情報処理学会論文誌　プログラミング　Vol. 4　No. 1　109–122 (Mar. 2011)

Ruby 向け動的コンパイラの実装

石井直也[†1]　村田俊哉[†1]
千葉雄司[†1]　土居範久[†1]

オブジェクト指向プログラミング言語 Ruby で記述したアプリケーションの実行効率の改善を目的として動的コンパイラを開発した．本論文では動的コンパイルや動的コンパイラ上に実装した動的最適化が Ruby アプリケーションの実行効率に与える影響を評価した結果を示す．実装した動的最適化は脱仮想化と，未実行パスのコンパイル対象からの除去である．Ruby Benchmark Suite を使って評価した結果，実装した全最適化を適用して動的コンパイルすると，インタプリタ実行した場合に比べ実行速度を最大 4 倍，相乗平均で 23.8%高速化できることが分かった．

An Implementation of a Dynamic Compiler for Ruby

Naoya Ishii,[†1] Shunya Murata,[†1] Yuji Chiba[†1]
and Norihisa Doi[†1]

In order to improve Ruby application performance, we developed a dynamic compiler for Ruby. This paper describes the implementation of our dynamic compiler. Our compiler optimizes Ruby applications using dynamic optimizations such as devirtualization and eliminating non-executed paths from compilation target. We evaluated dynamic compilation and optimization effect using Ruby Benchmark Suite, and the maximum performance improvement was 4 times and the geometrical mean was 23.8%.

1. はじめに

オブジェクト指向プログラミング言語 Ruby はその生産性の高さが注目を集めているが，他の言語に比べて処理速度が十分でないことがその適用範囲を狭めている．処理速度が問題となる原因は言語処理系の実装にあり，たとえば実装がインタプリタベースであることから，動的コンパイラを利用する言語処理系に比べて解釈実行のオーバヘッドがかかったり，十分に最適化できなかったりしている．

これらの問題の解決を目的として，我々は Ruby 向け動的コンパイラを実装し，また動的コンパイラ上に次の動的最適化を実装することで処理速度の改善を目指した．

脱仮想化　呼び出し先が唯一の仮想呼び出しを直接呼び出しに変換する最適化．呼び出し先のメソッドを静的に検索してしまうことで，動的な検索の手間を省く．

未実行パスのコンパイル対象からの除外　実行プロファイルから未実行と分かるコンパイル対象をコンパイル対象から除外する．結果として，除外した部分から除外しなかった部分への制御の合流がなくなり，冗長性除去などの最適化がかかりやすくなる．さらに，動的コンパイル済みコードのサイズが小さくなる．動的コンパイル済みコードの実行時に除外したパスに制御が到達した場合には，その先の実行をインタプリタに引き継ぐ．

また，これらの実装に必要な脱最適化もあわせて実装した．脱最適化とは適用した最適化を無効にする技術であり，動的最適化がプログラムをその一時的な実行状況に応じた最適化を適用した後で，実行状況が変化して，適用した最適化が不適切になったとき必要になる．

本論文では，まず次章で関連研究を示し，続く 3 章で実装について詳述する．4 章で Ruby Benchmark Suite を使って動的コンパイラや動的最適化が処理速度に与える影響を調査した結果を示す．5 章は結論である．

2. 関連研究

動的コンパイラは Java など他の言語向けにはすでに実用化されており[1)–8)]，我々が実装した動的最適化[5),9)]，すなわち脱仮想化[10)–12)] や未実行パスのコンパイル対象からの除外[6),13)]，それらを支える脱最適化[6),12),14)] も他の言語向けにはすでに実用化されている．

Ruby 向けにも動的コンパイラ Rubinius[15)] がリリースされているが，本論文のように，動的コンパイラや動的最適化の Ruby 向けの実装について詳述し，なおかつ，Ruby 向けの実用的なアプリケーション開発フレームワーク Ruby on Rails[16)] などを利用するベンチマーク Ruby Benchmark Suite[17)] を使ってその効果を評価した論文は過去にない．

3. 実　装

本章では我々が実装した動的コンパイラの実装について詳述する．まず，3.1 節で動的コン

†1 中央大学
Chuo University

パイラの起動から動的コンパイル済みコードの実行に至る処理の流れを概観し，次に 3.2 節で動的コンパイラの構成を示す．3.3 節では動的コンパイル済みコードの構成を示し，続く 3.4 節では例外処理の実装について述べる．3.5 節で脱仮想化について述べ，最後に 3.6 節で脱最適化の実装について詳述する．

3.1　動的コンパイラの起動から動的コンパイル済みコードの実行に至る流れ

我々の動的コンパイラは，仮想機械として Ruby のリファレンス実装である CRuby（version 1.9.1-p0）を利用する．動的コンパイラの実装にあたり，我々は CRuby に次の修正を加えた．

- 起動時に動的コンパイラのスレッドを生成させる．
- インタプリタにコンパイル対象となりうる処理，すなわち Ruby で定義されたメソッドおよびブロック，例外ハンドラ，クラス定義の呼び出し回数をカウントさせ，呼び出し回数が閾値を超えたらコンパイル待行列に登録させる．
- インタプリタがメソッド呼び出しをする際に動的コンパイル済みコードがあるか確認させ，あったら動的コンパイル済みコードを呼び出させる．

動的コンパイラのスレッドはコンパイル待行列を監視して，登録された処理を順次コンパイルする．動的コンパイルを専用のスレッドで実施するのは，アプリケーションのインタラクティブ性を確保するためである．動的コンパイルはアプリケーションスレッド（インタプリタや動的コンパイル済みコードを実行するスレッド．本論文では Ruby スレッドと記述する）でも実施不可能ではないが，コンパイル対象のメソッドが大きい場合などには長い時間がかかりうるので，その間アプリケーションの実行を止めるとインタラクティブ性を確保できなくなってしまう．

3.2　動的コンパイラの構成

動的コンパイラの構成要素を次に示す．

(1)　前処理部
(2)　最適化部
(3)　コード生成部

前処理部が入力として受け取るのは CRuby インタプリタが解釈実行の対象としている命令列であり，出力するのはコンパイラ向けの中間表現である．CRuby は Ruby で記述したソースコードをインタプリタ向けの命令列に変換してから解釈実行するが，この中間表現が前処理部への入力となる．前処理部が出力するコンパイラ向け中間表現はインタプリタ向けの命令とほぼ 1 : 1 に対応する．コード生成部はコンパイラ向け中間表現を受け取ってネイティブコードに変換する．

ソースコードから CRuby の中間表現，動的コンパイラの中間表現を経て機械命令列に変換する過程を**図 1** に示す．図 1 (a) は Ruby で記述したソースコードであり，図 1 (b)〜(d) は図 1 (a) の 2 行目にあるインスタンス変数への代入文に対応する中間表現と IA-32 [18] の機械命令列をニーモニックで表現したものである．図 1 (b) の CRuby の中間表現は，5 行目で局所変数 `localVar` の値を読み出してオペランドスタックに積み（CRuby の中間表現はスタックマシン向けのものなので計算の途中結果をオペランドスタックに積む），6 行目でオペランドスタックに積んでおいた値をインスタンス変数 `@instanceVar` に書き込む．図 1 (c) の動的コンパイラの中間表現は図 1 (b) の CRuby の中間表現と 1 : 1 に対応している．図 1 (d) の機械命令列は 10〜11 行目で局所変数の値をロードし，13〜17 行目でイン

```
1:  def setInstanceVar(localVar)
2:    @instanceVar = localVar;
3:    return (nil);
4:  end
```

(a) ソースコード

```
5:  getlocal localVar
6:  setinstancevariable :@instanceVar
```

(b) ソースコードの 2 行目に対応する CRuby の中間表現

```
7:  GetLocal(2); // 2 は局所変数 localVar を格納するスロット番号
8:  SetInstanceVariable(@instanceVar の ID);
```

(c) ソースコードの 2 行目に対応する動的コンパイラの中間表現

```
9:  // GetLocal(2);
10: movl LOCAL_FRAME_POINTER_OFFSET(%ebx), %ecx
11: movl -8(%ecx), %eax // -8 は局所変数 localVar を格納するスロットのオフセット
12: // SetInstanceVariable(@instanceVariable の ID);
13: movl %eax, 8(%esp)
14: movl $@instanceVar の ID, 4(%esp)
15: movl SELF_OFFSET(%ebx), %ecx
16: movl %ecx, (%esp)
17: call _rb_ivar_set
```

(d) ソースコードの 2 行目に対応する IA-32 の機械命令列のニーモニック表現

図 1　コンパイルの過程
Fig. 1　Compilation process.

スタンス変数に値をストアする．具体的には，10 行目でレジスタ `%ebx` が指示する制御フレームから局所変数を格納する記憶領域へのポインタを読み出す．ここで制御フレームとは，CRuby が呼び出しの処理を実施するたびに制御スタックというヒープ上の記憶領域に確保する固定長のデータ構造であり，呼び出した処理が必ず必要とする情報（たとえば変数 `self` や局所変数を格納する領域へのポインタ）を格納する役割を担う．我々の動的コンパイラが生成するコードもインタプリタと同様に，メソッドの実行に際して制御フレームを積む．我々の動的コンパイラは制御フレームの参照を高速化するために，レジスタ `%ebx` につねに制御フレームを参照させている．10 行目で局所変数を格納する記憶領域へのポインタを読み出したら，続く 11 行目で局所変数 `localVar` の値を取得する．13〜17 行目はインスタンス変数へのストアを行う関数 `rb_ivar_set` を呼び出すための処理で，13〜16 行目で実引数として `self`，インスタンス変数 `@instanceVar` の識別子，ストアする値を C スタックに積んでから 17 行目で関数 `rb_ivar_set` に制御を移す．

コード生成部が個々の中間表現に対応して出力する機械命令列は，オペランドスタック参照向けに次の最適化を適用している点を除けば，個々の命令に対応してインタプリタが実行するコード片とほぼ等しい．

- CRuby インタプリタがオペランドスタックに積むデータ（オペランドおよび局所変数）の保存先を，CRuby インタプリタがヒープ上に確保しているオペランドスタックではなく，C スタックとする．
- オペランドスタックに積んだオペランドの参照をフレームポインタからの定数オフセット参照で実現する．つまり，参照先のスタックスロットのオフセットを動的コンパイル時に計算してしまい，実行時にはオペランドスタックのポインタを上げ下げする操作を省略する．
  局所変数参照については，局所変数の保存先が手続きオブジェクトの生成時にヒープに変更されうることから，フレームポインタからの定数オフセット参照ではなく，制御フレーム中にある局所変数を格納する記憶領域へのポインタからの定数オフセット参照で実現する．定数オフセット参照による実現は，CRuby インタプリタにおけるインデックス参照による実現，すなわち CRuby の中間表現にエンコードしてある定数オフセットをレジスタにロードし，ロードした値をインデックスとして使いながら参照する実現より，定数オフセットをレジスタにロードする手間がないだけ効率的である．
- 基本ブロック内にある連続する 2 つの中間表現間のオペランドの受渡しは，オペランドスタック経由でなく，レジスタ経由で行う[⋆1]．

オペランドの格納先を C スタックにした理由は，我々の動的コンパイラが出力するコードではフレームポインタをつねにレジスタ上に保持しているため，C スタック上のデータならば，フレームポインタからの定数オフセット参照命令 1 つで参照できるからである．CRuby のオペランドスタックも，そのスタックポインタをつねにレジスタに格納しておけば C スタック上のデータと同じく効率的に参照可能になるが，そのためにレジスタを 1 本使うことは合理的でないと判断した．

我々が実現したオペランドスタック参照向けの最適化の効果は 4 章に示すとおり限定的であり，インタプリタが実行するコード片を貼り合わせたようなコードを出力しても，さほど実行性能を改善できないのは Manjunath らがすでに指摘しているとおりである[19]．そこで我々は，さらなる実行速度の向上を目指し，脱仮想化や未実行パスをコンパイル対象から除外する最適化を実装した．我々の実装では，これらの最適化の際に実行プロファイルが必要になるが，実行プロファイルは動的コンパイル済みコードを使って取得することにした．すなわち，コンパイルの開始時に，コンパイル対象の処理の実行プロファイルを取得済みか調査し，なければ最適化を適用せず，代わりに実行プロファイルを取得するための処理を挿入したコードを生成する．このコードは一定回数呼び出された時点で自身を無効化し，コンパイラに再コンパイルを促す．再コンパイルの際には実行プロファイルの取得が済んでいるので，コンパイラは実行プロファイルを使って最適化を適用する．

3.3　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要素は次に示すとおりである．

コード本体　コンパイル対象のインタプリタ向け命令列に対応するコード．

スタブ　コード本体を呼び出すとき，あるいはコード本体からコード本体もしくはインタプリタ，C 言語で記述されたライブラリ（C ライブラリ）を呼び出す際に使うコード片．呼び出し元と呼び出し先における呼び出し規約の差異を吸収し，呼び出しを実行可能にする役割を果たす．

例外エントリと例外エントリ表　表引き法による例外処理の実現のための表およびコード片．これらは例外の発生時に，動的コンパイル済みコード内での例外処理を実現するためのものである．これらの詳細については 3.4 節で述べる．

⋆1 この最適化の適用例は図 1 (d) の中にある．具体的には，11 行目でロードした値を 13 行目の命令に受け渡す際に，オペランドスタックを経由せず，レジスタ `%eax` を使って受け渡している．

デバッグ情報　脱最適化の際に利用するデータ構造．我々の実装では脱最適化の際に動的コンパイル済みコードからインタプリタへ実行を引き継ぐが，引継ぎの際には動的コンパイル済みコードのフレーム群の内容をインタプリタのフレーム群に書き写さなければならず，このとき動的コンパイル済みコードのフレームのどの欄がインタプリタのどのフレームのどの欄に対応するかを示す情報が必要になる．この情報を記録したデータ構造がデバッグ情報であり，インタプリタへ実行を引き継ぎうる次の箇所ごとに作成する．

- メソッド呼び出し
- スタブ呼び出し
- C 言語で実装した関数の呼び出し

3.4 例外処理

本節では動的コンパイル済みコード内での例外処理の実現について述べる．動的コンパイラを実現する際には，動的コンパイル済みコード内での例外処理だけでなく，動的コンパイル済みコードとインタプリタの間で例外を受け渡す処理も実現する必要があるが，その部分の実装については文献 20) に詳述してあるので本論文では言及しない．

動的コンパイル済みコード内での例外処理は表引き法によって実現する．具体的には，動的コンパイルの際に例外エントリ表という表を用意し，その中に，動的コンパイル済みコードの中にある関数もしくはメソッド呼び出しから例外が発生していない状態で返戻する場合の返戻先番地と，その番地に対応する例外エントリのアドレスを登録しておく．ここで例外エントリとは例外が発生した状態で呼び出し先から戻ってきた際に実施すべき処理（rescue 節への移動，もしくは，呼び出し元への返戻）を行うコード片である．そして，実行時に例外が発生して呼び出し元に戻ることになったら，返戻先番地に対応する例外エントリを例外エントリ表から求めてそこにジャンプする．

たとえば我々の動的コンパイラを使って**図 2** (a) のメソッド m1() をコンパイルすると図 2 (b) の例外エントリ表と図 2 (c)，(d) の例外エントリが生成される．図 2 (b) の例外エントリ表は，例外が発生していない場合の返戻先番地が L1 の際，つまりメソッド m2() の呼び出し中に例外が発生して m1() に返戻してくる際には例外ハンドラ 1 にジャンプせよと指示している．また，例外が発生していない場合の返戻先番地が L2 の際，つまり rescue ブロックの呼び出し中に例外が発生して m1() に返戻してくる際には例外ハンドラ 2 にジャンプするよう指示している．ここで rescue ブロックとは rescue 節の中身を保持するブロックであるとする．CRuby では rescue 節の中身を，その外側にあるコード（ここではメソッド m1() の rescue 節以外の部分）から切り離してブロックという独立したコード片

```
def m1()
 begin
  m2();
  # L1:  m2() からの例外が発生していない状態で返戻する場合の返戻先番地
 rescue
  m3();
 end
  # L2:  例外ブロックからの例外が発生していない状態で返戻する場合の返戻先番地
end
```

(a) ソースコード

例外が発生していない状態で返戻する場合の返戻先番地	例外エントリ
L1	例外エントリ 1
L2	例外エントリ 2

(b) 例外エントリ表

呼し出し前に C スタックに配置した実引数を破棄
rescue ブロックに引き渡す実引数（例外）を C スタックに配置
rescue ブロックからの返戻先番地 L2 を C スタックに配置
rescue ブロックにジャンプ

(c) 例外エントリ 1

呼び出し元の処理の例外エントリ表をみて返戻先番地に対応する例外エントリのアドレスを取得
呼び出し前に C スタックに配置した実引数と，C フレームを破棄
例外エントリにジャンプ

(d) 例外エントリ 2

図 2　例外エントリ表と例外エントリ
Fig. 2　An exception entry table and exception entries.

にしているので，rescue ブロックへの制御の移動はインタプリタでも動的コンパイル済みコードでもジャンプでなく呼び出しの処理になる．

例外エントリ 1 は，例外が発生して返戻してきた際に例外を処理する rescue 節がある場合の例外エントリの例である．この場合は resucue ブロックを呼び出すことになる．例外エントリ 1 では図 2 (c) に示したように，まず，m2() の呼び出しの後始末，すなわち m2() の呼び出しに際して積んだ実引数を破棄する．この処理はフレームポインタの指示先から固定オフセットの位置にスタックポインタを移動する処理として実現できる．次に，rescue ブロックに引き渡す実引数（例外への参照）と rescue ブロックから返戻する際の返戻先番

地 L2 を C スタックに配置してから rescue ブロックにジャンプすることで rescue ブロックを呼び出す．なお，返戻先番地を C スタックに配置して渡すのは IA-32 に固有な実装で，別のアーキテクチャではそのアーキテクチャに応じた渡し方をする必要がある．

例外エントリ 2 は，例外が発生して返戻してきた際に例外を処理する rescue 節がない場合の例外エントリの例である．この場合は呼び出し元（ここではメソッド m1() の呼び出し元）の例外エントリに戻ることになる．例外エントリ 2 では図 2 (d) に示したように，まず C フレームから返戻先番地，すなわちメソッド m1() から例外が発生していない状態で呼び出し元に戻る際の返戻先番地を取得する．次に，返戻先番地の処理に対応する例外エントリを，呼び出し元の例外エントリ表から求める．最後に，rescue ブロックの呼び出しの際に C スタックに配置した実引数と m1() の C フレームを破棄してから例外エントリにジャンプする．

3.5 脱仮想化

脱仮想化は仮想呼び出しを直接呼び出しに変換する最適化で，仮想呼び出しの実行サイクル数やコードサイズに少なくない影響を与える．直接呼び出しの実現に必要な命令数は，実引数の用意に必要な命令を除けば，アーキテクチャに依存するものの 1 個もしくは数個で済むのが一般的であるのに対し，仮想呼び出しの実現には多くの命令が必要になる．たとえば我々の動的コンパイラは，仮想呼び出しに脱仮想化を適用できなかった場合，**図 3** の擬似コードに相当するコードを出力して仮想呼び出しを実現するが，図 3 の擬似コードは，サイズも，実行コストも直接呼び出しのコードに劣る．

さて，脱仮想化では呼び出し先を唯一に特定できる仮想呼び出しを最適化するが，ここで問題になるのが仮想呼び出しの呼び出し先を特定する方法である．呼び出し先を特定するためには，レシーバのクラスを推定すればよいが，Ruby は変数の型を宣言しない言語なので，推定の根拠に宣言型を利用することはできない．

```
static Cache cache;
VALUE klass = CLASS_OF(レシーバ);
if (klass == cache.klass){
 method = cache.method;
}
else{
 method = LookUpMethod(klass, 呼び出すメソッドの ID);
 cache.klass = klass;
 cache.method = method;
}
method(レシーバを除く実引数の数，レシーバ，実引数 1，実引数 2，···);
```

図 3　仮想呼び出しの実現
Fig. 3　An implementation of a virtual call.

もっとも宣言型が使えないからといってクラスを推定できないということはない．我々の脱仮想化の実装ではプログラム中の次の箇所からクラスに関する情報を取得し，それをデータフローに沿って伝播することでレシーバのクラスを推定する．

リテラル　リテラルは既知のクラスを持つ．

変数 self　変数 self が指示するインスタンスのクラスは，メソッドを定義するクラスと is-a の関係にあるということができる．

メソッド呼び出し C.new() の返戻値　メソッド呼び出し C.new()（ここで C はクラスを指示する任意の定数とする）の返戻値は，定数 C の指示するクラスのメソッド new() が再定義されていなければ，定数 C の指示するクラスのインスタンスであるといえる．

クラス検査　我々の動的コンパイラはメソッド呼び出しをコンパイルする際に，実行プロファイルを参照してレシーバのクラスが単一か調査し，単一である場合には仮想呼び出しの前にレシーバのクラスが過去のレシーバのクラスと同一か検査するコードを挿入するが，この検査によってレシーバのクラスを唯一に絞り込むことが可能になる[*1]．

クラス検査の挿入から脱仮想化までの流れを**図 4** に示す．図 4 (a) はコンパイル対象の仮想呼び出しである．ここで実行プロファイルに記録されたレシーバ obj のクラスが C のみであったとする．このとき動的コンパイラは仮想呼び出しの前に obj のクラスが C か確認するコードを挿入する．挿入後のコードを図 4 (b) に示す．図 4 (b) では 3 行目でレシーバ obj のクラスが C か検査し，C ならば仮想呼び出しに進み，さもなくば UncommonTrap() を呼び出す．

UncommonTrap() は動的コンパイル済みコードが未実行のパスに到達した際に呼び出す処理である．動的コンパイラは UncommonTrap() をクラス検査に失敗した場合の制御パスに挿入するほかに，未実行のためコンパイル対象から除外した部分への入口にも挿入する．UncommonTrap() の内部では次に示す 3 つの処理を行う．

脱最適化　呼び出し元の動的コンパイル済みコードは UncommonTrap() に到達しないことを前提に最適化したものなので，到達したからには呼び出し元の動的コンパイル済み

*1 クラス検査の挿入を，実行プロファイルからレシーバのクラスの候補が単一といえない場合，たとえば候補が 2 つの場合に実施することは可能だが，今のところ対応していない．

```
 1:  obj.boo();
 2:  obj.foo();
```
(a) ソースコード

```
 3:  if (obj.klass == C)
 4:    obj.boo();
 5:  else
 6:    UncommonTrap();
 7:  if (obj.klass == C)
 8:    obj.foo();
 9:  else
10:    UncommonTrap();
```
(b) クラス検査の挿入

```
11:  if (obj.klass == C)
12:    C::boo(0, obj); // 直接呼び出し
13:  else
14:    UncommonTrap();
15:  if (obj.klass == C)
16:    C::foo(0, obj); // 直接呼び出し
17:  else
18:    UncommonTrap();
```
(c) 脱仮想化

```
19:  if (obj.klass == C)
20:    C::boo(0, obj); // 直接呼び出し
21:  else
22:    UncommonTrap();
23:  C::foo(0, obj); // 直接呼び出し
```
(d) 冗長なクラス検査の除去

図 4　実行プロファイルからレシーバのクラスが唯一と分かるメソッド呼び出しのコンパイル
Fig. 4　Compilation of a method call whose receiver's class the runtime profile told single.

コードを不適切になったものと見なし，無効化する．

到達した未実行パスの記録　どの未実行パスに到達したかを `UncommonTrap()` の呼び出し元のアドレスから求めて実行プロファイルに記録する．この記録により，無効化した処理が後に一定回数インタプリタで実行され，再度コンパイル対象になった際に，当該箇所に再度 `UncommonTrap()` を挿入しないようにする．

インタプリタへの実行の引継ぎ　呼び出し元の動的コンパイル済みコードの実行をインタプリタに引き継ぐ．この結果，`UncommonTrap()` から動的コンパイル済みコードに制御が戻ることはなくなる．

`UncommonTrap()` で単にインタプリタへの実行の引継ぎを行うだけでなく，脱最適化や到達した未実行パスの記録とを行う目的は，実行頻度が高くなりうる制御パスから `UncommonTrap()` を除去することで，`UncommonTrap()` の呼び出しが頻発して実行速度に悪影響をおよぼすのを防ぐことにある．

さて，図 4 (b) のようにクラス検査を挿入してからクラスの推定を行うと，4，8 行目にある仮想呼び出しのレシーバのクラスが唯一 `C` になると分かるので，これらの仮想呼び出しに脱仮想化を適用する．ここでクラス `C` がメソッド `boo()` を定義しているものと仮定すると，これらのメソッド呼び出しの呼び出し先はクラス `C` が定義したメソッド `boo()`，すなわち `C::boo()` になるので，脱仮想化を適用したコードは図 4 (c) に示すとおりとなる．ところで，クラスの推定は脱仮想化のためだけにあるものでなく，冗長なクラス検査を除去する際にも利用できる．たとえば図 4 (c) のコードでは 11 行目と 15 行目で同じ変数 `obj` が指示するインスタンスのクラスを検査しているが，14 行目の `UncommonTrap()` から制御が戻らないことを考えると 15 行目のクラス検査は冗長といえる．そこで我々の動的コンパイラは，このような冗長なクラス検査を除去する．除去後のコードは図 4 (d) に示すとおりとなる．

なお，クラス検査の挿入や未実行パスのコンパイル対象からの除外のように `UncommonTrap()` を挿入する処理には，未実行パスからの制御の合流を除去することで，冗長なクラス検査の除去のような最適化を促進する効果がある．実際，図 4 (c) の 14 行目には `UncommonTrap()` の代わりにメソッド `boo()` を仮想呼び出しするコードを挿入することもできるが，そうすると，仮想呼び出しからは制御が戻るため，11 行目のクラス検査による分岐が 15 行目で合流してしまい，結果として 15 行目で `obj` の指示するインスタンスのクラスが不定になり，15 行目のクラス検査を除去できなくなってしまう．

3.6　脱最適化

脱仮想化のようにプログラムの実行状況に特化した最適化を適用したコードが，実行状況の変化によって不適切になった場合，不適切になったコードの利用を継続させないための措置が必要になる．この措置のことを脱最適化という．脱最適化の実装方法はいくつかあるが[6),10)–12)]，我々は Java の HotSpot 仮想機械[6)] の実装にならい，不適切な部分を含みうる動的コンパイル済みコードの利用をやめることで実装することにする．利用をやめる処理

はコンパイル単位ごとに適用するものとし，やめる時点で当該コードを実行中の場合には，実行をインタプリタに引き継ぐものとする．動的コンパイル済みコードの利用をやめる際には，あわせて処理の呼び出し回数を数えるカウンタを 0 にリセットする．これにより，インタプリタで一定回数実行した後で再度コンパイルを行い，実行の高速化を図る．再コンパイルを即時実施するのではなく，ある程度，時間をおいてから実施する理由は，再度実行状況が変化して脱最適化や再コンパイルが必要になり，それにともなうオーバヘッドが発生することを防ぐためである．

利用をやめる動的コンパイル済みコードを検出する処理と，利用をやめる処理の実現について，順次その詳細を示す．

3.6.1　利用をやめる動的コンパイル済みコードの検出

我々の実装では，利用をやめる動的コンパイル済みコードを検出するために，CRuby の実行時システムを使って Ruby アプリケーションの挙動を監視する．そして，Ruby アプリケーションが実行時システムに次の処理を行うよう要請した際に，要請に対応する処理の実施に先立って，処理の実施によって不適切になりうる動的コンパイル済みコードがあるか調べ，あるなら，その利用をやめる．

(1)　メソッドの再定義，オーバライド，削除
(2)　メソッド `new()` の再定義，オーバライド
(3)　クラスを指示する定数の削除，クラスを指示する定数への再代入

(1) の場合には，脱仮想化を適用したコードの利用をやめる．すなわち，再定義やオーバライド，削除の対象となったメソッドの動的コンパイル済みコードを直接呼び出しする動的コンパイル済みコードの利用をやめる．

(2)，(3) の場合には，クラスの推定に `new()` がもたらすクラスの情報を使いつつ生成した動的コンパイル済みコードの利用をやめる．具体的には，`new()` を再定義したクラスを `C` とおくとき，メソッド呼び出し `C.new()` を含む処理の動的コンパイル済みコードの利用をやめる．

いずれの場合にも，利用をやめる動的コンパイル済みコードの集合を求める処理が必要になるが，我々はこの処理を効率的に実現するために，これらの集合をあらかじめ作成しておくことにした．すなわち，(1) の場合に備えてメソッドごとに集合を作成し，そこに当該メソッドを直接呼び出しする動的コンパイル済みコードを記録し，(2)，(3) の場合に備えて，クラスごとに集合を用意し，そこに当該クラスのメソッド `new()` を呼び出す動的コンパイル済みコードを記録することにした．記録を行うのは動的コンパイラである．これらの集合があれば，(1)〜(3) の場合に利用をやめる動的コンパイル済みコードを求める処理は，単に集合の内容を参照する処理として実現可能になる．

なお，ここで示した利用をやめる動的コンパイル済みコードの集合を求める処理の実現は保守的で，求める集合に利用をやめなくてよい動的コンパイル済みコードを含みうることに注意する必要がある．たとえば (1) の場合について考える．**図 5** (a) に示すようにクラス `S` と，それを継承するクラス `A` が定義されていたものとする．ここでクラス `A` のメソッド `foo()` を動的コンパイルすると，動的コンパイラは `foo()` の中にあるメソッド呼び出し `self.boo()` に脱仮想化を適用して，クラス `S` が定義するメソッド `boo()` への直接呼び出しに変換し，また，脱最適化に備えて，直接呼び出し先のメソッドを再定義やオーバライド，削除する前に利用をやめるべきコードの集合に，`foo()` の動的コンパイル済みコードを登録する．さて，ここで新たにクラス `S` を継承するクラス `T` を定義し，その中でメソッド `boo()` を定義しようとすると（図 5 (b)），この定義はクラス `S` のメソッド `boo()` をオーバライドすることから，我々の実装では，クラス `S` のメソッド `boo()` に対応する集合の中にある動的コンパイル済みコードの利用をすべてやめてしまうが，ここでクラス `A` のメソッド `foo()` の動的コンパイル済みコードについてまで，その利用をやめる必要はない．なぜなら，クラス `T` でメソッド `boo()` をオーバライドしても，メソッド `foo()` 内にあるメソッド呼び出し `self.boo()` の呼び出し先は変化しないからである．

```
class S
  def boo() ··· 略 ··· end
end
class A < S
  def foo()
    self.boo()
  end
end
```

(a) 定義済みのクラス

```
class T < S
  def boo() ··· 略 ··· end
end
```

(b) 新規追加対象のクラス

図 5　冗長な脱最適化が発生するケース
Fig. 5　A case where redundant deoptimization takes place.

また，(2)，(3) の場合については，new() がもたらす情報を使ってクラスを推定したとしても，必ずしもその結果を使って最適化がかかるわけではないが，我々の実装では最適化をかけたか否かにかかわらず new() を呼び出す動的コンパイル済みコードの利用をやめてしまう．動的コンパイル済みコードの利用を冗長にやめてしまったとしても，再コンパイルすればまた最適化がかかるが，脱最適化や再コンパイルのオーバヘッドが生じるのは望ましくない．

3.6.2　動的コンパイル済みコードの利用をやめる処理

3.6.1 項で示した処理で利用をやめる動的コンパイル済みコードを決めたら，それらの利用をやめるために，それぞれに次の処理を適用する．

(1)　利用をやめる動的コンパイル済みコードを新たに呼び出すことをなくすために，次の処理を適用する．

- 仮想呼び出しを通じた呼び出しをなくすために，脱最適化対象の動的コンパイル済みコードへの参照を，コンパイル元のコードに対応するデータ構造から削除する．たとえばメソッドに対応するデータ構造から，対応する動的コンパイル済みコードへの参照を削除する．
- 直接呼び出しを通じた呼び出しをなくすために，脱最適化対象でない動的コンパイル済みコードの中にある直接呼び出しのうち，脱最適化対象の動的コンパイル済みコードを直接呼び出しするものを書き替え，直接呼び出し先を一時的にインタプリタを呼び出すスタブに変更する．再コンパイルを行ったら再度修正して再コンパイルしたコードを直接呼び出しするようにする．

(2)　利用をやめる動的コンパイル済みコードを実行中か否か調べ，実行中であったなら，実行をインタプリタに引き継ぐ．

具体的には，全 Ruby スレッドの C スタックを走査し，C スタック上のフレーム中に記録されている返戻先番地が利用をやめる動的コンパイル済みコード内を指示していたら，返戻先番地を退避したうえで脱最適化ハンドラに書き替える．ここで返戻先番地の退避先は呼び出し元のフレーム，つまり利用をやめる動的コンパイル済みコードのフレーム中のフィールドとし，このフィールドは動的コンパイルの際にフレームポインタの指示先から固定オフセットの位置に確保しておくものとする．返戻先番地を書き替えると，返戻の際には，利用をやめた動的コンパイル済みコードに返戻する代わりに，脱最適化ハンドラを呼び出すことになる．

脱最適化ハンドラは実行をインタプリタに引き継ぐためのもので，呼び出されると，まず，呼び出し元のフレームポインタの指示先から固定オフセットの位置に退避しておいた返戻先番地を取得し，次に，取得した返戻先番地に対応するデバッグ情報を求め，最後に，求めたデバッグ情報を使って動的コンパイル済みコードのフレームをインタプリタのフレームに書き替えてからインタプリタを起動する．

我々は処理 (2) で行う全 Ruby スレッドの C スタックの走査を，Ruby スレッドで実施することにした．その目的は，CRuby が同時に動作する Ruby スレッドを 1 本に限定していることを利用して，他の Ruby スレッドが動作していない状況で走査を行うことにある．CRuby は動作するスレッドの切替えに明示的な処理を必要とするので，走査中にこの切替えの処理を実行しなければ，走査の完了まで他の Ruby スレッドが停止した状態を維持できる．なお，走査の開始時点で C スタック中にある動的コンパイル済みコードのフレームは，すべてどの Ruby スレッドのものかを問わず，いずれも処理 (2) によってインタプリタへ実行を引き継ぎうる状態になっている．ここで処理 (2) によって引き継ぎうる状態とは，フレームに対応する動的コンパイル済みコードから何らかの処理を呼び出し，結果として，動的コンパイル済みコードへの返戻先番地が C スタックに保存されている状態を意味する．処理 (2) は返戻先番地の書き替えによってインタプリタへの引継ぎを実現するので，この状態になっていないと引継ぎを実現できない．走査の開始時点で動的コンパイル済みコードの全フレームがこの状態になっている理由は，走査を担当する Ruby スレッドは C 言語で記述した走査の関数を呼び出して実行中であり，他の Ruby スレッドは C 言語で記述した動作するスレッドを切り替える関数を呼んだ状態で停止しているはずだからである．

ところで，処理 (2) における C スタックの走査の実現には事前に準備が必要になる．なぜなら C スタック上のフレームは単純には上から順にすべてをたどれるとは限らないからである．C スタック上のフレームを順にたどるには，個々のフレームに呼び出し元のフレームを指示するポインタ（フレームポインタ）を保存してあるならば，そのポインタを使って，保存していないならば，フレームのサイズを手掛かりに，順次，次のフレームを見つけていけばよいが，この方法が使えるのは，呼び出し元のフレームポインタを保存してあるか否かを示す情報や，フレームのサイズといった情報が利用できる場合に限られる．我々の動的コンパイラが生成したコードのフレームについては，その個々に呼び出し元のフレームポインタを保存しているので，順次たどることができるが，たとえば C ライブラリのように，我々の動的コンパイラ以外の手段で生成したコードについては，コード生成を行ったコンパイラやアセンブラがフレームをたどるのに必要な情報をデバッグ情報などの形で残しているとは限らないので，次のフレームをたどるのが難しい場合がある．

もっとも処理 (2) の実現にあたって C スタック上のフレームをすべてたどる必要はない．処理 (2) で実施すべき操作は，C スタックに保存されている返戻先番地のうち，利用をやめるコードを指示するものを書き替えることだが，書き替え対象の返戻先番地の保存先は動的コンパイラが生成したコードのフレーム，もしくはそれに隣接するフィールドなので[⋆1]，動的コンパイラが生成したコードのフレームをすべてたどることができれば処理 (2) を実現できる．そこで我々は C スタック上にある，動的コンパイラが生成したコードのフレームのみを含む連続した記憶領域をリスト状に連結することにした．こうすれば C スタック上にあるフレームのうち，動的コンパイラが生成したコードが確保したものはすべてたどれるようになる．

連結を利用して動的コンパイラが生成したコードのフレームをたどる要領について，**図 6** を使って詳述する．図 6 は走査対象の C スタックを表しており，その中には動的コンパイラが生成したコードのフレームのみを含む連続した記憶領域が 2 つある．これらの記憶領域のうち，C スタックの底に近い方を記憶領域 (1)，もう一方を記憶領域 (2) と略記することにする．図 6 には記憶領域 (1)，(2) を結ぶ連結があるが，これは具体的には記憶領域 (1) に最後に積んだフレーム，つまり C2I スタブのフレームを指示するポインタであり，記憶領域 (2) に最初に積んだ I2C スタブのフレームに格納してある．ここで C2I スタブと I2C スタブはともに動的コンパイラが生成するコード片であり，それぞれの概要は次に示すとおりである．

C2I スタブ　動的コンパイル済みコードからの出口となるスタブの 1 種で，呼び出し先がインタプリタの際に利用する．

I2C スタブ　動的コンパイル済みコードへの入口となるスタブの 1 種で，呼び出し元がインタプリタの際に利用する．

図 6 のように記憶領域 (1)，(2) を連結しておけば，動的コンパイラの生成したコードが積んだフレームを漏れなくたどることは難しくない．具体的には，まず，スレッド局所変数 `last_compiled_fp` を参照して動的コンパイラが生成したコードのフレームのうち，最後に積んだものを求める．スレッド局所変数 `last_compiled_fp` の値は，動的コンパイラが生成したコード（スタブもしくは動的コンパイル済みコード）が C で実装した関数を呼ぶ前に設定する．

⋆1 我々が実装した動的コンパイラは IA-32 向けのもので，IA-32 では返戻先番地を呼び出し元のフレームに隣接するフィールドに保存する．

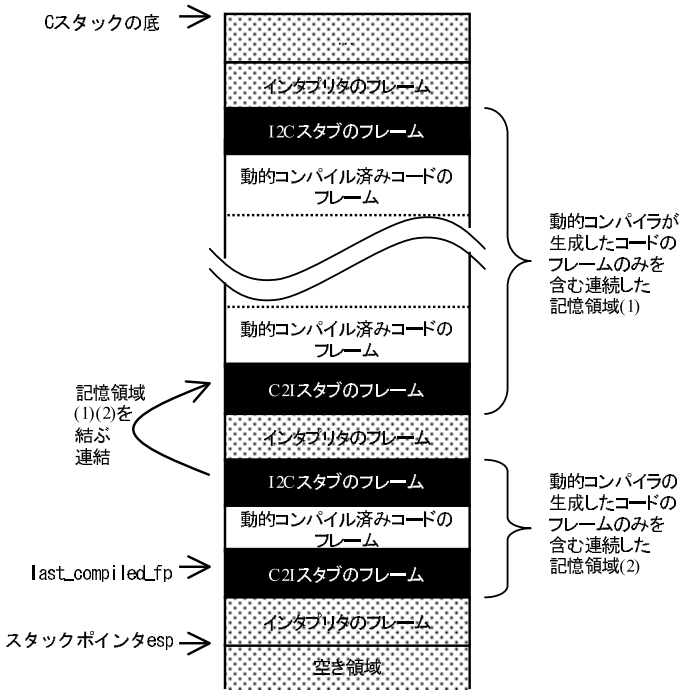

図 6　C スタック
Fig. 6　C stack.

最後に積んだフレームを見つけたら，フレーム内に保存しておいた呼び出し元のフレームポインタを順次たどっていく．するといずれ I2C スタブなど，動的コンパイル済みコードへの入口となるスタブのフレームに到達する．到達したか否かはフレームに保存されている返戻先番地をみれば判断できる．具体的には，返戻先番地が動的コンパイル済みコードを格納する記憶領域の外を指示していたら，そのフレームは入口のスタブのフレームと判断できる．入口のフレームには，次にたどるべき動的コンパイラが生成したコードのフレームがある場合，そのフレームポインタ（図 6 では記憶領域 (1)，(2) を結ぶ連結）を保存してあるので，これをたどって次のフレームを求め，たどる処理を継続する．保存してあったポインタが，スレッド局所変数 `last_compiled_fp` の初期値 `NULL` であった場合には，最後までたどったことになり，たどる処理は完了である．

なお，動的コンパイラが生成したコードのフレームのみを含む連続した記憶領域間を結ぶ連結を維持管理する役割はスタブが担う．C2I スタブおよび I2C スタブが実施する処理を擬似コードで表現すると**図 7** に示すとおりとなり，その 6，22，27，32 行目に維持管理の処理がある．これらの処理はスタブ呼び出しのたびに実行されてオーバヘッドの発生源となる．

C2I スタブでは，まず 6 行目で，脱最適化に備えて，連続領域の最後に位置する C2I スタブのフレームへの参照をスレッド局所変数 `last_compiled_fp` に格納し，次に呼び出し規約の違いを吸収する処理を行う．すなわち，8 行目で CRuby が提供する関数 `rb_vm_set_finish_env(th)` を呼び出してインタプリタ以外の箇所からインタプリタを呼び出す前に必要な処理を行い，9〜10 行目でインタプリタが利用する各種フレームを作成する．そして 12 行目で，インタプリタから例外が発生した状態で返戻する場合に備えて `_setjmp()` を実施してから 14 行目でインタプリタを呼び出す．インタプリタから例外が発生した状態で返戻してきた場合には 17 行目に制御を移し，動的コンパイル済みコードにおける例外処理を開始する．なお実際には，12 行目の `_setjmp()` にはオーバヘッド軽減のために最適化を適用している[20]．

I2C スタブでは，最初に 22 行目で局所変数 `previous_compiled_fp` に前の動的コンパイル済みコードのフレームの連続領域への参照を格納する．格納先は I2C スタブのフレームポインタの指示先から固定オフセットの位置にあるフィールドである．この固定オフセット値はフレームをたどる際に I2C スタブのフレームから前の動的コンパイル済みコードのフレームの連続領域への参照を取得する処理を実現する際に必要になる．格納が済んだら，動的コンパイル済みコードを呼び出す．呼び出しから例外が発生していない状況で返戻した場合の返戻先は 27 行目で，ここでは動的コンパイル済みコードのフレームを格納する最後の連続領域への参照を保持するスレッド局所変数 `last_compiled_fp` に局所変数 `previous_compiled_fp` の値を復帰し，29 行目で呼び出し元に返戻する．動的コンパイル済みコードから例外が発生した状態で返戻した場合の返戻先は 32 行目だが，ここでは 27 行目と同様に復帰の処理を行ったあとで 33 行目から `_longjmp()` によってインタプリタの例外ハンドラに返戻する．

4. 評　　価

我々が実装した動的コンパイラおよび動的最適化が Ruby アプリケーションの実行速度に与える影響を Ruby Benchmark Suite を使って評価した．評価に使用した計算機は Dell

```
1:   // 動的コンパイル済みコードのフレームを格納する最後の連続領域への参照
2:   __thread void* last_compiled_fp = NULL;
3:
4:   c2i_stub(target){
5:     // 脱最適化に備えた準備
6:     last_compiled_fp = このスタブのフレームポインタ;
7:     // 呼び出し規約の違いの吸収
8:     rb_vm_set_finish_env(th);
9:     Ruby フレームの作成および実引数の詰め込み;
10:    制御フレームの作成;
11:    jmp_buf buf;
12:    if (_setjmp(buf) == 0){
13:      // インタプリタによる target の実行
14:      return (vm_exec(th));
15:    }
16:    else{
17:      返戻先番地に対応する例外エントリに返戻;
18:    }
19:  }
```

C2I スタブ

```
20:  i2c_stub(target){
21:    // 脱最適化に備えた準備
22:    void* previous_compiled_fp = last_compiled_fp;
23:    // 動的コンパイル済みコードの呼び出し
24:    rerurnValue = target->compiled_entry();
25:  通常の返戻先：
26:    // 脱最適化に備えた準備
27:    last_compiled_fp = previous_compiled_fp;
28:    // インタプリタへの返戻
29:    return (returnValue);
30:  例外発生時の返戻先 (例外エントリ):
31:    // 脱最適化に備えた準備
32:    last_compiled_fp = previous_compiled_fp;
33:    _longjmp でインタプリタの例外ハンドラに返戻
34:  }
```

I2C スタブ

図 7　スタブの実装
Fig. 7　Stub implementations.

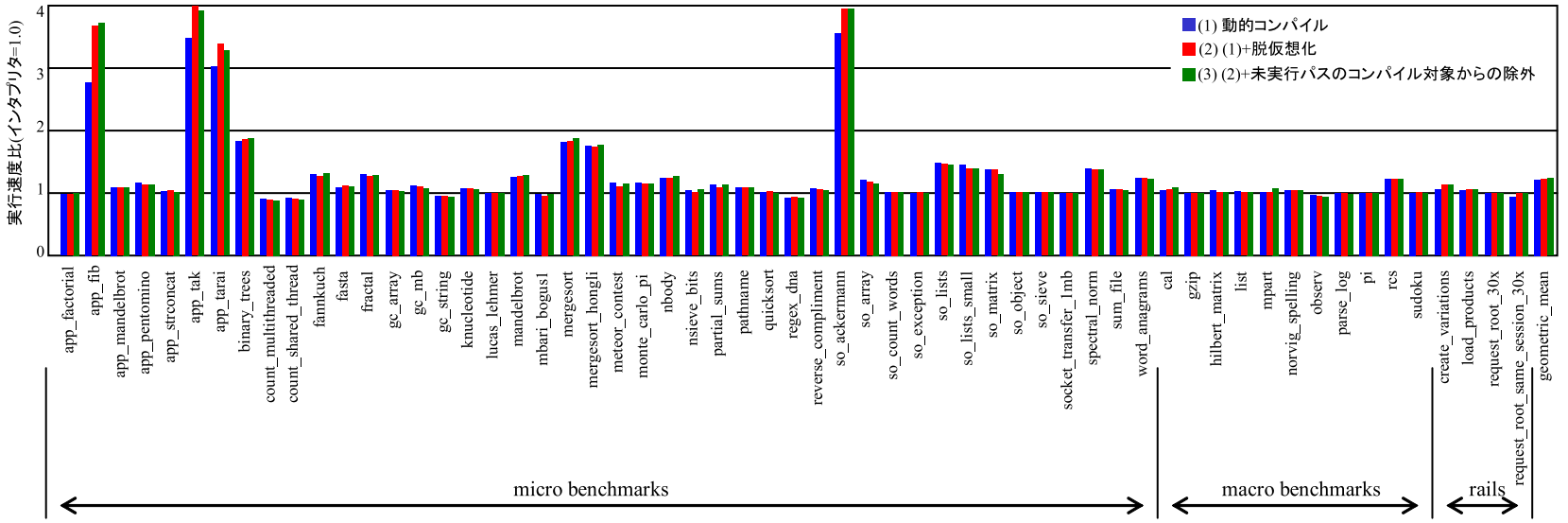

図 8　評価結果
Fig. 8　Benchmark results.

PowerEdge T100（Core 2 Duo E7300 2.66 GHz，RAM 2 GByte），OS は Ubuntu 10.04 LTS である．

評価結果を**図 8** に示す．図 8 において，動的コンパイル済みコードの速度は，実行プロファイル採取後の動的コンパイルが終了してから計測したものなので，動的コンパイルや実行プロファイルの採取にともなうオーバヘッドを含まない．動的コンパイルの対象とする処理はインタプリタで 8 回以上実行したものとした．また，実行プロファイル取得用の動的コンパイル済みコードは 8 回呼び出された時点で自身を無効化し，コンパイラに再コンパイルを促す．個々のベンチマークを 20 回実行し，そのうち最小の実行時間を測定結果として採用した．図 8 は次に示す 3 つの条件で実行を行った場合の実行速度が，インタプリタ実行に対して何倍速くなっているかを示す．

(1)　動的コンパイラあり．
(2)　(1) に加え脱仮想化を適用．
(3)　(2) に加え未実行のパスをコンパイル対象からの除外を適用．

また，(3) の評価を行う際に，実行時間のほかに次の値を測定した．測定結果を**表 1** に示す．

実行時間のぶれ　実行時間の標準偏差を平均値で除した値．実行時間が実行ごとにどの程度の割合で変化するかを表す．

コンパイル時間　動的コンパイルにかかった時間．

命令数　1 回以上動的コンパイルした処理に対応する CRuby の中間表現の命令数の総和．

脱最適化の発生回数　脱最適化が発生した回数．脱最適化は，すべて実行時間の測定を開始するより前に発生しているので，測定した実行時間は脱最適化のオーバヘッドを含まない．

図 8 より，実装したすべての最適化を有効にすることで，インタプリタ実行と比較して 23.8%高速化できることが分かる．また，動的コンパイルにより 21.6%，脱仮想化により 1.1%，未実行パスのコンパイル対象からの除外により 0.7%高速化できることが分かる．1.1%や 0.7%といった差異は大きなものでなく，実行時間のぶれから発生した無意な差異である可能性もあるが，表 1 から実行時間のぶれが相乗平均で 0.30%に収まっていると分かることから，ある程度は有意であると考える．

表 1　実行時間のぶれ，コンパイル時間，命令数および，脱最適化の発生回数
Table 1　Execution time inaccuracy, compilation time, instruction count and devirtualization count.

種別	ベンチマーク項目名	実行時間のぶれ（%）	コンパイル時間（μs）	命令数	脱最適化の発生回数
micro	app_factorial	4.08	2,337	298	2
	app_fib	0.08	2,145	310	2
	app_mandelbrot	0.16	2,906	408	4
	app_pentomino	0.19	9,489	959	2
	app_strconcat	0.08	2,246	255	0
	app_tak	0.13	2,202	276	0
	app_tarai	0.03	2,023	276	0
	binary_trees	5.62	4,449	574	4
	count_multithreaded	5.08	2,285	268	0
	count_shared_thread	3.35	2,509	271	0
	fannkuch	5.31	3,319	412	0
	fasta	0.22	4,555	468	0
	fractal	0.09	3,424	431	2
	gc_array	4.36	10,517	273	0
	gc_mb	0.86	2,485	252	0
	gc_string	0.49	3,016	286	0
	knucleotide	1.02	4,414	423	0
	lucas_lehmer	0.03	2,358	289	0
	mandelbrot	0.05	3,826	657	2
	mbari_bogus1	22.68	3,326	397	3
	mergesort	0.18	3,568	461	2
	mergesort_hongli	0.05	3,399	427	0
	meteor_contest	0.05	17,283	2,370	14
	monte_carlo_pi	0.06	2,442	281	0
	nbody	0.17	6,063	705	0
	nsieve_bits	0.08	3,068	385	0
	partial_sums	0.07	3,525	442	0
	pathname	0.08	8,402	937	0
	quicksort	0.13	2,898	308	0
	regex_dna	0.02	3,241	338	0
	reverse_compliment	0.07	2,628	302	0
	so_ackermann	0.01	2,274	278	0
	so_array	0.83	2,570	301	0
	so_count_words	0.47	3,224	353	0
	so_exception	0.16	2,962	342	0
	so_lists	0.20	2,717	328	0
	so_lists_small	0.66	2,705	328	0
	so_matrix	0.55	3,701	451	0
	so_object	0.12	3,216	370	0
	so_sieve	0.11	2,783	335	0
	socket_transfer_1mb	0.01	2,984	351	2
	spectral_norm	0.13	3,859	438	0
	sum_file	0.02	2,344	271	0
	word_anagrams	0.71	2,914	328	0
macro	cal	0.80	22,676	1,446	4
	gzip	0.16	2,852	306	0
	hilbert_matrix	1.38	20,225	1,398	4
	list	1.41	3,852	435	0
	mpart	3.69	3,007	383	0
	norvig_spelling	0.16	5,445	541	0
	observ	0.20	3,531	373	0
	parse_log	0.11	3,083	441	2
	pi	0.05	2,639	329	0
	rcs	0.18	3,445	423	0
	sudoku	0.13	3,915	613	2
rails	create_variations	7.31	230,139	33,225	151
	load_products	0.42	183,010	26,201	132
	request_root_30x	10.02	418,935	55,958	174
	request_root_same_session_30x	2.29	381,023	51,194	146

図 8 をみると脱仮想化が app_fib や app_tak，app_tarai，so_ackermann の性能向上に明確に貢献していることが分かるが，これらのベンチマークでは脱仮想化されたメソッド呼び出しの実行頻度が高く，秒間 1,000 万回を超えていた．メソッド呼び出しの実行頻度が高い項目は他にも fannkuch などいくつかあるが，それらにも脱仮想化を適用できていないか今後検討する必要がある．

コンパイル時間については，表 1 から，おおむね命令数に比例して増加する傾向があることが分かる．コンパイル時間は，命令数が最も大きな request_root_30x でも 0.42 秒弱であり，命令数の少ない micro や macro では 0.01 秒に満たないケースが大半を占める．また，脱最適化が発生しているベンチマーク項目のコンパイルに長い時間がかかる傾向があるが，これは脱最適化後に再コンパイルを行ったためである．

5. 結　　論

Ruby アプリケーション実行の高速化を目的として動的コンパイラおよび動的最適化を実装した．実装した動的最適化は，脱仮想化と，未実行のパスのコンパイル対象からの除外である．評価の結果，Ruby Benchmark Suite の実行を 23.8%高速化できることが分かった．

参 考 文 献

1) Aycock, J.: A brief history of just-in-time, *ACM Comput. Surv.*, Vol.35, No.2, pp.97–113 (2003).
2) Tatsubori, M., Tozawa, A., Suzumura, T., Trent, S. and Onodera, T.: Evaluation of a just-in-time compiler retrofitted for PHP, *VEE '10: Proc. 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, New York, NY, USA, pp.121–132, ACM (2010).
3) Lee, S.-W., Moon, S.-M., Jung, W.-K., Oh, J.-S. and Oh, H.-S.: Code size and performance optimization for mobile JavaScript just-in-time compiler, *INTERACT-14: Proc. 2010 Workshop on Interaction between Compilers and Computer Architecture*, New York, NY, USA, pp.1–7, ACM (2010).
4) Kotzmann, T., Wimmer, C., Mössenböck, H., Rodriguez, T., Russell, K. and Cox, D.: Design of the Java HotSpot™client compiler for Java 6, *ACM Trans. Archit. Code Optim.*, Vol.5, No.1, pp.1–32 (2008).
5) Suganuma, T., Yasue, T., Kawahito, M., Komatsu, H. and Nakatani, T.: Design and evaluation of dynamic optimizations for a Java just-in-time compiler, *ACM Trans. Program. Lang. Syst.*, Vol.27, No.4, pp.732–785 (2005).
6) Paleczny, M., Vick, C. and Click, C.: The Java Hotspot(tm) Server Compiler,

USENIX Java Virtual Machine Research and Technology Symposium, pp.1–12 (2001).
7) Bebenita, M., Brandner, F., Fahndrich, M., Logozzo, F., Schulte, W., Tillmann, N. and Venter, H.: SPUR: A Trace-Based JIT Compiler for CIL, Microsoft Research Technical Report MSR-TR-2010-27, Microsoft, Yokohama (2010).
8) 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, *PLDI '09: Proc. 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, pp.465–478, ACM (2009).
9) Arnold, M., Fink, S., Grove, D., Hind, M. and Sweeney, P.F.: Adaptive optimization in the Jalapeño JVM, *OOPSLA '00: Proc. 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, pp.47–65, ACM (2000).
10) Ishizaki, K., Kawahito, M., Yasue, T., Komatsu, H. and Nakatani, T.: A study of devirtualization techniques for a Java Just-In-Time compiler, *OOPSLA '00: Proc. 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, pp.294–310, ACM (2000).
11) 石崎一明，安江俊明，川人基弘，小松秀昭：コード書換えによる動的メソッド呼び出しの直接 devirtualization，情報処理学会論文誌，Vol.43, No.1, pp.124–136 (2002).
12) 今城哲二，布広永示，岩沢京子，千葉雄司：コンパイラとバーチャルマシン，オーム社 (2004).
13) Suganuma, T., Yasue, T. and Nakatani, T.: A region-based compilation technique for a Java just-in-time compiler, *PLDI '03: Proc. ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, New York, NY, USA, pp.312–323, ACM (2003).
14) Hölzle, U., Chambers, C. and Ungar, D.: Debugging optimized code with dynamic deoptimization, *PLDI '92: Proc. ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, New York, NY, USA, pp.32–43, ACM (1992).
15) The Rubinius team: Rubinius (2010). http://rubini.us
16) Ruby, S., Thomas, D., Hasson, D.H., et al.: *Agile web development with Rails, 4th edition*, The Pragmatic Programmers, LLC, Lewisville, Texas (2010).
17) Cangiano, A.: Ruby-Benchmark-Suite (2010). http://github.com/acangiano/ruby-benchmark-suite
18) Intel Corporation: Intel64 and IA-32 architectures software developer's manuals (2010). http://www.intel.com/products/processor/manuals
19) Manjunath, G. and Krishnan, V.: A small hybrid JIT for embedded systems, *SIGPLAN Not.*, Vol.35, No.4, pp.44–50 (2000).
20) 村田俊哉，石井直也，千葉雄司，土居範久：Ruby 向け動的コンパイラにおける例外処理の実装 (2010). SWoPP 2010.

(平成 22 年 7 月 6 日受付)
(平成 22 年 11 月 12 日採録)

石井　直也
1985 年生．2010 年中央大学大学院理工学研究科情報科学専攻博士課程前期課程修了．

村田　俊哉
1985 年生．2010 年中央大学大学院理工学研究科情報科学専攻博士課程前期課程修了．

千葉　雄司（正会員）
1972 年生．1997 年慶應義塾大学大学院理工学研究科計算機科学専攻修士課程修了．ルネサスソリューションズにてコンパイラの開発に従事．2008 年より中央大学大学院客員講師を兼任．

土居　範久（名誉会員）

中央大学研究開発機構教授，慶應義塾大学名誉教授．現在，ブロードバンドワイヤレスフォーラム会長，文部科学 HPCI 計画推進委員会主査，独立行政法人科学技術振興機構社会技術研究開発センター「問題解決型/サービス科学/研究開発プログラム（NEXER）」プログラム総括，独立行政法人科学技術振興機構社会技術研究開発センター参与，NPO 法人日本セキュリティ監査協会会長等．専門は計算機科学および情報セキュリティ．