Ruby 向け動的コンパイラの実装
An implementation of a dynamic compiler for Ruby

中央大学大学院　理工学研究科　情報工学専攻
石井　直也
Naoya ISHII

1　はじめに

本論文の目的は動的コンパイルおよび最適化技術が Ruby アプリケーションの実行性能に与える影響を調査した結果を示すことにある．

オブジェクト指向スクリプト言語 Ruby は，プログラミングの手軽さから世界中で利用されている．しかしながら現状では，言語処理系の実装がインタプリタのみを利用する簡易的なものであるため，オーバヘッドが大きく，実行速度が遅い．

インタプリタ実行に伴うオーバヘッドを軽減する手段に動的コンパイラがある．動的コンパイラは Java 向けには実用化されているが Ruby 向けの実装はまだない．そこで我々は Ruby 向けの動的最適化コンパイラを開発した．本論文では動的コンパイラの実装について概観し，また，実装した最適化である脱仮想化について述べる．更に，脱仮想化の実現に必要となる脱最適化の実装についても述べる．

実装したコンパイラが Ruby アプリケーションに与える影響を Ruby 向けベンチマーク集 Ruby Benchmark Suite を使って評価したところ，相乗平均で 20%高速化できることが判った．

本論文の構成は次のとおりである．まず，2 章で，既存の Ruby 実行環境である CRuby について述べ，インタプリタ実行の問題点を示す．3 章では動的コンパイラの実装について述べる．4 章では脱仮想化について述べる．5 章では，Ruby Benchmark Suite を使って，実装したコンパイラの性能を評価する．6 章は結論である．

2　CRuby

本章の目的は松本らによる Ruby の実装である CRuby について述べ，インタプリタ実行の問題点を示すことにある．

CRuby は，ソースコードを受け取るとそれを CRuby のインタプリタが解釈実行可能な中間表現に変換する．本論文ではこの中間表現を VM 命令列と呼ぶ．インタプリタはスタックマシンである．

インタプリタ実行の問題点を次に示す．まず，中間表現を解釈するオーバヘッドがある．インタプリタは命令を実行すると，プログラムカウンタを進め，次の命令を処理するコードにジャンプする．しかしながらジャンプは重い処理であるため，できる限り使用を避けることが望まれる．この問題は，動的コンパイルによって複数の命令をシーケンシャルな機械コード列に変換することで解決できる．次に，スタック操作の効率化が可能である．命令列の中には，命令の実行結果をすぐあとの命令が使用するといったケースが多く存在する．インタプリタはこのような場合，スタックの積み下ろしを連続しておこなうが，この処理は冗長である．この問題に対しては，動的コンパイルによって命令の実行結果の受け渡しをレジスタ経由でおこなうことなどにより高速化が可能である．

3　実装

本章では，動的コンパイラの実装について述べる．まず，3.1 節で基本的な作りと動的コンパイル済みコードの構成について述べるが，CRuby と動的コンパイル済みコードがお互いを呼び出しあうためには呼出し規約などの違いを吸収する必要がある．そこで続く 3.2 節で CRuby と動的コンパイル済みコードの接続に必要となるインタフェース部の作りについて述べる．

3.1 基本的な作り

コンパイラは，VM 命令列のうち呼出し回数が閾値を超えたものを CRuby から受け取ってコンパイルする．コンパイル単位は VM 命令列であるが，コンパイル対象の VM 命令列がブロックや例外ハンドラなど他の VM 命令列を参照する場合には参照先も含めてコンパイルする．コンパイラはまず，VM 命令列を中間表現に変換する．中間表現は VM 命令とほぼ一対一に対応する．続く中間表現から機械コードへの変換は，後述する脱仮想化の適用を除けば，基本的にはテンプレートのつなぎ合わせである．

3.1.1 動的コンパイル済みコードの構成

動的コンパイル済みコードの構成を次に示す．

- エントリ 2
- エントリ
- コード本体
- 例外ハンドラ群
- 例外ハンドラ表
- デバッグ情報

二つあるエントリのうち最初のエントリは CRuby から呼び出される際に用いられるものであり，主にスタックフレームの作成とコード本体へのジャンプをおこなう．もう一方のエントリは，動的コンパイル済みコード間の呼出しの際に利用される．このエントリは，スタックフレームの作成に加え，制御フレームの作成と，呼出し規約にのっとって渡された引数の積替えをおこなう．動的コンパイル済みコード間で利用する呼出し規約を次に示す．

- 引数は左から順に C スタックに積み，最後に引数の個数を積む
- レシーバおよびブロックはレジスタ経由で渡す
- （ブロック起動のとき）ブロックを評価するコンテキストをレジスタ経由で渡す

エントリにおける引数の積替えとは，C スタックに積まれた引数を，CRuby のメソッドローカルフレームのレイアウトに合わせる作業のことである．この作業は本来的に冗長なものであり，動的コンパイル済みコードにおいて独自メソッドローカルフレームを作成す

ることで除去可能となる．しかしながら Ruby では，図 1 のように，メソッドローカルフレームをメソッドの外から参照可能なため，CRuby のレイアウトに合わせることにした．

```
def m
  a = 1 # ローカル変数aに1を代入する
  binding
end
m.eval "a" # => 1
```

図 1　メソッドローカルフレームを外から参照する例

除去可能な積替えによるオーバヘッドを，CRuby をビルドする際におこなわれるライブラリのビルドを対象に計測したところ，積替えを除去可能な引数の個数が呼出し一回につき平均 0.96 個であることから，ロードとストアで高々1.92 メモリアクセス分であることが判った．

コード本体は，前述したようにテンプレートのつなぎ合わせである．テンプレートは，VM 命令におけるスタック操作をシミュレートするが，スタックポインタ操作を不要にするためにオペランドスタックではなく実行スタックを利用する．

例外ハンドラは，例外および `break`, `retry`, `redo`, `next`，`return` の際の着地点であり，実際にそれを捕捉するルーチンに振り分ける．例外ハンドラ表については次項で述べる．

3.1.2 例外ハンドラ表

例外を発生させる場合，その例外を捕捉する例外ハンドラが同一のメソッドに存在するならば，例外は例外ハンドラへのジャンプで実現することができる．しかし，例外を捕捉できる例外ハンドラが存在しない場合，呼出元にさかのぼって例外を処理する必要がある．本システムでは，戻り番地を鍵として例外ハンドラの番地を検索する．ここで検索を実現するために動的コンパイルの際に戻り番地と例外ハンドラの対を表にして持っておく．

3.2 インタフェース部の作り

3.1 節では基本的な作りについて述べたが，CRuby と動的コンパイル済みコードの間には呼出し規約や例外処理機構などの違いが存在することから，お互いを呼び出しあうためにはこの差異を吸収する必要がある．

まず，CRuby から動的コンパイル済みコードを呼び出すケースであるが，本システムではこの呼出しの際にスタブを介することで差異を吸収する．インタプリタから動的コンパイル済みコードを呼び出す際に用いるスタブであることからここでは I2C（Interpreter to Compiled）スタブと呼ぶこととする．

I2C スタブの処理に相当する疑似 C コードを図 2 に示す．

```
 1: i2c_stub(iseq)
 2: {
 3:     fp[-1] = last_compiled_fp;
 4:     ret = iseq->compiled_entry2();
 5: Le:
 6:     last_compiled_fp = fp[-1];
 7:     return ret;
 8: Ee:
 9:     last_compiled_fp = fp[-1];
10:     longjmp(buf, st);
11: }
```

from	to
Le	Ee

図 2　I2C スタブ

3 行目をはじめとする `last_compiled_fp` に関する処理は，後述する脱最適化の際にスタックフレームの走査を可能にするためのものである．続く 4 行目で動的コンパイル済みコードを呼び出す．

`i2c_stub()`は，通常は 7 行目の `return` で返戻するが，例外発生時には，8 行目に遷移する．

9 行目の `longjmp()`はインタプリタに例外を伝播するための処理である．

次に，動的コンパイル済みコードから呼び出すケースであるが，このケースにおける呼出し先には次の三つがある．

(1)　Ruby で定義されたメソッド
(2)　C で定義されたメソッド
(3)　実行時ライブラリ

呼出し先が(1)あるいは(2)の場合には，スタブを利用する．(1)および(2)を呼出し先とする際に使用するスタブを，それぞれ C2I（Compiled to Interpreter）スタブ，C2N（Compiled to Native）スタブと呼ぶことにする．

C2I スタブの処理に相当する疑似 C コードを図 3 に示す．

```
 1: c2i_stub(iseq, ...)
 2: {
 3:     last_compiled_fp = fp;
 4:     rb_vm_set_finish_env(th);
 5:     引数の積替え
 6:     制御フレームを作成する
 7:     if (setjmp(buf) == 0) {
 8:         return vm_exec(th);
 9:     }
10:     else {
11:         lookUpTableAndReturn();
12:     }
13: }
```

図 3　C2I スタブ

4 行目の `rb_vm_set_finish_env()`は，制御スタックフレームにインタプリタの終端を表す番兵を積む

ための実行時ライブラリである．7 行目の setjmp()は，インタプリタが投げる例外を捕捉するためのものである．初回は必ず 8 行目に遷移するが，インタプリタが longjmp()を実施した場合に限り setjmp()が 0 でない値を返戻し 11 行目に遷移する．11 行目の lookUpTableAndReturn()は表引きと返戻を同時におこなう関数である．

C2N スタブの処理は，番兵を積まない，引数の積替え先が C スタックである，呼び出す対象がインタプリタでなく C メソッドであるなどの違いはあるが，大まかには C2I スタブと同一である．

最後に，(3)について述べる．実行時ライブラリを呼び出す疑似 C コードを図 4 に示す．

```
1: if (setjmp(buf) == 0) {
2:     ret = runtime_library(...);
3: }
4: else {
5:     goto exception_handler;
6: }
```

図 4　実行時ライブラリの呼出し

setjmp()は，実行時ライブラリが投げる例外を捕捉するために必要となる．実行時ライブラリは C で記述されているため，CRuby のコンパイルに利用するコンパイラの呼出し規約にのっとって関数呼出しを実施すればよい．

4　脱仮想化

本章では，実装した最適化である脱仮想化について述べる．

Ruby アプリケーションの実行中にはメソッド呼出しが頻発するため，メソッド呼出しの最適化は必須である．しかし，Ruby におけるメソッド呼出しの最適化は簡単ではない．

Ruby のメソッド呼出しにおいて呼出し先はレシーバのクラスおよびそのスーパクラスのメソッドテーブルを検索することで特定できる．しかしながら Ruby は，型を宣言しない動的型言語であることから宣言型を使った型推論ができない．また，Ruby ではメソッドを動的に定義するため，レシーバのクラスを特定できても，メソッドテーブルから得ることができるメソッドが動的に変化しうる．

そこで本システムでは，容易にレシーバのクラスを推定可能で，呼出し先が唯一に定まるケースにだけ最適化を適用し，メソッドのオーバライドなどで呼出し先が変わりうる場合には適用した最適化を解除することにした．

4.1 self をレシーバとするメソッド呼出し

レシーバのクラスを容易に推定可能なメソッド呼出しに self をレシーバとするものがある．self とは，メソッドのレシーバをメソッドの内部から参照するための Ruby のキーワードである．Ruby においてレシーバを記述していないメソッド呼出しは暗黙に self をレシーバとする．

self は，メソッドを定義したクラスと is-a 関係にあるクラスのインスタンスである．なぜなら，特定のメソッドを呼び出すには，そのメソッドを定義したクラスと is-a 関係にあるクラスのインスタンスをレシーバにしなければならないからである．

```
 1: class C
 2:   def f() ... end
 3:   def g() ... end
 4:   def m()
 5:     f()
 6:     g()
 7:   end
 8: end
 9: class CC < C
10:   def g() ... end
11: end
12: o1 = C.new()
13: o1.m()
14: class CC
15:   def f() ... End
16: end
17: o2 = CC.new()
18: o2.m()
```

図 5　オーバライドが最適化を妨げる例

例えば図 5 のプログラムについて考える．図 5 のプログラムには 1～8 行目にクラス C の定義があり，クラス C の中にはメソッド f()，g()，m()の定義がある．続く 9～11 行目にはクラス CC の定義がある．クラス CC はクラス C を継承し，メソッド g()を再定義する．12 行目でクラス C のインスタンスが生成され，13 行目でメソッド m()が呼び出される．ここでメソッド m()を動的コンパイルすると，5 行目のメソッド呼出しはレシーバが self であり呼出し元がクラス C によって定義されているためレシーバのクラスを C と is-a 関係にあるクラスと推定できる．ここでメソッド f()はクラス C によって定義されていて子孫クラスには定義が存在しないことから，5 行目のメソッド呼出しは 2 行目のメソッド f()に対する直接呼出しに変換可能である．この変換を脱仮想化という．一方で 6 行目のメソッド呼出しは，5 行目のメソッド呼出しと同様にレシーバのクラスを推定可能であるが，呼出し元のメソッド m()のレシーバがクラス C のインスタンスであるケースとクラス CC のインスタンスであるケースで呼出し先が異なることから直接呼出しに変換することはできない．

4.2 脱最適化

図 5 の 13 行目において，4 行目のメソッド m()を動的コンパイルし，5 行目のメソッド呼出しを 2 行目のメソッド f()への直接呼出しに変換したとする．この動的コンパイル済みコードは 11 行目まで正しく機能する．しかしながら 12 行目でメソッド f()をオーバライドすると，5 行目のメソッド呼出しに適用した脱仮想化を無

効化する必要が生じる．なぜならオーバライドの結果，5 行目のメソッド呼出しの呼出先が単一でなくなったからである．

適用した脱仮想化を無効化する手順は次に示すとおりである．

① メソッド構造体からコンパイル済みコードへの参照を切る．これによって以降のメソッド呼出しに当該コンパイル済みコードを利用されることがなくなる．

② 実行スタックを走査して当該コードへの戻り番地を脱最適化ハンドラに差し替える．同時に差し替えた箇所数をカウントし，この箇所数（当該コンパイル済みコードに対応するスタックフレームの数）をコンパイル済みコードの破棄カウンタ値として記録する．

脱最適化ハンドラはまず，フレームをインタプリタフレームに変換する．次に，破棄カウンタをカウントダウンする．最後に，インタプリタを呼び出す．ここでフレームの変換を実現するために動的コンパイルの際に，実行時ライブラリの呼出しやメソッド呼出しなど脱最適化のきっかけとなる処理の番地と，その番地において引継ぎをおこなう際に必要な情報の対を表に記録しておく．本システムで必要な情報は，オペランドスタックの深さと，オペランドスタックの各スロットと C スタックおよびレジスタの対応である．

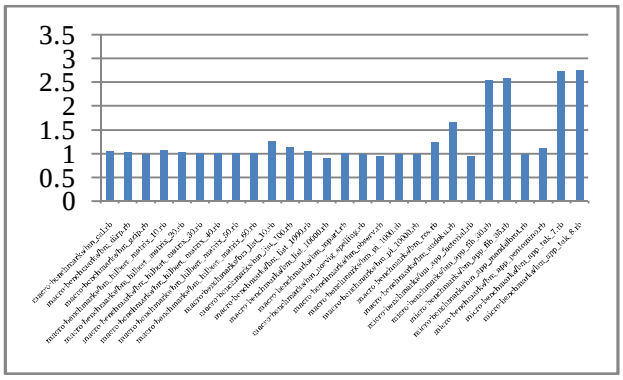
5 評価

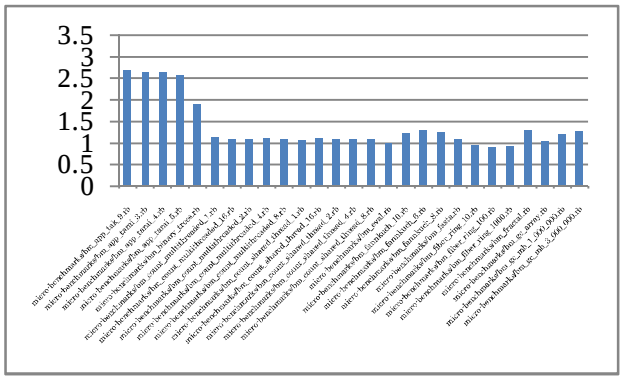
実装したコンパイラが Ruby アプリケーションに与える影響を Ruby 向けベンチマーク集 Ruby Benchmark Suite を使って評価した．評価環境を表 1 に示す．

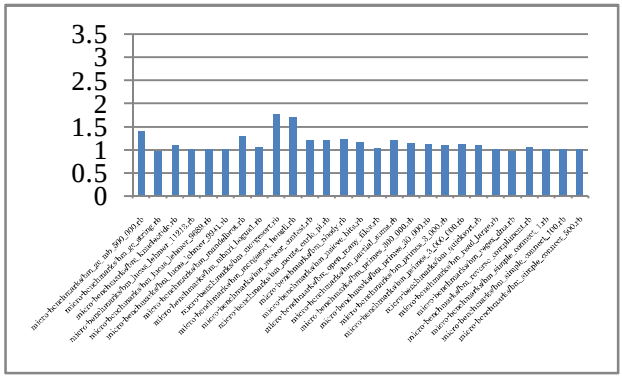
表 1 評価環境

プロセッサ	Xeon3060 2.40GHz
メモリ	2GByte
OS	Ubuntu 9.10

評価結果を図 6 に示す．図 6 の縦軸は本システムによる実行速度の向上率を表す．







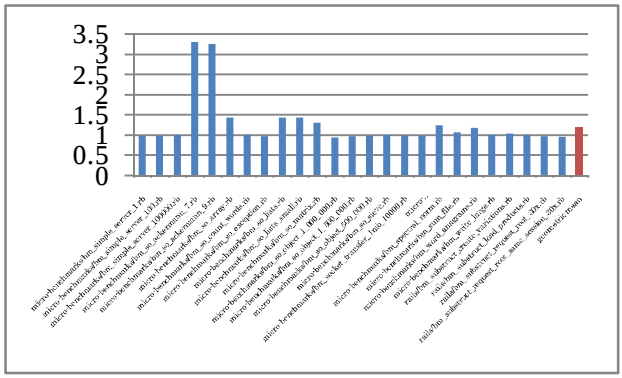


図 6 評価結果

6 結論

動的コンパイラの実装について概観し，また，実装した動的最適化であるメソッド呼出しの最適化について述べた．更に，動的最適化の実現に必要となる脱最適化の実装についても述べた．実装したコンパイラが Ruby アプリケーションに与える影響を評価したところ，相乗平均で 20%高速化できることが判った．

参考文献

[1] オブジェクト指向スクリプト言語 Ruby, http:..www.ruby-lang.org/ja/

[2] 今城哲司，布広永示，岩澤京子，千葉雄司，「コンパイラとバーチャルマシン」，オーム社，2004