照する命令列が不変であると仮定をおき，ブロックに対するガード命令に対して不変式の巻上げの最適化を適応しトレースの遷移回数の削減を行う．

ブロックに対するガード命令の巻き上げは，子トレースの構築を完了し，親とレースに子トレースをリンクする際に，リンク時最適化の 1 つとして以下の手続きに基づき行われる．

まず親トレースと子トレースにおいて利用される同じブロックのうち，参照する命令列が不変なものとして以下の条件を満たす巻き上げの対象となるブロックを選択する．

(1) ブロックが 2 つのトレース間でプログラムの字面上で同じ名前を持ち，ブロックの生存区間の開始点が 2 つのトレースの間で同じ．
(2) 親トレース，子トレース，巻き上げ対象となっているブロックがいずれも動的評価（eval など）を含まない．
(3) ブロックがトレース構築の段階で他のメソッドやブロックから動的書き換え対象となっていない．

3 つの条件はそれぞれ，条件 1 は親トレースと子トレースで，ともにトレース中で呼び出すブロックが存在し，それぞれブロックの呼び出しがトレース内に展開されていることを保証している．そして条件 2 はブロックの保持するバイトコード列がトレースに展開されたコード中では書き換わらないことを保証している．また，Ruby 言語には Binding と呼ばれるローカル変数といった実行コンテキストを取得し動的書き換えを可能となっている．条件 3 は Binding によるブロックの書き換えが起こらないことを保証している．以上の条件を満たすブロックに関して，ブロックに対するガード命令の巻き上げを行う．

対象となるブロックが見つかった場合，巻き上げの前処理として，親トレースと子トレースそれぞれのコピーを作成する．コピーしたトレースは以降で述べるトレースの変形で利用し，オリジナルのトレースは後述する脱最適化を行う際に利用する．また，親トレースのうちトレースの起点からブロックに対するガード命令までを子トレースに複製する（図 7 Trace2）．

最後にブロックに対するガード命令について巻き上げを行い，ブロックの生存区間の開始点までをガード命令の巻き上げを行う（図 7 Trace3）．以上より得られたトレースはブロックに関する実行オーバヘッドを削減可能となる．

なお，ブロックに対するガード命令の巻き上げは対象と

表 1　トレースに含める C 言語で実装されたメソッド一覧
Table 1　List of Ruby methods which RuJIT includes into a trace.

Fixnum	-@, to_f, +, -, *, <, <=, >, >=, !=, ==, &, |, ; >>, <<
Float	-@, to_i, to_f, +, -, *, /, %,>, >=, <, <=, !=, ==
Array	[], []=, <<, +, -, &, |, uniq, to_s, to_a, at, fetch, first, last, concat, push, pop, shift, unshift, insert, length, join, reverse, sort, clear, fill
String	+, *, %, [], length, size, empty?, succ, next, upcase, downcase, split, inculude?
Object	new

表 2　RuJIT が再実装したメソッド一覧
Table 2　List of Ruby methods which RuJIT re-implements.

Enumerable	find, find_all, collect, map, reduce, all?, any?, one?, none?　each_with_index, inject, detect
Array	each, each_index, reverse_each, index, rindex
Fixnum	times, upto, downto

なるブロックが参照する命令列が不変であることを仮定においている．そのためプログラムの実行の過程で動的評価などが行われた場合，脱最適化を行い，巻き上げが行われたトレース（Trace3）を破棄し，巻き上げが行われる前の状態にトレースを復元する（図 7 Trace1A，Trace1B）．

4.2　トレース区間の最大化

CRuby では Ruby スクリプトで記述した際の実行性能の観点からいくつかのメソッドが C 言語で記述されている．とくに CRuby で用意されている基本クラス（String や Array など）のメソッドは C 言語で記述されている．しかしこれらメソッドはバイトコードからはどのような処理を行うか解析できないため，これらの C 言語で記述されたメソッドの呼び出しは頻繁なトレースの中断を引き起こし，これは多くの Ruby プログラムについて RuJIT がカバーできる範囲を狭める．

RuJIT では，C 言語で記述されたメソッドを以下に述べる工夫により対応を行った．

- トレースへの C 言語で記述されたメソッドの追加
- C 言語で記述された繰返しを行うメソッドの Ruby スクリプト化

まず我々は，C 言語で記述されたメソッドのうち，例外の throw や外部変数への書き込みを行わないメソッド（**表 1**）をトレースの終了条件に含めないよう変更を行った．これらのメソッド呼び出しをトレースに含めることが可能となればよりトレースがカバーする区間が長くなり，またトレースとインタプリタ間の遷移によるランタイムのオーバヘッドが削減できる．

同様にブロックを引数にとり，繰返しを扱うメソッドについてもすでに C 言語で記述されているメソッドが存在する．これらのメソッドの呼び出しは Ruby では頻繁に利用されるため，トレースの起点となりうるコードを含むメソッドであると考えられる．RuJIT では，これらのメソッドを Ruby スクリプトでの再実装を行い，トレース可能とした．なお，この方法では，再実装を行ったメソッドのうちトレースの対象とならないメソッドはインタプリタでの実行となるため，実行性能が劣化する可能性がある．RuJIT では，経験的にトレース対象となりやすい，繰返しを扱うメソッドのみを選択し Ruby スクリプトでの再実装を行った．**表 2** に再実装を行ったメソッド一覧を示す．

表 3　実験環境
Table 3　Experiment environment.

CPU	Intel(R) CoreTM i7 2.2 GHz
Memory	8 GB, 1333 MHz
OS	MacOSX 10.9.5
C compiler	Clang 3.5

5.　評価

本章では，RuJIT が Ruby アプリケーションの実行速度に与える影響を，ベンチマークプログラムを用いて評価した結果を示す．

5.1　実験環境

我々は**表 3** に示す実行環境を用いて RuJIT が Ruby アプリケーションの実行速度に与える影響の評価を行った．評価対象とする Ruby 処理系は CRuby-2.2 執筆時点の最新のバージョン（リビジョン 47854）を使用した．RuJIT のベースとなる Ruby 処理系も同じものを利用した．

なお本論文ではトレースとして選択された範囲の最大化やブロックの同値性検査巻上げの最適化の効果の評価に注力するため，我々は，RuJIT の内部表現から機械語へのコード生成器として既存の C コンパイラの 1 つである clang を用いて機械語への変換を行った．なお実験では，clang の最適化オプションは-O3 を利用している．

性能評価は CRuby に付属するマイクロベンチマークプログラム 36 種類を用いて評価を行った．各ベンチマークプログラムはそれぞれ 10 行程度から数百行程度の規模のプログラムである．

本章では次の項目について評価を行う．

- RuJIT がコンパイルした機械語の実行性能
- 4 章で導入した工夫による性能比較

なお，評価はベンチマークプログラムの実行にかかった実経過時間をそれぞれ 4 回計測し，実行時間が最も短いものを計測結果として選択した．また RuJIT の実行時間にはベンチマークプログラムの実行時間に加えてトレースの選択，コンパイルの実行時間を含むものとした．

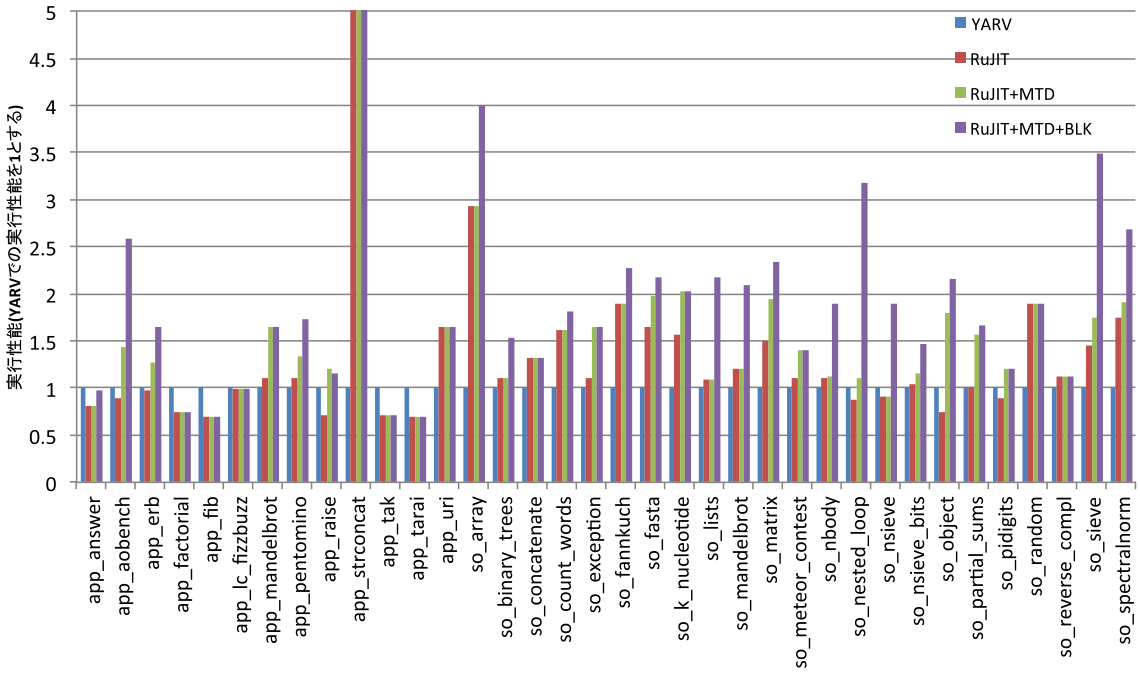

図 8　マイクロベンチマークの実行速度の比較
Fig. 8　Performance comparison between RuJIT and YARV.

5.2　実行時間の比較

図 8 は各ベンチマークプログラムを YARV と，RuJIT で実行した場合の実行速度の比較である．なお，縦軸はそれぞれ YARV で実行した場合（YARV），RuJIT で実行した場合（RuJIT），RuJIT に対し 4.2 節で述べた工夫を適用した場合（MTD），そして MTD に対してさらに 4.1 節で述べた工夫を適用した場合（MTD+BLK）の実行速度を表している．それぞれの値はそれぞれの場合の実行時間を YARV で実行した際の実行時間で割った値の逆数である．またそれぞれの値は YARV の場合の値を 1 として正規化されており，この値が大きいほうが高速であることを示している．

図 8 より，RuJIT は多くの場合で YARV と比べ，最大で 5 倍以上の高速化を達成できたことが確認できる．また 4 章で述べた工夫を適用することで，高速化の実現を確認できる．app_strconcat，so_random のベンチマークは，そのほとんどの処理がトレースとしてコンパイルされており，その結果 2 倍から 5 倍高速化している．

一方で so_reverse_complement や app_lc_fizzbuzz といったベンチマークでは，実行時間の改善はほとんど見られなかった．これはファイル操作やクロージャの実行といった，トレースに含むことができなかった処理を多く含むためと考えられる．なお，app_fibo など，YARV で実行した場合に比べ低速化したベンチマークもある．これらプログラムでは実行のほとんどをインタプリタが占めているベンチマークであった．これらのベンチマークは再帰呼び出しを含むベンチマークであり，執筆時点で RuJIT が再帰呼び出しを含むトレースを機械語に変換できないためである．

最後に，本実験で RuJIT はコード生成器として clang を用いて評価を行った．Clang を用いたトレースのコンパイルは C コードへの変換，Ruby ヘッダファイルの読み込み，ライブラリへのリンク，機械語のロードといったオーバヘッドが生じる．本実験で用いたベンチマークプログラムでは，このオーバヘッドは十分に小さく，平均してプログラム実行時間の 3%程度であった．しかし実行時間の短いプログラムを対象とした場合，コンパイル時間のほうがプログラム実行時間よりも長くなり全体では速度が低下する可能性がある．

5.3　得られたトレースの評価

RuJIT，MTD，MTD+BLK でベンチマークプログラムを実行した際に得られたトレースの平均の長さ，トレースとインタプリタ間の遷移回数を**表 4** に示す．ここで得られたトレースの平均の長さはトレース 1 つを構成する RuJIT の中間表現での命令数の平均値である．

まずトレース区間の最大化を適応することによって，so_nested_loop，so_matrix などのブロックを引数にとる繰返しを扱うメソッドの実行や，C 言語で実装されたメソッド呼び出しを複数含むベンチマークにおいて RuJIT の場合の平均トレース長に比べ MTD での平均トレース長が長くなったことが確認できる．一方で遷移回数については，いくつかのベンチマークで RuJIT のときに比べ遷移回数の増加している場合と減少している場合の 2 つの場合が見られた．これは，トレース区間が長くなることでインタプリタ・トレース間の遷移回数が減少した一方で，新たにコンパイル対象となったコードでさらにインタプリ・トレー

表 4　各ベンチマークにおけるトレースの平均長とトレース・インタプリタ間の遷移回数
Table 4　Trace length and number of transitions between interpreter and traces.

名前	RuJIT トレース長	MTD トレース長	MTD+BLK トレース長	RuJIT 遷移回数	MTD 遷移回数	MTD+BLK 遷移回数
app_answer	0.0	23.0	23.0	0	2	2
app_aobench	0.0	47.6	48.4	0	629,146	201,327
app_erb	0.0	12.0	11.0	0	198	198
app_factorial	0.0	0.0	0.0	0	0	0
app_fibo	0.0	0.0	0.0	0	0	0
app_lc_fizzbuzz	0.0	0.0	0.0	0	0	0
app_mandelbrot	20.0	34.2	35.9	2,209,661	591,884	31,875
app_pentomino	8.0	17.7	18.1	1,833,360	833,360	533,360
app_raise	11.0	11.0	11.0	0	0	0
app_strconcat	14.0	14.0	14.0	2	2	2
app_tak	0.0	0.0	0.0	0	0	0
app_tarai	0.0	0.0	0.0	0	0	0
app_uri	14.3	65.0	70.0	101,016	60,610	59,397
so_array	0.0	37.5	37.5	0	996	996
so_binary_trees	0.0	84.7	84.7	0	14,286,980	14,286,980
so_concatenate	14.0	24.5	24.5	10	10	10
so_count_words	8.0	38.0	38.0	486,381	632,295	632,295
so_exception	38.0	61.0	61.0	249,995	249,995	249,995
so_fannkuch	15.0	26.0	26.0	650,155	845,202	845,202
so_fasta	0.0	41.0	42.2	0	2,032,838	1,580,781
so_k_nucleotide	17.0	25.2	25.3	32	179,151	170,049
so_lists	15.0	31.0	31.0	401	451,873	451,873
so_mandelbrot	0.0	57.5	59.0	0	2,379,060	1,979,060
so_matrix	0.0	21.6	23.1	0	26,403,167	1,240,316
so_meteor_contest	12.0	47.9	51.1	38	9,500,356	8,935,619
so_nbody	34.0	60.8	60.8	0	16,801,052	14,801,052
so_nested_loop	0.0	28.2	28.2	0	36,909,858	5,913,770
so_nsieve	11.0	27.0	27.0	8,257,703	9,601,980	9,601,980
so_nsieve_bits	0.0	37.1	38.5	0	13,453,946	8,745,065
so_object	0.0	29.3	29.8	0	2,980,035	2,961,110
so_partial_sums	0.0	116.0	116.0	0	1	1
so_pidigits	0.0	83.0	81.3	0	910,581	611,763
so_random	43.0	43.0	43.0	1	1	1
so_reverse_complement	5.0	31.3	31.3	940,269	625,007	625,007
so_sieve	0.0	30.5	30.1	0	4,903,982	3,286,996
so_spectralnorm	0.0	79.9	80.1	0	16,977,280	9,908,118

ス間の遷移が発生したためと考えられる．

また，同様に各ベンチマークにおいて，ブロックを考慮したトレースの構成を利用可能な場合，MTD の場合に比べ MTD+BLK の場合のトレース・インタプリタ間の遷移回数が減少していることが確認できる．一方で平均トレース長は大きく増減することは見られなかった．この結果は，ブロック呼び出しを持つループにおけるブロックに対するガード命令の巻き上げによるものだと考えられ，実行速度の向上にも効果があったと考えられる．

6. 関連研究

Dynamo [13] はトレース方式コンパイラを備えた最初の最適化コンパイラである．Dynamo は，機械語を解釈するインタプリタを用いて機械語からトレースを構成する．RuJIT で用いたトレースの選択方式やコンパイルされたトレースの管理は，Dynamo が用いた手法と同等のものを用いて行っている．

近年では，トレース方式 JIT コンパイラは動的型付き言語のコンパイル手法として広く検討が行われてきた．たとえば PyPy [3] は Python 言語用の，SPUR [14]，Trace-

Monkey [4] は JavaScript 言語用，LuaJIT [5] は Lua 言語用のトレース方式 JIT コンパイラとしてあげられる．このようなコンパイラでは，多くの最適化手法により複雑なトレースを最適化している．たとえば PyPy では冗長なガード命令の除去やエスケープ解析，SPUR ではループ展開や不変式の移動を使用している．

Ruby 言語用の JIT コンパイラはメソッド方式を中心に検討されてきた．MacRuby [11] と Rubinius [12] は LLVM コンパイラ基盤 [15] をコード生成器として利用したメソッド方式 JIT コンパイラを持つ．これらの処理系は，Ruby プログラムをそれぞれ独自に実装したバイトコードに変換し，LLVM を用いて機械語に変換する．また，Ishii らは Ruby 処理系を拡張し，独自のコード生成器を持つメソッド方式 JIT コンパイラを提案している [10]．

また，Ruby 言語用トレース方式 JIT コンパイラとしては Topaz がある．Topaz は Python 言語処理系の 1 つである PyPy 上に構築された Ruby 処理系である．RuJIT は，YARV を拡張し実装されており，既存のプログラムや，C 言語で実装された拡張ライブラリを変更せずに利用できる点で異なっている．

最後に，本論文で用いたトレース区間を最大化するための工夫の 1 つであるトレースにネイティブメソッドの呼び出しを含める方法は同様の手法として Inoue ら [16] の手法がある．Inoue らの手法は Java を対象としており，GC 処理や例外の発行，他の Java メソッドの呼び出しを行わない Java の基本ライブラリの JNI メソッドの呼び出しをトレース記録の終了条件とせずトレースに含める方法をとっている．

また，ブロックを引数にとるメソッド呼び出しの最適化として，CastOff [17] ではブロックを呼び出すメソッドおよび起動するブロックを呼び出し元にインライン展開するブロックインライニングを提案している．ただし CastOff ではこのブロックインライニングはインライン展開対象であるメソッドが C 言語で定義された場合に限定しており，また手作業でインライン化を行う処理を実装する必要がある．一方で我々の手法では Ruby で記述されたブロックを引数にとるメソッドを対象としており，また C 言語で記述された Ruby での再実装を行うことで自動的にインライン化が可能である．

7. まとめ

本研究では，Ruby 言語処理系の実行速度改善のため Ruby 向けトレース方式 JIT コンパイラ RuJIT について，その設計と実装方法，性能向上のための工夫，そしてその性能について述べた．

実装では，バイトコードを RuJIT が持つ中間表現に変換した後，最適化を施し高速化したコードの生成を試みた．また，Ruby 言語で多用されるブロックを用いた繰返し文をもとに生成されたトレースは，複数の関数オブジェクトを含む呼び出しの際に side exit を多発させるという問題があった．RuJIT では，コード生成を行う際，トレース生成時に構築した仮定条件の実行時検査の巻き上げを行い，実行時オーバヘッドを削減する工夫を適用し，この問題に取り組んだ．実験結果より数値計算をはじめとした多くのベンチマークにおいて，本研究における JIT コンパイル機構が有効であったことが確認できた．今後の課題として，まずコード生成手法について，とくに小さなプログラムに対する本手法が生成するコードはコンパイル時間，コードサイズの面でコストが高く，改善の余地が見られる．また，もう 1 つの課題として CRuby との非互換性がある．CRuby で提供されている YARV 内部で発生したイベント（たとえば分岐命令の実行や例外の発生，メソッドの call や return など）を扱うトレース機能（TracePoint）に対応していないことがあげられる．そのため Ruby プログラムのデバッガ，プロファイラなどいくつかのアプリケーションが動作しないといった問題がある．RuJIT では，TracePoint の機能と同等の機能の提供などを行うことでこれらの課題を解決し，Ruby 処理系としての完成度を高めていきたい．

参考文献

[1] Matsumoto, Y.: Ruby Programming Language, available from ⟨https://www.ruby-lang.org/⟩.

[2] Paleczny, M., Vick, C. and Click, C.: The Java HotSpot™ Server Compiler, *JVM'01: Proc. 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*, Berkeley, CA, USA, USENIX Association, pp.1–1 (2001).

[3] Bolz, C.F., Cuni, A., Fijalkowski, M. and Rigo, A.: Tracing the meta-level: PyPy's tracing JIT compiler, *ICOOOLPS '09: Proc. 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, New York, NY, USA, ACM, pp.18–25 (online), DOI: http://doi.acm.org/10.1145/1565824.1565827 (2009).

[4] Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M.R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E.W., Reitmaier, R., Bebenita, M., Chang, M. and Franz, M.: Trace-based just-in-time type specialization for dynamic languages, *Proc. 2009 ACM SIGPLAN conference on Programming language design and implementation, PLDI '09*, New York, NY, USA, ACM, pp.465–478 (online), DOI: http://doi.acm.org/10.1145/1542476.1542528 (2009).

[5] Pall, M.: The LuaJIT Project, available from ⟨http://luajit.org/⟩.

[6] Tatsubori, M., Tozawa, A., Suzumura, T., Trent, S. and Onodera, T.: Evaluation of a just-in-time compiler retrofitted for PHP, *Proc. 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, VEE '10*, New York, NY, USA, ACM, pp.121–132 (online), DOI: http://doi.acm.org/10.1145/1735997.1736015 (2010).

[7] Google: V8 JavaScript Engine, available from ⟨http://code.google.com/p/v8/⟩.

[8] Lua projects: The Programming Language Lua,

available from ⟨http://lua.org/⟩.
[9] Sasada, K., Matsumoto, Y., Maeda, A. and Namiki, M.: YARV: Yet Another RubyVM: The Implementation and Evaluation, 情報処理学会論文誌プログラミング，Vol.47, No.2, pp.57–73 (online), available from ⟨http://ci.nii.ac.jp/naid/110004078712/en/⟩ (accessed 2006-02-15).
[10] Ishii, N., Murata, S., Chiba, Y. and Doi, N.: An Implementation of a Dynamic Compiler for Ruby, 情報処理学会論文誌プログラミング，Vol.4, No.1, pp.109–122 (online), available from ⟨http://ci.nii.ac.jp/naid/110008616668/en/⟩ (2011).
[11] Sansonetti, L.: MacRuby, available from ⟨http://macruby.org/⟩.
[12] Phoenix, E.: Rubinius, available from ⟨http://rubini.us/⟩.
[13] Bala, V., Duesterwald, E. and Banerjia, S.: Dynamo: A Transparent Dynamic Optimization System, *SIGPLAN Not.*, Vol.35, No.5, pp.1–12 (online), DOI: 10.1145/358438.349303 (2000).
[14] Bebenita, M., Brandner, F., Fahndrich, M., Logozzo, F., Schulte, W., Tillmann, N. and Venter, H.: SPUR: A trace-based JIT compiler for CIL, *Proc. ACM international conference on Object oriented programming systems languages and applications, OOPSLA '10*, New York, NY, USA, ACM, pp.708–725 (online), DOI: 10.1145/1869459.1869517 (2010).
[15] Lattner, C. and Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, *Proc. 2004 International Symposium on Code Generation and Optimization, CGO '04*, Palo Alto, California (2004).
[16] Inoue, H., Hayashizaki, H., Wu, P. and Nakatani, T.: A Trace-based Java JIT Compiler Retrofitted from a Method-based Compiler, *Proc. 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '11*, Washington, DC, USA, pp.246–256, IEEE Computer Society (2011).
[17] Shiba, S., Sasada, K. and Hiraki, K.: Cast Off: A Compiler for Ruby Implemented as a Library, 情報処理学会論文誌論文誌トランザクション，Vol.2012, No.1, pp.1–22 (2012).

井出 真広　（学生会員）

1988 年生．2012 年横浜国立大学工学府物理情報工学専攻修士課程修了．言語処理系の研究に従事．

倉光 君郎　（正会員）

1972 年生．愛知県出身．2000 年東京大学大学院理学系研究科情報科学専攻博士課程中途退学．同年東京大学大学院情報学環助手．2007 年より横浜国立大学工学部准教授．博士（理学）．平成 20 年山下研究記念賞受賞．ACM，日本ソフトウェア科学会各会員．