情報処理学会論文誌　プログラミング　Vol. 4　No. 1　90–108 (Mar. 2011)

実用的なRuby用AOTコンパイラ

芝　哲史[†1]　笹田耕一[†1]　卜部昌平[†2]
松本行弘[†2]　稲葉真理[†1]　平木敬[†1]

本稿では，Ruby処理系とほぼ完全な互換性を持つAOTコンパイラの設計と実装について述べる．Rubyは数多くのライブラリを持ち，数多くの環境をサポートしているプログラミング言語の1つである．本研究では，Ruby処理系との互換性，および可搬性に優れた手法を用いて，既存のすべてのRubyプログラムを，Rubyがサポートするすべての環境で高速化することを目標としている．我々は，この目標を達成するために，Rubyスクリプトをコンパイルしたバイトコード列をC言語に変換し，Ruby処理系の仮想マシン（RubyVM）上で動作させるAOTコンパイラを開発した．開発したAOTコンパイラは，生成するC言語ソースコードをRubyVMのメソッド呼び出し機構，例外処理機構などを利用して動作させることで，Ruby処理系との互換性をほぼ完全に保ちながら，Rubyプログラムの実行を高速化する．本稿では開発したAOTコンパイラの設計と実装，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，開発したAOTコンパイラの機能と性能を評価する．

A Practical AOT Compiler for Ruby

Satoshi Shiba,[†1] Koichi Sasada,[†1] Shohei Urabe,[†2]
Yukihiro Matsumoto,[†2] Mary Inaba[†1]
and Kei Hiraki[†1]

In this paper, we will describe the design and the implementation of an AOT compiler which is almost compatible with Ruby Interpreter (CRuby). Ruby is one of the programming languages that has a lot of libraries and has been available for many different computing environments. Our research is aiming at speeding up all existing Ruby programs on all Ruby available computing environments with an approach that takes care of compatibility with CRuby and portability for different computing environments. To achieve this goal, we developed an AOT compiler converts the bytecodes that are compiled from Ruby script to C program. The generated C code runs on the RubyVM (Ruby virtual machine). Using some RubyVM mechanism, such as method invocation and exception handling, the AOT compiler speeds up the execution of Ruby program, while makes the generated C code compatible with CRuby. In this paper, we detail the design, the implementation of the AOT compiler and the acknowledge that we had learned from the development. Moreover, we show the evaluation of the functionality and performance for the AOT compiler.

1. はじめに

オブジェクト指向スクリプト言語Rubyは数多くのライブラリを持ち，数多くの環境に対応している世界中で広く利用されているプログラミング言語の1つである[6]．現在でも広く利用されているRuby 1.8までのRuby処理系の評価器は，抽象構文木を再帰的に評価する構造であったため，他の言語処理系と比較して実行速度が低かった．

これに対し，最新版であるRuby 1.9から，高速化のためにYARV（Yet Another Ruby VM）[1]という仮想マシンがRuby処理系に組み込まれた．本稿では以降YARVのことをRubyVMと呼ぶ．RubyVMが組み込まれたことにより，Ruby処理系の実行速度が大きく向上した．しかし，JavaのHotSpotVM[10]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．

より多くのRubyユーザに高速化による利益を得てもらうためには，既存の処理系との互換性，および可搬性は重要である．そこで，本研究では，既存の処理系との互換性，および可搬性に優れた手法を用いて，Ruby処理系がサポートするすべての環境で，既存のすべてのRubyプログラムを高速化することを目標とする．

言語処理系を高速化するための手法として，より抽象度の低い低レベルの言語への変換という手法がある．たとえば，MacRuby[9]やRubinius[8]というRuby処理系では，高速化のためにJIT（Just-In-Time）コンパイルという手法を用いてRubyスクリプトを実行時に機械語に変換する．JITコンパイルは実行時情報を用いた最適化を行うことができるため，Rubyのような動的型付け言語では特に有効である．しかし，JITコンパイラはアーキテクチャごとに異なる機械語を生成する必要があるため，可搬性の面で問題がある．MacRubyやRubiniusはJITコンパイラのバックエンドとしてLLVM[12]というコンパイラ基盤を使用しているため，LLVMが動作しない環境に対応することができない．

†1 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
†2 株式会社ネットワーク応用通信研究所
Network Applied Communication Laboratory, Inc.

そこで我々は，Ruby スクリプトを実行前に C 言語に変換する AOT（Ahead-of-Time）コンパイラの開発を続けてきた[2]．AOT コンパイルは，JIT コンパイルがプログラムの実行時にコンパイルを行う手法であるのに対し，プログラムの実行前にコンパイルを行う手法である[3]．Ruby 処理系のビルドが C コンパイラを用いて行われることから，Ruby 処理系が動作するすべての環境に対して C コンパイラを用いた機械語生成を行うことが可能である．また，Python 用 AOT コンパイラである 211[4] や，PHP 用 AOT コンパイラである PHC[5] では，プログラムを実行前に C 言語ソースコードに変換し，変換した C 言語ソースコードを処理系と協調動作させているため，可搬性だけでなく，高い互換性を持つ．

我々の AOT コンパイラは，これにならい Ruby スクリプトをコンパイルしたバイトコード列を C 言語に変換し，RubyVM 上で動作させる．これにより，RubyVM の持つメソッド呼び出し機構，例外処理機構，組み込みクラスやメソッドなどのすべての機能をそのまま利用することができ，高い互換性を実現できる．

また，バイトコード列から C 言語ソースコードへの変換は，各バイトコードに対応する C 言語ソースコードのテンプレートを用いて行う．このテンプレートは，RubyVM 生成系[1] を用いて自動生成することで，Ruby 処理系のバージョンアップや仕様変更にも正しく追従することができる．

本稿では，この基本方針に基づいて完成度を向上した，より実用的な AOT コンパイラについて述べる．開発した AOT コンパイラは様々な環境で動作し，Ruby 処理系とのほぼ完全な互換性を持つ．具体的には，Ruby 処理系に付属する 7,806 のユニットテストを通過し，1 万行を超える Ruby アプリケーションを動作させることができた．

また，コンパイラドライバを開発し，容易に単体実行可能ファイル，もしくは共有ライブラリの生成を行うことができるようにした．生成した単体実行可能ファイルはコマンドラインから直接実行することができる．また，共有ライブラリは Ruby がライブラリを読み込むときに用いる `require` メソッドを用いて実行することができる．

本稿では，既存の処理系との，ほぼ完全な互換性を持つ実用的な AOT コンパイラの設計と実装，および得られた知見について詳しく解説する．そして，開発した AOT コンパイラの機能と性能を，Ruby 処理系に付属するベンチマークやテスト，他の Ruby 処理系との比較を用いて評価する．

本稿の構成を以下に示す．まず，2 章で RubyVM の構造を示し，我々の AOT コンパイラの設計と実装を理解するうえで必要となる箇所について解説する．次に 3 章で開発した AOT コンパイラの全体像を述べる．そして，Ruby スクリプトから C 言語ソースコードを生成する C ソースコード生成器について 4 章で設計，5 章で実装を詳しく解説する．5 章ではさらに，AOT コンパイラで互換性を達成することができなかった点について，その理由と解決のための考察を示す．6 章で AOT コンパイラの機能と性能を評価し，7 章で関連研究を述べ，8 章でまとめと今後の課題を述べる．

2. RubyVM の構造

我々の AOT コンパイラは，Ruby スクリプトを C 言語に変換し，RubyVM 上で動作させる．このため，我々は AOT コンパイラを RubyVM の構造に合わせて設計した．本章では，RubyVM の構造のうち，AOT コンパイラとのかかわりが特に深い箇所について解説する．

2.1 RubyVM バイトコード列の構造

開発した AOT コンパイラは，RubyVM のバイトコード列に対応する C 言語の関数を生成する．本節では **図 1** の例を用いて，RubyVM のバイトコードが Ruby スクリプトに対してどのような単位で生成されるか，そして，RubyVM のバイトコード列がどのような構造になっているかを解説する．

RubyVM はスタックマシンとして実装されており，メインループ内で RubyVM のプログラムカウンタが指すバイトコードに対応した処理を実行する．RubyVM のバイトコード列は，バイトコードとバイトコードが処理に使用するオペランドによって構成される．

RubyVM の実行するバイトコード列は，Ruby スクリプトのトップレベル，メソッド，クラス定義文，ブロック[⋆1]，`rescue` 節などを区切りに生成される．たとえば図 1 に示すスクリプトからは，(a)，(b)，(c)，(d) の 4 つのバイトコード列が生成される．

図 1 の (a)，(b)，(c)，(d) は，`RubyVM::InstructionSequence#disasm` という Ruby 処理系の組み込みメソッドを用いて Ruby スクリプトをバイトコード列にディスアセンブルした結果である．図 1 の (a)，(b)，(c)，(d) の各行は左から順に，バイトコード列上のインデックス，バイトコード名，バイトコードが処理に用いるオペランドを示す．図 1 では，(a) が図 1 に示すスクリプトのトップレベルに，(b) が `foo` メソッドに，(c) が `times` メソッドが評価するブロックに，(d) が `rescue` 節にそれぞれ対応している．

2.2 RubyVM バイトコードの行う処理

本節では **図 2** に示す `send` バイトコードの実装を例に，RubyVM のバイトコードが行う

⋆1 メソッドに引数として渡すことができる手続き，図 1 では times メソッドに渡された foo (i) が相当する．

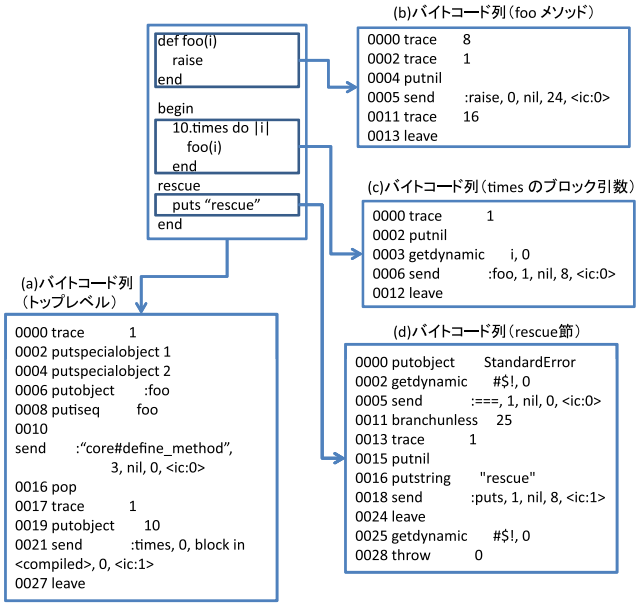


図 1　RubyVM の扱うバイトコード
Fig. 1　RubyVM bytecode.

処理について述べる．

RubyVM のバイトコードは，図 2 に示すように，4 つの処理に分かれている．まず最初に，(1) バイトコード列からのオペランドのフェッチや，RubyVM の値スタック（RubyVM が計算に用いる Ruby オブジェクトのスタック）からの Ruby オブジェクトの取り出しを行い，バイトコードの処理内で用いる変数を初期化する．次に，(2) プログラムカウンタの加算を行う．そして，(3) バイトコードの本質的な処理を行う．たとえば，`send` バイトコードでは，メソッドの起動を行う．最後に，(4) バイトコードが生成した Ruby オブジェクトを RubyVM の値スタックへ積む．

RubyVM のバイトコードが行う処理は，(1) のオペランドフェッチや (2) のプログラムカウンタの加算，(1)，(4) の Ruby オブジェクトの積み込み，取り出しという，(3) のバイトコードの本質的な処理以外のオーバヘッドを含んでいる．

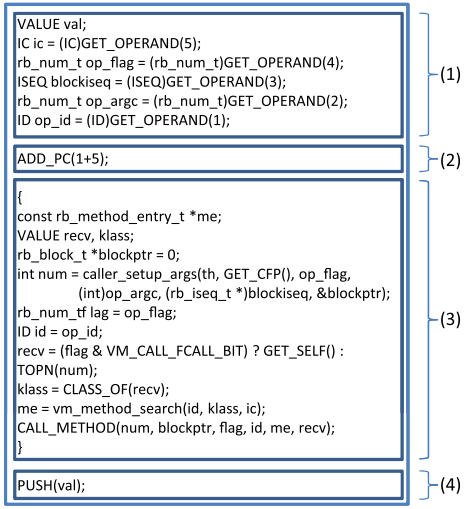


図 2　send バイトコードの実装
Fig. 2　Send bytecode implementation.

2.3　メソッド呼び出し

開発した AOT コンパイラが生成した C 言語ソースコードは，メソッド呼び出しに RubyVM のメソッド呼び出し機構を利用する．RubyVM の扱うメソッドには Ruby で記述されたメソッドと，組み込みライブラリや拡張ライブラリなどの C 言語で記述されたメソッドの 2 種類がある．本稿では Ruby で記述されたメソッドを Ruby メソッド，C 言語で記述されたメソッドを C メソッドと呼ぶ．本節では**図 3** を用いて，Ruby メソッド `r0`，C メソッド `c0` から，Ruby メソッド `r1`，C メソッド `c1` を呼び出す流れを解説する．ブロック呼び出しの流れも，メソッド呼び出しの流れと同様である．

(a) Ruby メソッド `r0` から Ruby メソッド `r1` の呼び出しでは，PC（RubyVM のプログラムカウンタ）を変更し，図 3 中の `VMMain`（RubyVM のメインループ）内で実行するバイトコード列を，`r0` のものから `r1` のものに切り替える．

(b) C メソッド `c0` から Ruby メソッド `r1` の呼び出しでは，まず，例外処理のための準備として，図 3 中の `VMExc`（RubyVM の例外処理用の関数）を呼び出す．そして，`VMMain` を呼び出し，PC に `r1` のバイトコード列を設定する．

(c)，(d) Ruby メソッド `r0`，もしくは C メソッド `c0` から，C メソッド `c1` の呼び出し

は，C 言語の関数呼び出しによって行う．

このように，Ruby メソッドの呼び出しは，`VMMain` 内で PC の指すバイトコード列を切り替えることによって行い，C メソッドの呼び出しは，C 言語の関数呼び出しによって行う．

2.4 例外処理

開発した AOT コンパイラが生成した C 言語ソースコードは，例外処理に RubyVM の例外処理機構を用いる．本節では RubyVM の例外処理機構を解説する．

RubyVM は表引き法と `setjmp` 法を併用して例外処理を実装している[1]．RubyVM は，VM のメインループを実行する前に RubyVM の例外処理用の関数で `setjmp` を呼ぶ．例外発生時には `longjmp` による大域脱出で，RubyVM のメインループや C メソッドから例外処理用の関数へ戻り，例外発生時のプログラムカウンタを基に例外をキャッチするメソッドフレームを検索する．

AOT コンパイラを実装するうえで，`longjmp` による大域脱出を発生させるのは，`throw` という例外を発生させるために定義されたバイトコードや，C メソッドだけではないという点に注意する必要がある．RubyVM では様々なバイトコードの処理内で例外を発生し，`longjmp` による大域脱出が発生する可能性がある．

たとえば図 1 では，`send` バイトコードの処理内で不正な引数が渡されたときや，登録されたハンドラを実行する `trace` 命令によってイベントハンドラが実行されたときに例外が発生し，`longjmp` による大域脱出が発生する可能性がある．RubyVM には，例外が発生しうる関数を呼び出すバイトコードが数多く存在するため，AOT コンパイラはこれらのバイトコードの処理から大域脱出が発生しても正しく動作するようなコードを生成しなければならない．

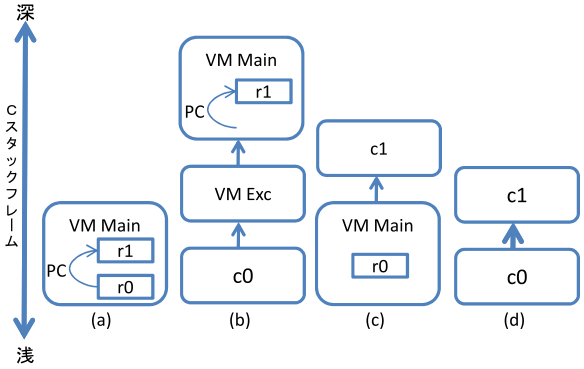


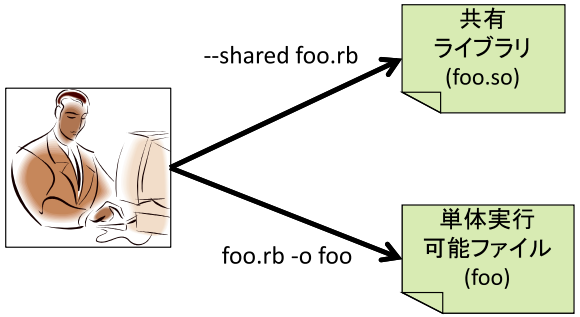
図 3　メソッド呼び出しの流れ
Fig. 3　The scheme of method invocation.

3. AOT コンパイラの全体像

本章では開発した AOT コンパイラの使い方と実行の流れ，および生成した機械語ファイルの実行の流れを解説することで，AOT コンパイラの全体像を示す．

3.1 AOT コンパイラの使い方

我々の AOT コンパイラは，1 つの Ruby スクリプトから 1 つの実行可能な機械語ファイルを生成する．利用者は，**図 4** のようにコマンドラインでコンパイルしたい Ruby スクリ



コマンドラインオプション	解説
--shared	共有ライブラリファイルの出力
-o	出力先の指定
--save-temps	中間ファイル（.c, .o）の保存
-e 'expr'	expr をコンパイル
-v, --metaverbose	コンパイルの途中経過を表示
-h, --help	ヘルプ表示

図 4　AOT コンパイラの使い方
Fig. 4　Usage of the AOT compiler.

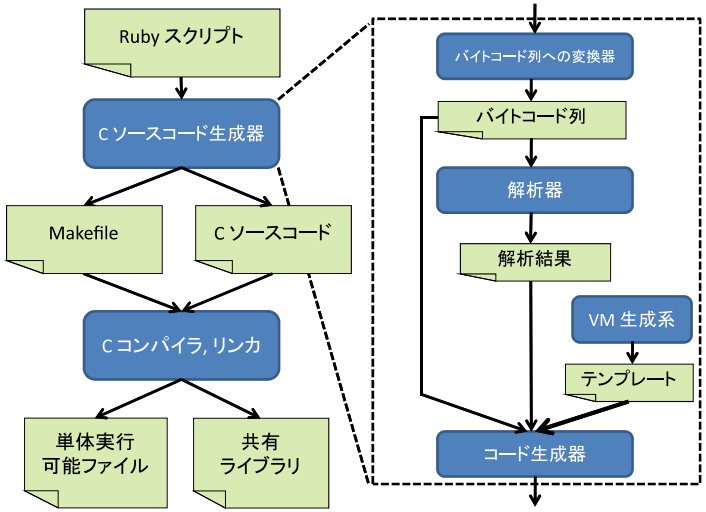


図 5　AOT コンパイラの実行の流れ
Fig. 5　Execution sequence of the AOT compiler.

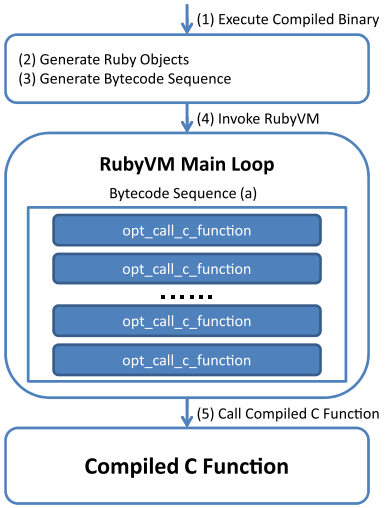


図 6　生成した機械語ファイルの実行の流れ
Fig. 6　Execution sequence of output binary files.

プト，ここでは `foo.rb` を指定して実行可能な機械語ファイルを生成する．生成する機械語ファイルは，単体実行可能ファイル（`foo`），もしくは共有ライブラリ（`foo.so`）のどちらかを指定することができる．

生成した機械語ファイルが単体実行可能ファイルの場合は，通常の Ruby スクリプトをコマンドラインから `ruby foo.rb` と実行するのに対し，`./foo` と実行する．共有ライブラリの場合は，既存の Ruby ライブラリを読み込むのと同様に，`require` メソッドを用いて実行する．

図 4 下部の表に我々の AOT コンパイラのコマンドラインオプションの一部を示す．コマンドラインオプションでは，出力フォーマットの選択や，出力先の指定，ヘルプ表示，中間ファイルの保存などを指定できる．

3.2　AOT コンパイラの実行の流れ

開発した AOT コンパイラの内部的な処理はすべて，AOT コンパイラのコンパイラドライバが管理している．コンパイラドライバは，**図 5** に示す流れに従って，Ruby スクリプトから実行可能な機械語ファイルを生成する．まず，Ruby スクリプトを C ソースコード生成器に通し，対応する C 言語ソースコードを生成する．C ソースコード生成器の設計と実装は 4 章と 5 章で詳しく解説する．次に，C 言語ソースコードをコンパイルするための Makefile を生成し，C コンパイラ，およびリンカを呼び出して指定された種類の機械語ファイルを生成する．このように，図 5 に示す流れはコンパイラドライバが管理しているため，AOT コンパイラ利用者は Ruby スクリプトを入力として AOT コンパイラを起動するだけで，実行可能な機械語ファイルを生成することができる．

単体実行可能ファイルの生成は，Ruby 処理系のビルド時に用いるものと同様の Makefile を生成し，Ruby のランタイムライブラリである libruby と，生成した C 言語ソースコードを C コンパイラでコンパイルしたファイルをリンクすることで行う．

共有ライブラリの生成は，C 言語で記述されたライブラリである拡張ライブラリのビルドに用いる Makefile を生成し，既存の拡張ライブラリと同様にビルドすることで行う．

3.3　生成した機械語ファイルの実行の流れ

本節では **図 6** を用いて，AOT コンパイラが生成した機械語ファイルを実行するときの流れを解説する．Ruby 処理系との互換性を保つためには，生成した C 言語ソースコードはメソッド呼び出し，ライブラリの読み込み，`eval` などの Ruby 処理系が有するすべての機能に対応しなければならない．我々はこれに対応するために，生成した C 言語ソースコード

を RubyVM の機能を用いて動作させる．これにより，RubyVM の持つメソッド呼び出し機構，例外処理機構，組み込みクラスやメソッドをすべてそのまま利用することができる．

C ソースコード生成器はメソッド，ブロックなどを区切りに生成される RubyVM のバイトコード列に対して，C 言語の関数を生成する．生成した関数の実行には RubyVM から C 言語の関数を呼び出すためのバイトコードである `opt_call_c_function` を用いる．

図 6 に AOT コンパイラが生成した機械語ファイルの実行の流れを示す．AOT コンパイラが生成した機械語ファイルは 3.1 節で述べたように，(1) コマンドライン，もしくは `require` メソッドを用いて実行が開始される．実行時には，まず，(2) 変換元のバイトコード列のオペランドなどが使用する Ruby オブジェクトを生成する．そして，(3) `opt_call_c_function` のバイトコード列 (a) を生成する．最後に，(4) RubyVM を起動し，生成したバイトコード列 (a) を実行することで，(5) `opt_call_c_function` バイトコードの実行時に C ソースコード生成器が生成した C 言語の関数が実行される．

4. C ソースコード生成器の設計

本章では，我々の AOT コンパイラの核となる C ソースコード生成器の全体方針について議論し，設計を解説する．

4.1 C ソースコード生成器の全体方針の議論

本節では，C ソースコード生成器の全体方針を議論する．

4.1.1 変換後の C 言語ソースコードの動作手法の議論

C ソースコード生成器の設計において，まず議論すべきなのは変換後の C 言語ソースコードをどのように動作させるかという点である．3.3 節でも述べたように，Ruby 処理系との互換性を保つためには，Ruby スクリプトを変換した C 言語ソースコードは，Ruby 処理系が有するすべての機能に対応する必要があるため，変換後の C 言語ソースコードは RubyVM の機能を用いて動作させる．

単純に変換後の C 言語ソースコードを RubyVM の機能を用いて動作させるだけならば，C 言語ソースコードに Ruby スクリプトを埋め込み，C 言語ソースコードから `eval` メソッドを呼び出せばよい．しかし，`eval` による実行は，通常の Ruby スクリプトの実行と同様の流れで行われるため，本研究の目標である実行速度の向上を達成することができない．

そこで，我々は RubyVM のバイトコード処理の実行サイクルと同等の C 言語の処理をバイトコード列から生成し，動作させる．これにより，RubyVM の持つメソッド呼び出し機構や例外処理機構を用いながら，C コンパイラの最適化や，バイトコードディスパッチの削減による高速化が期待できる．

このような C 言語の処理の生成手法には，各バイトコードごとに，バイトコードの処理を行う関数を作り，バイトコード列をその関数呼び出しの列に変換するという手法[13] と，関数呼び出しではなく，バイトコードの処理そのものの列に変換する手法[4] がある．本稿では前者を関数呼び出し方式，後者を展開方式と呼ぶ．

展開方式は，バイトコードの処理の切替えコストを削減することができるが，バイトコードの処理本体を埋め込むことからコードサイズが大きいため，関数呼び出し方式と比較して，コンパイル時間やバイナリサイズの面で劣る．関数呼び出し方式は，展開方式よりもコンパイル時間やバイナリサイズに優れているが，バイトコードの処理の切替えのコストが関数呼び出しとして残ってしまうため，展開方式ほどの速度向上が期待できない．我々は，本研究の目標である実行速度の向上を達成するために，展開方式を用いる．なお，速度のためには展開方式の方が関数呼び出し方式よりも良いことを確認するために，6.2.1 項では関数呼び出し方式と展開方式をともに評価する．

4.1.2 C 言語ソースコードへの変換の対象の議論

変換後のコードサイズが増すほど，コンパイル時間の増加による AOT コンパイラの利便性の低下や，変換後のバイナリサイズの増加によるディスクスペース使用量の増大，および実行時の消費メモリの増大を招く．このため，バイトコード列から C 言語ソースコードへの変換においては，速度を向上させるだけでなくコードサイズを小さくすることが望ましい．例外ハンドラなどの実行頻度が低いと判断することができるバイトコード列は，変換による速度向上が期待できないため，コードサイズのために変換するべきではない．

これを考慮し，C ソースコード生成器は，バイトコード列の種類やバイトコード列がループを含むかどうかという情報から，実行頻度が低いと判断できるバイトコード列を，C 言語の関数に変換しないことにした．そして，これら以外のバイトコード列はバイトコード列全体を C 言語の関数に変換する．

4.2 C 言語ソースコードへの変換

C ソースコード生成器は，図 5 の右に示す流れで Ruby スクリプトから C 言語ソースコードを生成する．C ソースコード生成器は，まず，Ruby スクリプトをバイトコード列へ変換し，バイトコード列を C 言語ソースコードに変換するかどうかを決定する．そして，変換すると決定したバイトコード列に対して C 言語ソースコードを生成する．本節では C ソースコード生成器が Ruby スクリプトをどのように C 言語ソースコードへ変換するかを解説する．

4.2.1　Ruby スクリプトからバイトコード列への変換

C ソースコード生成器はまず，Ruby スクリプトをバイトコード列に変換する．そして，変換したバイトコード列を用いて CFG（コントロールフローグラフ）を生成し，バイトコード列がループを含んでいるかどうかを解析する．クラス定義文や例外ハンドラ，ループを含まないトップレベルのようなバイトコード列は，実行頻度が低いと判断できるため，C 言語の関数に変換しない．これらのバイトコード列に対しては，バイトコード列の情報を，生成する C 言語ソースコード上に保存する．そして，保存したバイトコード列の情報を用いて RubyVM が扱うバイトコード列のデータ構造を生成する，初期化コードを生成する．なお，ここで解析するのはバイトコード列上のループ構造の有無であり，Ruby で繰返しを記述するのに使用されるイテレータというメソッドの有無は判定していない．C ソースコード生成器はメソッドやブロックをすべて C 言語ソースコードに変換するため，イテレータによる繰返し実行の対象となるブロックはつねに C 言語ソースコードに変換する．

Ruby スクリプトからバイトコード列への変換は，Ruby 処理系に含まれるパーサや，バイトコードへのコンパイラを用いて行う．本稿執筆時の実装では，Ruby スクリプトをバイトコード列に変換する Ruby 処理系の組み込みメソッド `RubyVM::InstructionSequence.to_a` を用いてバイトコード列を取得している．

4.2.2　バイトコード列に対応する C 言語ソースコードの生成

C 言語ソースコードの生成には，RubyVM 生成系[1] から得られる各バイトコードに対応する C 言語ソースコードのテンプレートを用いる．このとき，分岐命令などの一部のバイトコードに対しては，テンプレートを用いるほかに `goto` 文などの C 言語の制御文を使用する．

RubyVM の分岐命令では RubyVM のプログラムカウンタを加減算し，次に処理するバイトコードを制御している．しかし，バイトコード列を変換した関数では，バイトコードの変更ではなく，`goto` 文などを用いた C 言語による分岐の記述が必要になる．

これに対応するために，分岐先にラベルをつけ，**図 7** (a) のような，RubyVM のプログラムカウンタをキーに対応するラベルへの分岐を行う `switch` テーブルを用いる．分岐命令時には，図 7 のように，まず，(1) プログラムカウンタの値を設定し，(2) `switch` テーブルに分岐する．そして，(3)，(4) `switch` を用いて目的のプログラムカウンタに対応するラベルへ分岐する．

しかし，この手法では，(2) `switch` テーブルの先頭への分岐，(3) `switch` テーブル内部の分岐，(4) `switch` テーブルから目的のプログラムカウンタに対応するラベルへの分岐と

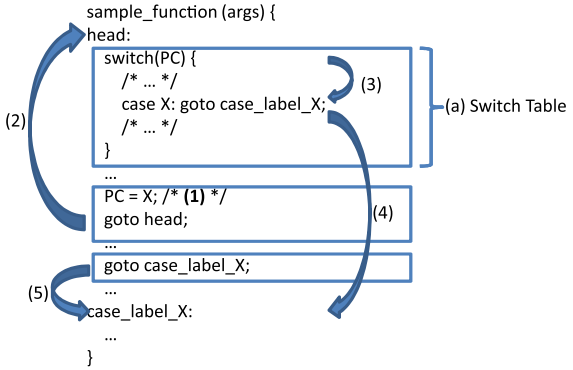


図 7　例外処理と分岐命令への対応
Fig. 7　The method of conducting exception handling and branch instructions.

いう 3 回の分岐が必要となる．そこで，分岐の回数を減らすために，分岐先が単一である場合は，(5) のように，対応するラベルへ直接 `goto` 文を用いた分岐を行う．`switch` テーブルを用いた分岐は，Ruby の `case` 文を処理するためのバイトコードなど，分岐先が複数あるバイトコードの処理を扱う場合に使用する．

4.2.3　バイトコード処理のオーバヘッドの削減

2.2 節で述べたように，RubyVM バイトコードの処理には，プログラムカウンタ操作やオペランドフェッチ，RubyVM の値スタック操作などの RubyVM のバイトコードの本質的な処理以外のオーバヘッドが存在する．本項では，生成する C 言語ソースコードでこれらのオーバヘッドを削減するために行った工夫について解説する．

(1)　プログラムカウンタ操作の削減

プログラムカウンタ操作は，RubyVM が，処理するバイトコードを先に進めるために必要な操作である．これに対し，バイトコード列を変換して得た C 言語の関数では，命令を先に進めるのに RubyVM のプログラムカウンタを操作する必要がない．このため，バイトコード列を変換して得た C 言語の関数では，プログラムカウンタ操作の多くを削除することができる．

しかし，4.2.2 項と 4.4 節で述べるように，C 言語の関数への変換後も一部の処理でプログラムカウンタの値を使用するため，これらが必要とするプログラムカウンタ操作は削除せずに残さなければならない．たとえば，下記の 2 点のバイトコードの処理の実行時には，

```
PUSH(LOCAL_VARIABLE(x));

LOCAL_VARIABLE(y) = POP();
```

図 8　y = x に対応する処理（最適化前）
Fig. 8　Corresponding Code of y = x (Before Optimization).

```
# local は C 言語のローカル変数
local = LOCAL_VARIABLE(x)

LOCAL_VARIABLE(y) = local
```

図 9　y = x に対応する処理（最適化後）
Fig. 9　Corresponding Code of y = x (After Optimization).

C 言語の関数に変換した後でも異なるプログラムカウンタの値をとる必要がある．

- 実行するバイトコード列を異なるバイトコード列に切り替える可能性のあるバイトコード
- 例外を発生しうるバイトコード

前者でプログラムカウンタを異なる値にする理由は，4.4 節で述べる例外処理機構への対応のためである．バイトコード列を切り替えた先で例外がキャッチされた場合は，前者の命令の次の命令から処理を再開するために，プログラムカウンタを使用する．これに関しては 5.1 節で具体例をあげる．

後者でプログラムカウンタを異なる値にする理由は，RubyVM が例外時のバックトレースの取得にプログラムカウンタの値を用いるためである．たとえば，例外を発生しうる `trace` 命令の実行時には，バックトレースを正確に取得するために，プログラムカウンタの値を，他の例外を発生しうる命令の実行時と異なる値にしなければならない．しかし，同一のバックトレースを発生させる命令どうしは，実行時のプログラムカウンタの値を異なる値にする必要はない．たとえば，同一バックトレースを発生させる命令が並んでいた場合は，これらの処理の間のプログラムカウンタ操作は削除することができる．

2.4 節で述べたように，RubyVM では様々なバイトコードの処理で例外が発生する可能性があるため，正しいバックトレースを表示するには，例外を発生しうるバイトコードを見落とさないように注意して実装をする必要がある．

（2）　RubyVM の値スタック操作の変換

RubyVM の値スタックに対する操作，つまりヒープの操作を削減するために，値をやりとりするバイトコードの間に例外処理が発生する可能性のあるバイトコードがない場合は，RubyVM の値スタックに対する操作を C 言語のローカル変数に対する操作に変換する．

たとえば**図 8** に示す，Ruby のローカル変数 `x` の値を Ruby のローカル変数 `y` へ代入する場合は，**図 9** のように，RubyVM の値スタックへの積み込み操作を C 言語のローカル変数への代入に，取り出し操作を C 言語のローカル変数への参照に変換する．RubyVM の値スタックに対する操作はヒープに対する操作であるため，これを C 言語のローカル変数への操作へと変換することにより，Ruby のローカル変数から Ruby のローカル変数への代入などの処理に対して，C コンパイラの最適化を期待することができる．

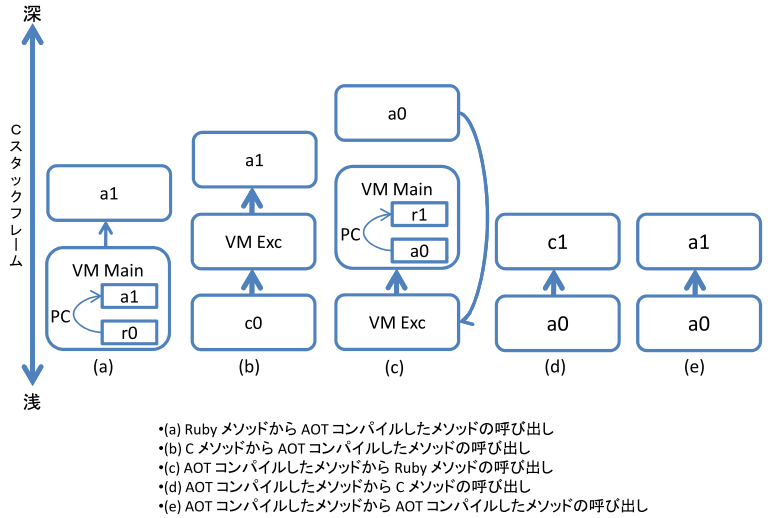


図 10　C ソースコード生成器が生成した関数が関わる，メソッド呼び出しの流れ
Fig. 10　The scheme of method invocation that are related to C code generator.

（3）　オペランドフェッチの高速化

オペランドフェッチを高速化するために，下記に示す変換を適用することで，オペランドフェッチ時のメモリアクセスの回数を削減する．

まず，分岐命令の分岐先や，ローカル変数テーブルのインデックス，数値などの事前に計算することができる種類のオペランドは，オペランドをフェッチする命令を即値に置き換える．

次に，Ruby オブジェクトなどの実行時に計算する必要がある種類のオペランドは，AOT コンパイラが生成した C 言語ソースコード内に static 宣言したグローバル変数に代入する．そして，オペランドをフェッチする命令をグローバル変数への参照に置き換える．

4.3　メソッド呼び出しへの対応

C ソースコード生成器が生成する関数はメソッド呼び出しに RubyVM のメソッド呼び出し機構を利用する．本節では**図 10** を用いて，C ソースコード生成器が生成した関数が関

わる，メソッド呼び出しの流れを解説する．ブロック呼び出しの流れも，メソッド呼び出しの流れと同様である．

(a) Ruby メソッド r0 から C ソースコード生成器が生成した関数 a1 の呼び出しでは，まず，Ruby メソッドから Ruby メソッドを呼び出すときと同様に，RubyVM のプログラムカウンタを変更し，図 10 中の VMMain（RubyVM のメインループ）内で実行するバイトコード列を，a1 を含むものに切り替える．そして，3.3 節で述べたように，opt_call_c_function バイトコードによって，a1 を実行する．

(b) C メソッド c0 から C ソースコード生成器が生成した関数 a1 の呼び出しでは，まず，C メソッドから Ruby メソッドを呼び出すときと同様に，例外処理のための準備として，図 10 中の VMExc（VM の例外処理用の関数）を呼び出す．そして，VMMain と opt_call_c_function バイトコードを経由せずに，a1 を直接呼び出す．

(c) C ソースコード生成器が生成した関数 a0 から Ruby メソッド r1 を呼び出す場合，まず，longjmp による大域脱出を用いて VMMain に戻る．そして，VMMain 内で実行するバイトコード列を r1 のものに切り替える．r1 を実行した後は，4.4 節で述べる例外発生時と同様の流れで，a0 の処理を再開する．

(d) C ソースコード生成器が生成した関数 a0 から C メソッド c1 の呼び出しは，Ruby メソッドから C メソッドを呼び出すときと同様に，C 言語の関数呼び出しによって行う．

(e) C ソースコード生成器が生成した関数 a0 から C ソースコード生成器が生成した関数 a1 の呼び出しは，VMMain に戻り (a) と同様の流れで行うこともできるが，VMMain と行き来するコストを削減するために，VMMain には戻らずに (d) と同様に C 言語の関数呼び出しによって行う．

4.4　例外処理への対応

C ソースコード生成器が生成する関数は，例外処理に RubyVM の例外処理機構を用いる．2.4 節で述べたように，RubyVM の例外処理機構は longjmp による大域脱出を利用するため，C ソースコード生成器が生成する関数の関数フレームは，例外発生時に longjmp による大域脱出で破壊される．たとえば，**図 12** に示す Ruby スクリプトを AOT コンパイルし実行すると，foo メソッドと bar メソッドが定義される．この foo メソッドを実行すると，**図 11** のように，まず，(1) VMMain（RubyVM のメインループ）から foo メソッドに対応する関数を呼び出し，そこからさらに，(2) bar メソッドに対応する関数を呼び出す．このとき，図 12 の bar メソッドは (a) で raise メソッドによって例外を起こし，(3) longjmp による VMExc（VM の例外処理用の関数）への大域脱出を発生させる．これにより，VMExc

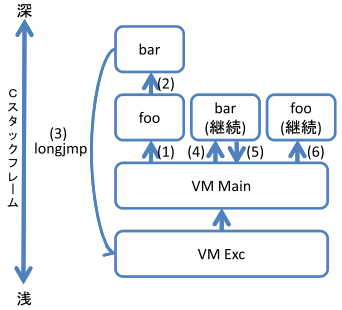


図 11　例外処理の流れ
Fig. 11　The scheme of exception handling.

```
def bar()
  begin
    raise()       #(a)
  rescue
    yyy           #(b)
  end
  zzz             #(c)
end

def foo()
  bar()
  xxx             #(d)
end
```

図 12　例外処理のサンプルコード
Fig. 12　Sample code of exception handling.

の上に積まれている VMMain，foo，bar の関数フレームは破壊される．

ここで注意しなければならないのは，longjmp によって AOT コンパイラが生成した関数，たとえば foo というメソッドをコンパイルして得た関数から抜けた後も，foo メソッドの処理を継続しなければならないという点である．図 12 の場合，例外発生後に (b) の例外ハンドラを実行した後は，bar メソッドの処理を (c) から継続しなければならない．そして，bar メソッドの実行を終えた次は，foo メソッドの処理を (d) から継続しなければならない．

これに対応するためには，C ソースコード生成器が生成する関数が，例外にともなう大域脱出によって関数から抜けた後に，しかるべき位置から処理を継続できればよい[2]．もしくは，例外発生にともなう大域脱出による関数フレームの破壊を防ぐことができればよい．

大域脱出後に処理を継続できるようにする場合，たとえば図 12 だと，(b) の例外ハンドラを実行した後に，まず，(4) VMMain から bar メソッドの関数を呼び出す．そして，bar メソッドの関数内部で (c) に対応する箇所へとジャンプし，bar メソッドの処理を (c) から継続する．同様に，(5) bar メソッドの実行後は VMMain（RubyVM のメインループ）に戻り，(6) VMMain から foo メソッドの関数を呼び出す．そして，foo メソッドの関数内部で (d) に対応する箇所へとジャンプし，foo メソッドの処理を (d) から継続する．

このような方針で大域脱出後に処理を継続することは，4.2.2 項であげた，図 7 のような switch テーブルを用いることで可能である．AOT コンパイラが生成した関数は，大域脱出をともなう可能性のある処理を行う前に，実行コンテキストを RubyVM の管理するスタックに保存し，switch テーブルから再開先に分岐できるようにプログラムカウンタの値を設

定する．たとえば図 12 の foo メソッドの場合は，bar メソッドを呼び出す前に，(d) からの処理の継続に必要な実行コンテキストを，RubyVM の管理するスタックに保存する．そして，switch テーブルから (d) へとジャンプできるように，プログラムカウンタの値を設定する．switch テーブルを用いて (d) に対応する箇所へとジャンプし，RubyVM の管理するスタックに保存した実行コンテキストを用いることで，foo メソッドの処理を (d) から継続することができる．RubyVM の管理するスタックを指すスタックポインタや，プログラムカウンタは，メソッド呼び出し時に RubyVM の管理するメソッドフレーム上に保存されるため，大域脱出が発生した後も参照することができる．

大域脱出による関数フレームの破壊を防ぐことは，setjmp を用いることで可能である．まず，例外をキャッチする処理を変換して得た関数の呼び出し時に setjmp を呼ぶ．そして，longjmp によって setjmp を呼んだ箇所まで大域脱出してきたときに，例外をキャッチできるかどうか調べる．例外をキャッチできるならば例外ハンドラを呼び出し，できないならば longjmp で前の関数フレームへ大域脱出すればよい[3]．

setjmp を用いる手法では，例外をキャッチする可能性のある処理を実行するたびに，setjmp を呼び出して例外に備える必要があるため，setjmp が頻繁に実行され，例外処理に対応するための実行時のオーバヘッドが大きくなる．このため，本 AOT コンパイラでは，setjmp を用いる手法ではなく，switch テーブルを用いる手法で例外処理に対応する．

5. C ソースコード生成器の実装

本章では **図 13** を例に，我々が開発した AOT コンパイラの C ソースコード生成器の実装について解説する．

5.1 生成する C ソースコード

C ソースコード生成器はバイトコード列から C 言語の関数を生成する．たとえば，単に bar というメソッドを呼び出すだけの foo からは，図 13 (c) のような C 言語の関数が生成される．関数 (c) の生成時は，まず，foo メソッドが図 13 (a) のバイトコード列に変換される．このとき，(a) は send 命令で 2 つの区間 (1)，(2) に区切ることができる．1 つ目の区間 (1) は，図 13 (b) の実行時のバイトコード列の 0 番のインデックスに，2 つ目の区間 (2) は，2 番のインデックスに対応している．

このように send 命令で区間を分けるのは，send 命令が実行するバイトコード列を切り替える可能性があるためである．たとえば foo メソッドが send バイトコードの処理によって呼び出す bar メソッド内で例外が発生し，bar メソッドで例外がキャッチされたとする．

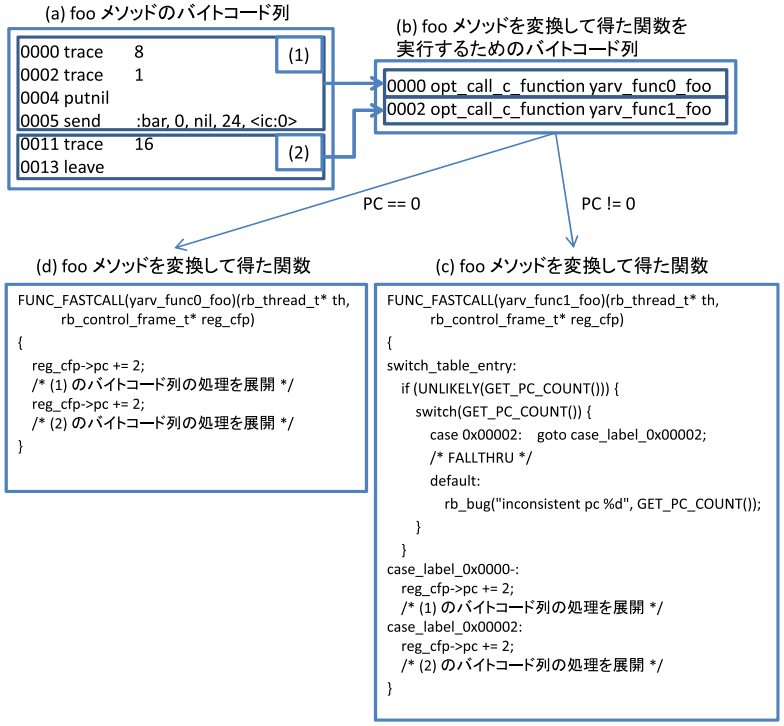


図 13　bar というメソッドを呼び出す foo メソッドを変換した関数
Fig. 13　Compiled code sample.

このとき，bar メソッドから戻った後は foo メソッドの処理を，send 命令の次の命令から再開する必要がある．foo メソッドの C 言語の関数フレームは，例外発生時の longjmp によって破壊されているため，プログラムカウンタと switch テーブルを用いて，send バイトコードの処理の次の trace バイトコードの処理から foo メソッドの処理を再開する．

(c) では，send 命令を実行する前に，プログラムカウンタの加算命令（(c) 中の reg_cfp->pc += 2）によって，プログラムカウンタはバイトコード列上の 2 番のインデックスを指しているため，(c) 先頭の switch テーブルによって，大域脱出後に，区間 (2) の処理から，(c) の実行を再開することができる．

5.1.1　変数名，関数名の生成

高級言語を C 言語ソースコードに変換するとき，C 言語で使用できる変数名の長さ制限や予約語との衝突が問題となる．本稿執筆時の実装では，変数名，関数名には Ruby スクリプト上での名前に対してプレフィックスをつけた名前を利用する．名前が衝突した場合や，長さが一定値を超えた場合は，UUID を用いて生成したユニークな名前を利用する．すべての名前の生成に UUID を用いないのは，AOT コンパイラのデバッグ時の可読性を向上させるためである．

たとえば，図 13 では，`foo` メソッドのバイトコード列から生成した関数に対して，`yarv_func0_foo` という関数名をつけている．`yarv_func0_foo` では，`yarv_func0_` がプレフィックスとなっている．もし，`foo` メソッドが複数定義されていた場合は，関数名が衝突するため，UUID を用いて生成したユニークな関数名がつけられる．

5.1.2　switch テーブルを取り除いた関数の生成

4.4 節で述べたように，C ソースコード生成器が出力する関数，たとえば図 13 の (c) は，大域脱出後に処理を再開するために，再開先の処理のラベルと，再開先へ分岐するための `switch` テーブルを持つ．しかし，このようなラベルと `switch` テーブルは，条件分岐によるオーバヘッドや，C コンパイラの最適化の抑制を招く可能性がある．

そこで，図 13 の関数 (c) に対し，例外からの復帰用の `switch` テーブルとラベルを取り除いた関数 (d) を生成する．そして，関数の実行を先頭から開始するとき，つまり，PC（プログラムカウンタ）が 0 のときは，(c) ではなく (d) を呼び出す．PC が 0 以外で C ソースコード生成器が生成した関数が実行されるのは，主に例外発生時であるため，関数の実行はほぼ (d) で行うことが期待できる．

4.1.2 項で述べたように，変換後のコードサイズは小さいほうが望ましいが，本研究ではコードサイズによるコンパイル時間やバイナリサイズの増加よりも速度向上を優先させ，このような工夫を施した．なお，この工夫の有効性を検証するために，6.2.1 項ではこの工夫を施した場合と施していない場合をともに評価する．

5.2　残る課題

本稿執筆時の実装では，Ruby スクリプトからバイトコード列への変換時に Ruby スクリプトの持つ情報の一部が欠落するという点と，AOT コンパイラが出力した共有ライブラリの名前が衝突するという点，初期化に用いる関数の名前の取り決めにより，C コンパイラによるコンパイルに失敗するという点，現在の Ruby 処理系の仕様により，Ruby 処理系の組み込みメソッドである `load` メソッドから AOT コンパイラが出力した共有ライブラリを読み込むことができないという点が課題として残っており，これらの点のみ互換性がない．本節では，これらの課題とその解決策の検討について述べる．

5.2.1　Ruby スクリプトの持つ情報の欠落

C ソースコード生成器は，バイトコード列の取得に，RubyVM の持つパーサを利用している．RubyVM の持つパーサを使用すると，AOT コンパイラが出力したファイルを実行するときに決定すべき情報を，バイトコード列の生成時に決定されてしまうという問題がある．

たとえば，Ruby には `__FILE__` という擬似変数が存在する．この擬似変数は，実行時に自身が記述されている Ruby スクリプトのパスの文字列オブジェクトへと変換される．ファイルの移動によって `__FILE__` のとるべき値は変化するため，`__FILE__` は実行時に解決しなければならない．しかし，RubyVM の持つパーサは，パース時に `__FILE__` の値を決定し，`__FILE__` を Ruby の文字列オブジェクトに置き換えてしまう．このため，バイトコード列の生成時に `__FILE__` の値が決定されてしまい，AOT コンパイルしたファイルを移動すると，実行時に `__FILE__` がとる値は移動後のパスではなく移動前，つまりコンパイル時のパスになってしまう．

この問題を解決するために，RubyVM のパーサを拡張し，AOT コンパイラが出力したファイルの実行時に解決すべき値を，バイトコード列への変換時には未解決にできるようにし，AOT コンパイラが出力したファイルの実行時に，これらの値を解決できるようにするという対策を検討している．

`__FILE__` とは異なる問題として，`DATA` という組み込み定数の問題がある．Ruby 処理系は `__END__` という文字列のみの行をプログラムの末尾であると解釈する．`__END__` より先の行はすべてプログラムとしては解釈されない．これに対し，Ruby 処理系には，`__END__` より先を指した `File` オブジェクトを示す，`DATA` という組み込み定数が存在する．我々の現在の実装では，バイトコード列の取得時に，元々の Ruby スクリプトのソースコードの情報を失ってしまう．このため，ソースコードの情報を用いる `DATA` を活用するスクリプトが動作しない．

この問題を解決するために，生成した C 言語ソースコードに元々の Ruby スクリプトを埋め込み，埋め込んだスクリプトを Ruby 処理系から参照できるように拡張するという対策を検討している．

5.2.2　拡張ライブラリとのファイル名の衝突

本稿執筆時の実装では，AOT コンパイラが出力した共有ライブラリの読み込みには Ruby

処理系の持つ拡張ライブラリの読み込み機構を利用している．しかし，これには AOT コンパイラが出力する共有ライブラリと，既存の拡張ライブラリとのファイル名の衝突の可能性という問題点が存在する．

拡張ライブラリがサポートするファイルの拡張子は，`.so` や `.bundle` などのように，環境によって固定されている．これらの拡張子を利用しなければ，拡張ライブラリの機構を用いて読み込むことができない．そのため，たとえば `foo.rb` という Ruby スクリプトから `foo.so` という共有ライブラリを生成する場合，すでに `foo.so` という拡張ライブラリが存在したとき，ファイル名が衝突してしまう．このため，拡張ライブラリ `foo.so` の読み込みを意図しているプログラムで，Ruby スクリプト `foo.rb` を AOT コンパイルして得た `foo.so` が読み込まれてしまう可能性がある．

この問題を解決するために，AOT コンパイラが出力する共有ライブラリには，拡張ライブラリがサポートするファイルと衝突しない拡張子をつけることにする．そして，Ruby 処理系の拡張ライブラリ読み込み機構を拡張し，AOT コンパイラが出力する共有ライブラリの拡張子を読み込めるようにするという対策を検討している．

5.2.3　拡張ライブラリのファイル名の制限

Ruby 処理系は共有ライブラリの読み込み時に `Init_XX`（XX は共有ライブラリのファイル名）という関数を実行する．たとえば，`foo.so` という拡張ライブラリを読み込んだ場合，`Init_foo` という関数が呼び出される．拡張ライブラリ `foo.so` はこの `Init_foo` という初期化関数内で，拡張ライブラリの変数の初期化やクラス，メソッドの登録などを行う．

Ruby 処理系は `foo-bar.so` という共有ライブラリを読み込むとき，`Init_foo-bar` という名前の関数を呼び出そうとする．しかし，ハイフンつきの関数名は C 言語では有効ではないため，`foo-bar.rb` という Ruby スクリプトから `foo-bar.so` という共有ライブラリを出力するときに，`Init_foo-bar` という C 言語の関数のコンパイルに失敗してしまう．

この問題を解決するために，5.2.2 項であげた対策に加えて，AOT コンパイラが出力する共有ライブラリを読み込み時の，初期化用関数の命名規則を新しく定義するという対策を検討している．

5.2.4　load メソッドの仕様

Ruby 処理系の組み込みメソッドである `load` メソッドは，Ruby スクリプトのパスを引数として受け取り，実行する．現在の Ruby 処理系の仕様では，`load` メソッドはパスで指定したファイルを Ruby スクリプト（.rb）であると仮定して実行する．このため，共有ライブラリ（.so）を `load` メソッドで実行させても，Ruby スクリプト（.rb）として実行されてしまい，正常実行することができない．

この問題を解決するためには，AOT コンパイラが出力した共有ライブラリを読み込むことができるように，`load` メソッドを拡張する必要がある．しかし，拡張ライブラリと同一の拡張子を `load` メソッドで読み込めるようにしてしまうと，Ruby 処理系との互換性が崩れてしまう．

我々は，この問題に対し，5.2.2 項で述べたように AOT コンパイラが出力する共有ライブラリに独自の拡張子を付加するように変更し，そして，`load` メソッドを AOT コンパイラが出力した共有ライブラリを読み込むことができるよう拡張するという対策を検討している．

6. 評　　価

本章では我々が開発した AOT コンパイラの機能と性能を評価する．評価では**表 1** に示す CPU，OS，コンパイラと**表 2** に示す Ruby 処理系を用いた．表 2 のバージョンの項目は，各処理系を `--version` オプションを指定して起動した結果を示す．本評価を行った時点では，我々の開発した AOT コンパイラは Ruby 処理系のレポジトリの 2010 年 05 月 26 日のリビジョン 28028 をベースに開発しているため，比較評価に用いる Ruby 処理系（表 2 中の CRuby）も同様のリビジョンを使用する．また，本評価では AOTC が我々の AOT コンパイラを示すものとし，C ソースコード生成器の挙動を示すために，**表 3** の用語を使用する．なお，FC では，各バイトコードの処理を行う関数の定義に inline を指定している

表 1　評価環境
Table 1　Evaluation environments.

環境	OS	CPU	メモリ	コンパイラ
Windows	Windows7 32-bit	IntelCore2Duo 2.13 GHz	4 GB	VC2008
Linux32	GNU/Linux 2.6.31 32-bit	IntelCore2Quad 2.66 GHz	4 GB	GCC4.4.1
Linux64	GNU/Linux 2.6.26 64-bit	IntelXeon 2.00 GHz	2 GB	GCC4.3.2
Solaris	OpenSolaris 5.11 64-bit	IntelCore2Quad 2.40 GHz	4 GB	GCC3.4.3
MacOSX	MacOSX 10.5.8 64-bit	IntelCore2Duo 2.33 GHz	3 GB	GCC4.2.1
FreeBSD	FreeBSD 8.0 32-bit	IntelCore2Quad 2.40 GHz	4 GB	GCC4.2.1

表 2　評価に用いた Ruby 処理系
Table 2　Ruby interpreters using experiments.

環境	バージョン，備考
CRuby [6]	ruby 1.9.3dev (2010-05-26) [x86_64-darwin10.0.0] configure では –enable-shared を指定
JRuby [7]	jruby 1.5.1 (ruby 1.8.7 patchlevel 249) (2010-06-24 6586) (Java HotSpot (TM) 64-Bit Server VM 1.6.0_15) [x86_64-java]
Rubinius [8]	rubinius 1.0.1 (1.8.7 2010-06-03 JI) [x86_64-apple-darwin10.0.0] llvm のバージョンは 2.7 を使用
MacRuby [9]	MacRuby version 0.6 (ruby 1.9.0) [universal-darwin10.0, x86_64] llvm のバージョンは 2.7 を使用
MacRubyAOT	上記の MacRuby に付属する AOT コンパイラ

表 3　C ソースコード生成器の挙動
Table 3　Behavior of C code generator.

略称	挙動
ALL	本稿で議論し，設計，実装した機能をすべて有効化
CA	実行時間が低いと推定できるバイトコード列まですべて関数に変換する
SF	5.1 節で述べた，switch テーブルを取り除いた関数の生成を行わない
FC	4.1.1 項で議論した，展開方式ではなく，関数呼び出し方式（inline 指定）を用いる

6.1　機能の評価

本節では開発した AOT コンパイラが十分な互換性を持つことを示すための評価を行う．互換性の評価は，表 1 に示す Linux32 環境で Ruby 処理系の `test` ディレクトリ下の 535 個のテストスクリプトに含まれる，7,850 のユニットテスト，1,849,242 のアサーションを用いて行う．

テストはまず，Ruby 処理系の `test` ディレクトリ下に含まれる `text_XX.rb`（たとえば `test_literal.rb`）という名前のテスト用 Ruby スクリプトを，表 3 の ALL に従う挙動ですべて共有ライブラリにコンパイルし，実行することで行う．また，今回行うテストでは，テストスクリプトが読み込むライブラリは，コンパイルせずに Ruby スクリプトのまま読み込んで実行する．これは，5.2.2 項で述べた拡張ライブラリとのファイル名の衝突の問題のためである．たとえば，Ruby ライブラリ `foo.rb` を共有ライブラリにコンパイルし，拡張ライブラリ `foo.so` を含むディレクトリよりも先にライブラリの検索が行われるディレクトリに `foo.so` を生成したとき，拡張ライブラリ `foo.so` の読み込みは，`foo.rb` をコンパイルして得た共有ライブラリ `foo.so` になってしまい，拡張ライブラリを使用するテストが正常に行えなくなるためである．また，本評価で用いる AOT コンパイラは Ruby 処理系の開発版をベースに開発したため，Ruby 処理系でも失敗するテストについては，本節で行うテストの対象外とする．

テストの結果，ファイル名にハイフンを含む 3 つのファイル `test_open-uri.rb`，`test_xml-stylesheet.rb` と，`test_maker_xml-stylesheet.rb`，700 行超のメソッドを持つ `test_rational2.rb`，`test_complex2.rb` の計 5 つのファイルのコンパイルに失敗し，コンパイルに成功した 530 のテストスクリプトの，7,806 のユニットテスト，1,847,860 のアサーションを通過させた．コンパイルに成功したテストでは，Ruby 処理系でも失敗するテストを除いてすべて通過させることができた．

ファイル名にハイフンを含むファイルのコンパイルに失敗する問題については，すでに 5.2.3 項で述べたとおりである．ファイル名のハイフンの部分をアンダースコアに変えたところ，ファイル名にハイフンを含む 3 つのテストスクリプトをコンパイルし，テストに通過させることができた．`test_rational2.rb`，`test_complex2.rb` のコンパイルに失敗するのは，これらのスクリプトが持つ 700 行超のメソッドに対する C 言語ソースコードのコンパイルに，C コンパイラがメモリ不足によるエラーを発生させたためである．C ソースコード生成器の挙動を表 3 の FC に変化させたところ，この 2 つのテストスクリプトをコンパイルし，テストを通過させることができた．以上の結果により，我々のコンパイラは 5.2 節ですでにあげた問題点を除いて，Ruby 処理系とのほぼ完全な互換性を持つことが分かった．

6.2　性能の評価

本節では開発した AOT コンパイラの性能を評価する．性能の評価は Ruby 処理系の `benchmark` ディレクトリ下のマイクロベンチマークや，Ruby 処理系の基本的処理のベンチマーク，表 2 に示す Ruby 処理系を用いて行う．本評価では次の 3 点を評価する．

(1)　C ソースコード生成器の挙動の違いによるバイナリサイズと速度差
(2)　実行環境の違いによる速度差
(3)　他の Ruby 処理系との速度差

速度差の比較は，ベンチマークの実行時間の比を用いて行う．本評価では，ベンチマークの実行時間は 3 回実行したうち最速のものを用いた．また，AOT コンパイラを用いたベンチマークの実行は，6.2.2 項で用いる RDoc，および Ruby on Rails [14] を除いて，すべて単体実行可能ファイルで行う．

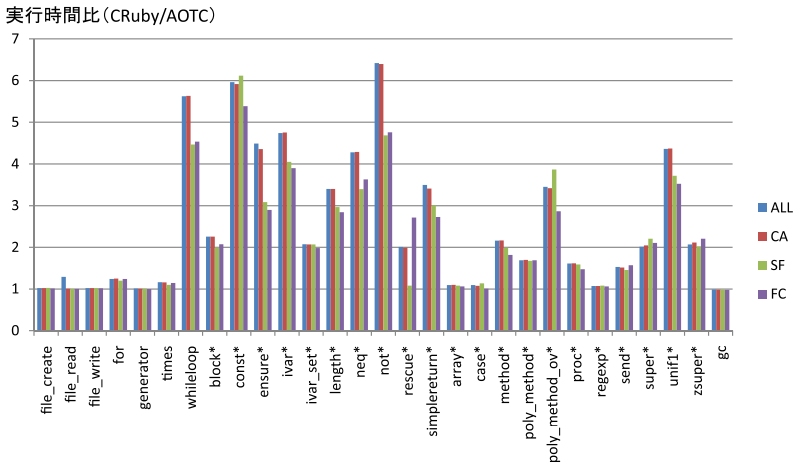


図 14　Ruby 処理系の基本的処理のベンチマークの実行結果
Fig. 14　Benchmark result of Ruby interpreter primitive instructions.

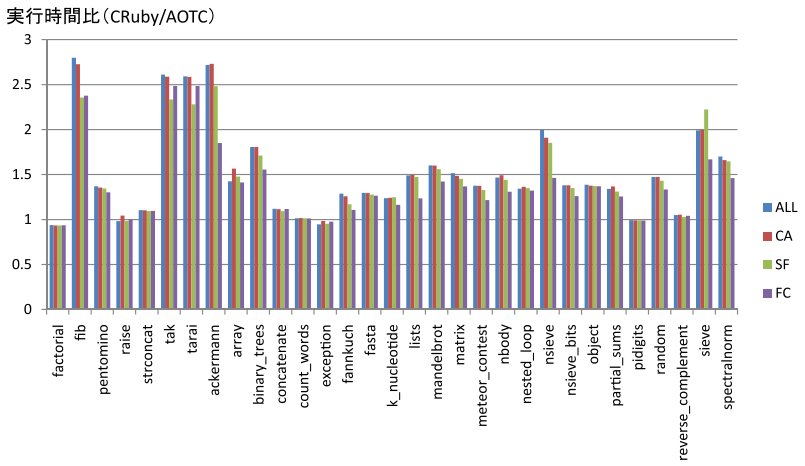


図 15　マイクロベンチマークの実行結果
Fig. 15　Micro benchmark result.

6.2.1　C ソースコード生成器の挙動の違いによるバイナリサイズと速度差

本項では C ソースコード生成器の挙動による速度差を計測するために，表 3 のように C ソースコード生成器の挙動を変化させ，Ruby 処理系の基本的処理のベンチマークや，マイクロベンチマークの実行時間を比較する．

本方式の AOT コンパイラによって速度に影響を受ける処理は主に次の 3 点である．変換の対象である 2.2 節で述べた RubyVM のバイトコードの処理．変換後の関数が関わるメソッド呼び出し，およびブロック呼び出し．GC 時に C のローカル変数から参照される Ruby オブジェクトをマークするために行う，C のスタックフレームの走査（本稿では以降，保守的 GC のコストといった場合，この C スタックフレームの走査を指すものとする）．これらの処理のうち，RubyVM のバイトコードの処理は，4.2.3 項で述べた工夫により，高速化が期待できる．また，Ruby 処理系では組み込みメソッドは Ruby ではなく C 言語で記述されているため，AOT コンパイラの変換対象とはならない．このため，Ruby 処理系が組み込みメソッドとして提供する，配列，文字列や，ファイルに対する基本的な処理は，本方式の AOT コンパイラによって速度に影響を受けない．

Ruby 処理系の基本的処理のベンチマークの実行結果を**図 14** に示す．* 付きの名前の基本的処理のベンチマークは，while ループの中で基本処理を行う形式となっている．しかし，いくつかの基本的処理では，while ループの繰返しのコストの方が高い場合があるため，これらの処理のベンチマークでは while ループの実行時間を無視することができない．このため，* つきの名前のベンチマークでは，実際に実行した時間から while ループのベンチマーク（whileloop）の実行時間を引いた値を用いた．

while ループ（whileloop），Ruby の定数へのアクセス（const），インスタンス変数操作（ivar，iver_set）などの RubyVM のバイトコードの処理が支配的なベンチマークでは，ループは 4.5 倍から 5.5 倍，定数アクセスは 5 倍から 6 倍，変数操作は 2 倍から 5 倍高速化している．一方で，ファイル操作（file_read，file_write，file_create），配列操作（array），GC（gc）などの，RubyVM のバイトコードの処理以外が支配的なベンチマークでは，AOT コンパイラによる速度の向上は見られなかった．

マイクロベンチマークの実行結果を**図 15** に示す．RubyVM では，数値に対する加減乗除の演算子は，特化命令という最適化により，メソッド呼び出しを省略して実行することができる．特に，暗黙の型変換を生じさせないような小さな値を扱う非多倍長整数の演算では，Ruby オブジェクトの割当て処理が行われず，また，演算自体のコストも文字列操作や配列操作などと比較して小さいため，実行時間のほぼすべてが RubyVM のバイトコードの処理に費やされる．

表 4　一部のマイクロベンチマークのバイナリサイズ（バイト）
Table 4　Binary size of compiled micro benchmark (Bytes).

マイクロベンチマーク	ALL	CA	SF	FC
pidigits	411,094	465,886	290,700	276,595
nbody	697,449	757,642	438,088	326,651
object	214,951	237,871	163,246	190,080

このため，フィボナッチ数列（fib）やたらい回し関数（tarai，tak），アッカーマン関数（ackermann）などの単純な整数演算を行うマイクロベンチマークでは，実行時間の多くが RubyVM のバイトコードの処理に費やされており，他のマイクロベンチマークと比較して高速化の割合が大きい．fib が 2.3 倍から 2.8 倍，tak と tarai が 2.4 倍から 2.6 倍，ackermann が 1.4 倍から 2.7 倍高速化している．一方で，文字列の結合（concatenate）や，単語数の計算（count_words）などの実行時間のほぼすべてが文字列処理に費やされるようなマイクロベンチマークでは，AOT コンパイラによる速度の向上は見られなかった．

階乗計算を行うマイクロベンチマーク（factorial）では，AOTC の結果が CRuby よりも 1 割弱遅くなっている．これは，保守的 GC の影響によるものだと考えられる．`factorial` の場合，最も実行に時間がかかっているのは，多倍長整数の計算であるため，C 言語に変換したことによる高速化の利点が小さい．また，Ruby メソッドどうしの呼び出しでは C の関数フレームを新しく積まないのに対し，AOT コンパイルしたメソッドどうしの呼び出しは，C の関数フレームを新しく積むため，保守的 GC のコストが大きい．`factorial` では，再帰呼び出しを繰り返しながら，いくつもの多倍長整数オブジェクトを生成し，GC を発生させるため，C 言語に変換したことによる高速化の利点よりも，保守的 GC のコストが上回り，AOTC の結果が CRuby よりも 1 割弱遅くなっていると考えられる．

クラス定義文を含むマイクロベンチマークである pidigits，nbody，object の，コンパイル後のバイナリサイズを**表 4** に示す．表 4 に含まれるマイクロベンチマークの結果は，図 15 より，ALL と CA で同程度となっていることから，クラス定義文などのバイトコード列を C 言語に変換しないことで，速度を落とさずにバイナリサイズを軽減できることが分かった．

また，SF や FC と，ALL を比較すると，ALL のバイナリサイズが SF や FC と比較して大きい．たとえば，SF，FC の pidigits（円周率を計算するベンチマーク）のバイナリサイズがともに ALL のおよそ 7 割になっている．そして，図 14 や図 15 より，ALL は SF や FC と比較して高速である．たとえば，fib では ALL が CRuby の 2.8 倍高速であるのに対し，SF と FC は 2.4 倍高速となっている．

以上より，速度の面では展開方式のほうが関数呼び出し方式よりも良いことが分かった．また，速度のためには 5.1.2 項で述べた，プログラムカウンタの値による呼び出す関数の変更を，行ったほうがよいことが分かった．しかし，図 14 や図 15 の結果には，ALL と FC，ALL と SF で，速度がほぼ変わらないものも多く見られる．展開方式や 5.1.2 項で述べた工夫がバイナリサイズを増加させることを考慮すると，これらによって高速化が期待できないバイトコード列に対しては，関数呼び出し方式を適用し，5.1.2 項で述べた工夫を適用しないなどの使い分けを行っていくことが望ましいと考えている．

6.2.2　実行環境の違いによる速度差

実行環境の違いによる速度差の評価を，表 1 に示す環境で CRuby と AOTC でマイクロベンチマーク，およびマクロベンチマークを動作させ，実行時間を比較することで行う．AOTC の挙動は，6.2.1 項であげた ALL に従う．

実行するマイクロベンチマークは前項と同様のものを利用する．さらに，現実的なアプリケーションによる評価を行うため，マクロベンチマークとして，RDoc というソースコードからドキュメントを生成する，全 64 個のファイルから構成される 13,875 行の Ruby プログラム，および Ruby on Rails（以降 Rails と略す）という Ruby プログラムを用いる．RDoc は Ruby 処理系に付属する `rdoc` というディレクトリに含まれるソースコードをすべて共有ライブラリに変換し，実行する．Rails は RubyGems [15] で Rails をインストールしたときに生成される `actionmailer`，`actionpack`，`activerecord`，`activeresource`，`activesupport`，`rails`，および `rack` ディレクトリに含まれるソースコードをすべて共有ライブラリに変換し，実行する．RDoc，および Rails が読み込む Ruby スクリプトのうち，上記に列挙したディレクトリ以外に含まれるものは，5.2.2 項で述べた拡張ライブラリとのファイル名の衝突の可能性のため，コンパイルは行わない．

また，Rails に含まれる，`core_ext.rb` という Ruby スクリプトでは，ディレクトリに含まれる `.rb` という拡張子の Ruby スクリプトを読み込むという処理を行っているため，AOT コンパイラが出力した共有ライブラリではなく，コンパイルしていない Ruby スクリプトが読み込まれてしまう．今回行った評価では，`core_ext.rb` からも AOT コンパイルした共有ライブラリを読み込むように，`core_ext.rb` 内の処理を一部書き換えて実行した．

本評価で用いた RDoc は，Ruby 処理系に付属する，バージョン 2.5.8 を，Rails は RubyGems でバージョン 2.3.8 をインストールし，用いた．RDoc は Ruby 処理系に付属するライブラリを入力として，ドキュメントの生成にかかる時間を計測した．Rails は `webrick` という Ruby 処理系に付属する web サーバを用いてサーバを構築し，サーバを動

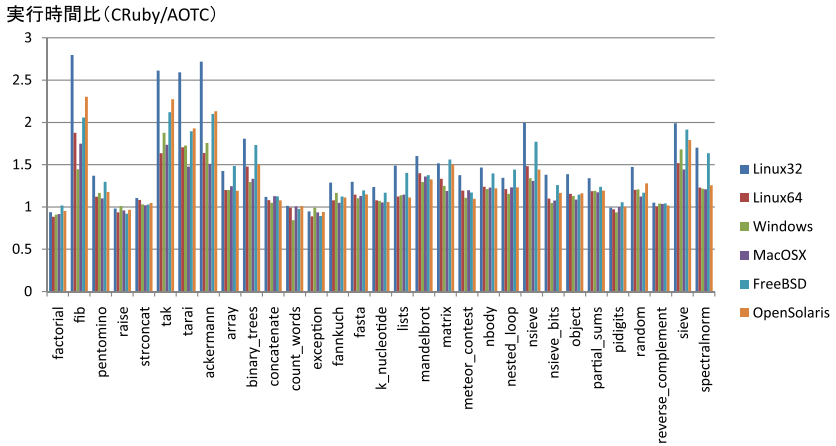


図 16　実行環境ごとのマイクロベンチマークの実行結果
Fig. 16　Micro benchmark result of each execution environments.

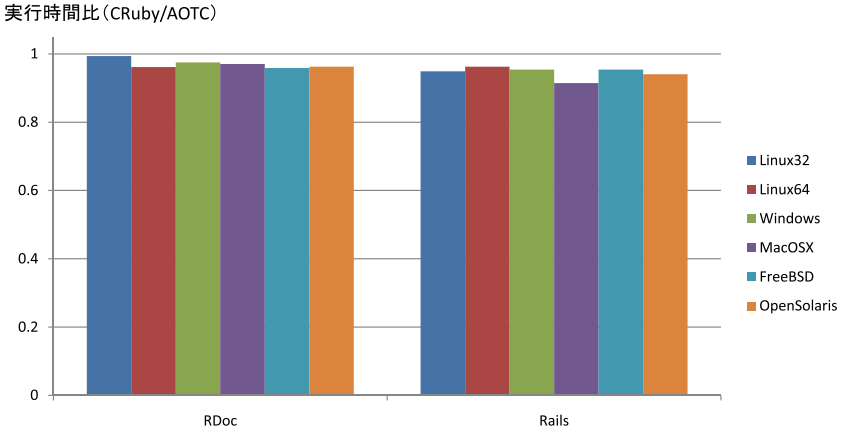


図 17　実行環境ごとのマクロベンチマークの実行結果
Fig. 17　Macro benchmark result of each execution environments.

作させるマシンと同一のマシンから，トップページへ 1 万回アクセスするためにかかる時間を計測した．

図 16 に示す，マイクロベンチマークの実行結果より，評価を行ったすべての環境で，マイクロベンチマークの実行が全体的に CRuby よりも高速となっているが，環境によって，高速化の度合いが異なることが分かる．すべての環境に共通して fib や tarai，tak，ackermann は他のマイクロベンチマークと比較して高速化の割合は大きく，concatenate や count_words などは CRuby と実行時間がほぼ変わらない．環境によって極端に速度が低下するマイクロベンチマークがないのに対し，fib や tak，tarai などの高速化が顕著なマイクロベンチマークは，環境によって実行時間比が大きく異なることが分かった．

マクロベンチマークの実行結果を**図 17** に示す．図 17 より，RDoc，Rails ともに，すべての環境で実行時間が CRuby よりも 1 割弱長くなった．これは，命令キャッシュミスの増加と，AOT コンパイルしたメソッドから Ruby メソッドの呼び出しコストの増加によるものと考えられる．Linux32 環境でサンプリングプロファイラを用いて命令キャッシュミスの回数を確認したところ，Rdoc では約 46%，Rails では約 5%命令キャッシュミスの回数が増加していた．Rdoc や Rails では，AOT コンパイラによって高速化できる，RubyVM のバイトコードの処理の割合が，約 10%から 20%と小さい．このため，特に RDoc では，命令キャッシュミスの増加による速度低下が，RubyVM のバイトコードの処理の速度向上を上回ってしまったと考えられる．また，RDoc や Rails は，今回の評価ではコンパイルしなかった Ruby ライブラリを用いて動作しているため，実行時に AOT コンパイルしたメソッドから Ruby メソッドの呼び出しが，繰り返し発生する．4.3 節で述べたように，AOT コンパイルしたメソッドから Ruby メソッドの呼び出しには例外処理と同様の処理を行うため，メソッド呼び出しのコストが増加する．これらの理由により，実行速度が CRuby よりも低下したと考えられる．

なお，本項で行った評価は性能の評価としてだけでなく，AOTC が様々な OS，コンパイラ，プロセッサアーキテクチャで Ruby プログラムを実行できることを確認するための，可搬性の有無に関する評価とも見ることができる．評価の結果，図 17 に示すように，今回用いた 5 種類の OS と各 OS で利用できるコンパイラ，2 種類のプロセッサアーキテクチャ上で，RDoc，および Rails のような現実的な Ruby プログラムをコンパイルし，実行できたことから，可搬性の高さを確認することができた．

6.2.3　他の Ruby 処理系との比較評価

本項では，我々の AOT コンパイラの，CRuby 以外の Ruby 処理系もあわせたなかでの位置づけを示す参考とするため，他の Ruby 処理系との実行速度の比較を行う．他の Ruby 処理系との比較評価は，MacOSX 環境で表 2 に示す Ruby 処理系，および AOTC を用い

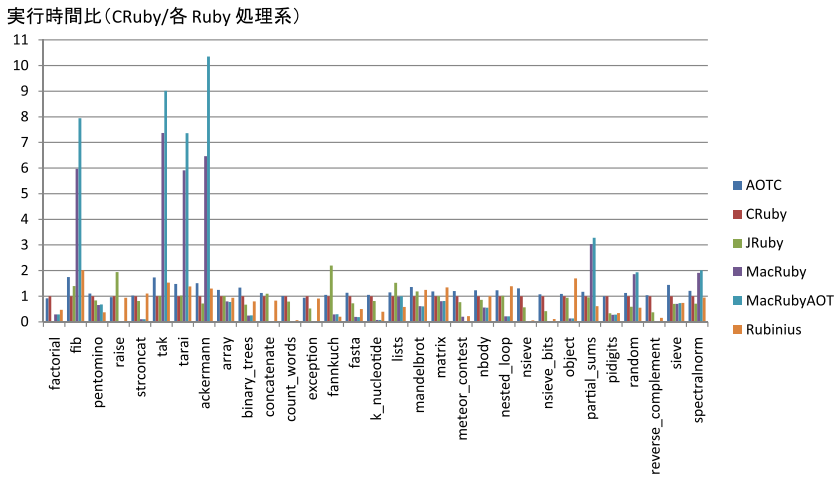


図 18　他の処理系とのマイクロベンチマークの実行結果
Fig. 18　Micro benchmark result of each Ruby interpreters.

てマイクロベンチマークを動作させ，実行時間を比較することで行う．AOTC の挙動は，6.2.1 項であげた ALL に従う．また，スタックオーバフローなどのエラーを理由に動作しなかったマイクロベンチマークは実行時間を無限大として比較した．

図 18 にマイクロベンチマークの実行結果を示す．図 18 より，各処理系ごとに高速に動作するマイクロベンチマークが異なることが分かる．AOTC，MacRuby，MacRubyAOT，Rubinius ともに，fib や tarai，tak，ackermann が，CRuby と比較して高速化の割合が大きい．

特に MacRuby，MacRubyAOT はこれら 4 つのマイクロベンチマークで，CRuby の 6 倍から 10 倍高速であった．しかし，いくつかのマイクロベンチマークでは，実行時間が極端に増加する傾向が見られた．この傾向は，MacRuby，MacRubyAOT だけでなく，Rubinius にも見られた．一方で，AOTC，CRuby，JRuby には極端な実行速度の変化は見られなかった．

7. 関 連 研 究

プログラムを実行前に C 言語に変換する AOT コンパイラは様々なプログラミング言語に対して実装されている[3)–5),11)]．また，Ruby を対象とする AOT コンパイラもすでに開発されている[9),11)]．

Java を対象とした AOT コンパイラである Toba[3)] や，Python を対象とした 211[4)] は，本研究と同様に，バイトコード列を C 言語ソースコードに変換する．本稿の 4.2.2 項で述べた，分岐命令に対して C 言語の goto 文を用いる手法や，4.2.3 項で紹介した，プログラムカウンタ操作の削減，VM の管理するスタックに対する操作の変換などは，これらの研究ですでに提案されているものであり，本稿ではこれらの手法を Ruby 用 AOT コンパイラに適用する方法について紹介している．

211 や，PHP を対象とした AOT コンパイラである PHC[5)] では，我々と同様に生成した C 言語ソースコードを既存の処理系と協調動作させている．既存の処理系と協調動作するためには，既存の処理系の構造に合わせた設計をする必要があるため，設計や実装は協調動作の対象とする処理系の構造によって異なる．本稿では，Ruby 処理系と協調動作する，AOT コンパイラの設計，実装を紹介した．

Ruby2C[11)] は，Ruby に強い制限を加えたサブセットを定義し，実行前に C 言語ソースコードへ変換する．C 言語に変換することから，Ruby2C は我々と同様に高い可搬性を有している．しかし，Ruby2C は高速化のために，暗黙の型変換などの Ruby の高速化を困難にする性質に強い制限を加えているため，互換性の面で問題がある．これに対し，我々は Ruby 処理系とほぼ完全な互換性を持つ AOT コンパイラを開発した．

MacRuby[9)] は，Ruby 1.9 処理系を基に開発されている Ruby 処理系である．MacRuby は高速化のために，JIT コンパイルや AOT コンパイルを行い，Ruby スクリプトを機械語に変換する．特に AOT コンパイルの場合，Ruby スクリプトをパースして得られる抽象構文木を，LLVM のバイトコードへと変換し，LLVM に付属するコンパイラを用いてアセンブリ言語のソースコードを生成する．そして，アセンブラ，リンカを用いて実行可能な機械語ファイルを生成する．これに対し，本研究で開発した AOT コンパイラは，Ruby スクリプトを RubyVM のバイトコード列へと変換し，各バイトコードに対応するテンプレートを基に C 言語ソースコードを生成する．そして，C コンパイラを用いて，実行可能な機械語ファイルを生成する．MacRuby の AOT コンパイラと，本研究で開発した AOT コンパイラは，主に可搬性の面で異なる．MacRuby は，MacOSX 環境のみをサポートしており，Objective-C などの，限られた環境でのみ使用できる機能を用いて実装されている．これに対し，本研究で開発した AOT コンパイラは，Ruby 処理系がサポートするすべての環境に対応するために，MacRuby のように限られた環境でのみ使用できる機能は用いていない．また，6.2.3 項で，MacRuby の AOT コンパイラとの性能比較を行った結果，フィボナッチ

数列の計算などの，単純な整数演算を行う小さな Ruby スクリプトの実行時間は MacRuby の AOT コンパイラの方が高速だが，MacRuby の AOT コンパイラは Ruby スクリプトによって実行時間に偏りがあり，本研究で開発する AOT コンパイラの方が高速に実行できるベンチマークも多く存在した．

8. おわりに

本稿では，Ruby 処理系との，ほぼ完全な互換性を持つ AOT（Ahead-of-Time）コンパイラの設計と実装，得られた知見について述べ，AOT コンパイラの機能と性能を評価した．Ruby スクリプトを C 言語に変換し，RubyVM 上で動作させる，我々の AOT コンパイラの全体像を述べた．そして，AOT コンパイラの核となる，C ソースコード生成器の設計を議論し，C 言語ソースコードへの変換方針，高速化のための工夫，RubyVM のメソッド呼び出し機構，例外処理機構への対応について解説した．

Ruby 処理系に付属するテストを用いて互換性を確認し，機能を評価した．テストの結果，5.2 節であげた問題点を除き，Ruby 処理系とのほぼ完全な互換性を保つことが分かった．また，Ruby 処理系に付属するベンチマークや，他の処理系を用いて性能を評価した．評価の結果，多くの環境で Ruby 処理系よりもベンチマークを高速に実行できることが分かった．

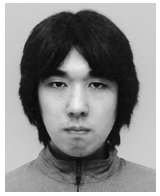
今後の課題として，まず，5.2 節で述べた互換性に関する問題点を解決する．既存の Ruby スクリプトに変更を加えることなく，コンパイル後のファイルを使用することができるように，ユーザにとって使いやすい解決策を模索する．次に，AOT コンパイラによるさらなる高速化を目指す．本稿で紹介した C ソースコード生成器は，Ruby スクリプトから得られるバイトコード列に変更を加えずに，等価な C 言語ソースコードに変換している．そこで，バイトコード列に対する静的解析を行い，最適化を適用したバイトコード列を C 言語ソースコードに変換することで，さらなる高速化を目指す．

謝辞　本研究は科学研究費補助金（課題番号 21220001）の助成を受けたものである．

参 考 文 献

1) 笹田耕一，松本行弘，前田敦司，並木美太郎：Ruby 用仮想マシン YARV の実装と評価，情報処理学会論文誌：プログラミング，Vol.47, No.2, pp.57–73 (2006).
2) 五嶋宏通，笹田耕一，三好健文，稲葉真理，平木　敬：Ruby 用仮想マシンにおける AOT コンパイラ，情報処理学会論文誌：プログラミング，Vol.2, No.1, pp.21–21 (2009).
3) Proebsting, T.A., Townsend, G., Bridges, P., Hartman, J.H., Newsham, T. and Watterson, S.A.: Toba: Java For Applications — A Way Ahead of Time (WAT) Compiler, *Proc. 3rd Conference on Object-Oriented Technologies and Systems* (1997).
4) Aycock, J.: Converting Python Virtual Machine Code to C, *Proc. 7th Intl. Python Conf* (1998).
5) Biggar, P., de Vries, E. and Gregg, D.: A practical solution for scripting language compilers, *SAC '09: Proc. 2009 ACM Symposium on Applied Computing, Programming Languages track* (2009).
6) オブジェクト指向スクリプト言語 Ruby．http://www.ruby-lang.org/ja/
7) JRuby.org. http://jruby.org/
8) Rubinius: Use Ruby. http://rubini.us/
9) MacRuby. http://www.macruby.org/
10) Java SE HotSpot at a Glance. http://java.sun.com/javase/technologies/hotspot/
11) RubyForge: ruby2c — ruby to c translator: Project Info. http://rubyforge.org/projects/ruby2c/
12) Lattner, C. and Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, *Proc. International Symposium on Code Generation and Optimization: Feedback-directed and runtime optimization*, p.75, March 20-24, 2004, Palo Alto, California (2004).
13) Berndl, M., Vitale, B., Zaleski, M. and Brown, A.D.: Context Threading: A Flexible and Efficient Dispatch Technique for Virtual Machine Interpreters, *CGO '05: Proc. International Symposium on Code Generation and Optimization*, pp.15–26 (2005).
14) Ruby on Rails. http://rubyonrails.org/
15) RubyGems.org — your community gem host. http://rubygems.org/

(平成 22 年 7 月 8 日受付)
(平成 22 年 11 月 7 日採録)



芝　　哲史（学生会員）

1985 年生．2009 年千葉大学理学部数学・情報数理学科卒業．現在，東京大学大学院情報理工学系研究科博士前期課程創造情報学専攻在学中．

笹田　耕一（正会員）

2004 年東京農工大学大学院工学研究科博士前期課程情報コミュニケーション工学専攻修了．2006 年同大学院工学教育部博士後期課程電子情報工学専攻退学．博士（情報理工学）（東京大学情報理工学系研究科，2007 年）．2006 年東京大学情報理工学系研究科助手，2008 年同講師（現職）．システムソフトウェア，特に並列処理システム，言語処理系に関する研究に興味を持つ．

卜部　昌平

2008 年電気通信大学大学院電気通信工学研究科博士前期課程情報通信工学専攻修了．修士（工学）．2005 年（株）トランス・ニュー・テクノロジー入社．2009 年（株）ネットワーク応用通信研究所入社（現職）．オープンソースソフトウェアの開発，教育に従事．

松本　行弘（正会員）

1990 年筑波大学第三学群情報学類卒業．同年（株）日本タイムシェア入社．1994 年トヨタケーラム（株）入社．1997 年（株）ネットワーク応用通信研究所入社．2007 年より（株）楽天技術研究所フェローおよび合同会社 Ruby アソシエーション理事長も兼務．プログラミング言語の設計と実装に興味を持つ．ACM 会員．

稲葉　真理（正会員）

東京大学工学部建築学科卒業．武市コンサルティングオフィス，株式会社リコーに勤務した後，東京大学大学院理学系研究科修士課程修了，博士課程中退．理学系研究科助手，講師，情報理工学系研究科特任助教授，准教授．博士（理学）．アルゴリズム，ネットワークの研究に従事．

平木　　敬（正会員）

東京大学理学部物理学科卒業，東京大学大学院理学系研究科物理学専門課程博士課程退学，理学博士．工業技術院電子技術総合研究所，米国 IBM 社 T.J. Watson 研究センターを経て現在東京大学大学院情報理工学系研究科勤務．数式処理計算機 FLATS，データフロースーパーコンピュータ SIGMA-1，大規模共有メモリ計算機 JUMP-1 等多くのコンピュータシステムの研究開発に従事，現在は超高速ネットワークを用いる遠隔データ共有システム Data Reservoir システムの研究，超高速計算システム GRAPE-DR の研究を行っている．