情報処理学会論文誌　プログラミング　Vol. 4　No. 1　123–133 (Mar. 2011)

# Ruby 向け動的コンパイラにおける例外処理の最適化

村田俊哉[†1]　石井直也[†1]
千葉雄司[†1]　土居範久[†1]

CRuby 向け動的コンパイラでは，例外処理の実現にあたり，C で記述した関数で発生する例外に対処する方法が問題になる．C で記述した関数は Ruby が提供する C API を使って Ruby の例外を投げることができるが，このとき既存の CRuby の実装では例外を_longjmp() によって投げるので，これをとらえるためには_setjmp() する必要がある．しかしながら_setjmp() は大きなコストがかかる処理であり，頻繁に実行すると性能劣化の原因となる．そこで我々は，CRuby 向け動的コンパイラの開発にあたって，_setjmp() による性能劣化を軽減する 2 つの技法を開発した．開発した技法の 1 つは_setjmp() の呼び出し元と呼び出し先にある命令のうち，C で記述された関数が投げる例外の捕捉に不要なものを除外する技法であり，もう 1 つは，_setjmp() で初期化したバッファを使いまわすことで，_setjmp() が必要になる箇所を減らす技法である．これらの技法の効果を評価した結果，例外処理向けの_setjmp() の実行時間を 73%削減でき，また，Ruby Benchmark Suite における Ruby アプリケーション全体の実行時間を 1.4%削減できることが分かった．

## Optimizing Exception Handling in a Dynamic Compiler for Ruby

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

One of the problems found in implementing a dynamic compiler for Ruby stays in handling exceptions thrown by functions implemented in C language. Functions written in C can throw Ruby exceptions using C API provided by Ruby and CRuby implements this API using _longjmp(). Thus, in order to catch the exception, dynamically compiled codes for CRuby must call _setjmp(), but it causes performance loss especially when frequently called. To cope with this problem, we developed two techniques in the development of our dynamic compiler for CRuby. The first technique eliminate redundant instructions in _setjmp() and around its call site, and the other technique eliminates _setjmp() call site by reusing jump buffer. We evaluated their effect and found that they eliminate 73% of _setjmp() execution time for the exception use, and 1.4% of overall Ruby Benchmark Suite application execution time.

## 1. はじめに

プログラミング言語処理系を実装するうえで，パフォーマンスへの影響が大きいことから注意深く実装すべき機能の 1 つに例外処理[1)] がある．例外処理とは，プログラム実行中に例外的事象が発生した箇所からそれを対処するハンドラへ大域ジャンプする機能である．たとえば Ruby[2)] では，キーワード begin，raise，rescue，else，ensure を使用することで，例外処理を利用できる．具体的には，例外ハンドラが有効になる範囲の先頭にキーワード begin を記述し，末尾にキーワード rescue または，else，ensure を記述する．そうすると，例外ハンドラが有効な範囲内で例外が発生した場合には，キーワード rescue で始まる節に，しなかった場合には，キーワード else で始まる節に制御が移る．キーワード ensure で始まる節には例外発生の有無にかかわらず最後に制御が移る．また，例外を発生させる場合には，キーワード raise を使用する．

例外処理を利用するプログラムの例を **図 1** に示す．図 1 は Ruby を使って記述したもので，実行すると，2 行目にあるキーワード begin で例外ハンドラが有効になった後で，4 行目でメソッド function() を呼び出し，呼び出し先の 14 行目でキーワード raise によって例外を発生させると，5 行目にあるキーワード rescue の位置に制御が移る．

Ruby では，このように例外処理を行うが，例外処理を行うための大域ジャンプには，準備や実行が大きなオーバヘッドとなりうるため，例外処理の実装は慎重に検討する必要がある．

例外処理を実装する方法は，つぎの 3 種類がよく知られている．

- 表引き法[3)–6)]
  例外が発生した点でのプログラムカウンタの値から対応する例外ハンドラを表引きで探してジャンプする方法．各メソッドについてプログラムカウンタと例外ハンドラの対応表を作成しておき，例外が発生した際には，その時点のプログラムカウンタの値を鍵

†1 中央大学
Chuo University


© 2011 Information Processing Society of Japan

```
1:   ...
2:     begin
3:       # 通常処理時
4:       function()
5:     rescue xtype
6:       # 例外発生時
7:       exception_handler(xtype)
8:     end
9:   ...
10:  def function()
11:    ...
12:    if (xtype)
13:      # 例外発生 大域ジャンプする
14:      raise xtype
15:    end
16:    ...
17:  end
```

**図 1**　例外処理を利用する Ruby ソースコード
Fig. 1　Ruby source code using exception handling.

として対応表を引き，例外ハンドラを求めてジャンプする．例外ハンドラがみつからなかった場合，つまりメソッドの中に対応する例外ハンドラがなかった場合にはメソッドから返戻し，返戻先で返戻時のプログラムカウンタの値を鍵として対応表を引き，例外ハンドラを求める操作を，みつかるまで再帰的に繰り返す．

- 2 返戻値法[7)–11)]
  メソッドの返戻値を通常の返戻値と例外用の返戻値の 2 つにすることで実現する方法．例外発生時には，例外用の返戻値に例外の値を格納し，返戻する．そして，メソッド呼び出しからの返戻点の直後に，例外用の返戻値をチェックすることで例外が発生しているか調べる．例外が発生していれば，例外ハンドラへジャンプする．そのメソッドに例外ハンドラがなければ，例外用の返戻値に例外の値をセットしたまま返戻する．
- setjmp 法[7),12),13)]
  例外が発生する可能性のある範囲の先頭で`_setjmp()` を行い，例外が発生した際に`_longjmp()` で`_setjmp()` した位置に大域ジャンプする方法．大域ジャンプ後，`_setjmp()` の返戻値をチェックし，返戻値が 0 以外だったら`_longjmp()` から戻ったと判断して例外処理を行う．たとえば図 1 のプログラムに記述した例外処理を`_setjmp()` によって実現すると**図 2** のプログラムになる．

```
1:   ...
2:     if ((xtype = _setjmp(jmpbuf)) == 0) {
3:       // 通常処理時
4:       function();
5:     } else {
6:       // 例外発生時
7:       exception_handler(xtype);
8:     }
9:   ...
10:  void function()
11:  {
12:    ...
13:    if (xtype) {
14:      // 例外発生 大域ジャンプする
15:      _longjmp(jmpbuf, xtype);
16:    }
17:    ...
18:  }
19:
```

**図 2**　setjmp 法による例外処理の実装
Fig. 2　An exception handling implementation using `_setjmp()`.

図 2 では，まず例外が発生する可能性のある範囲の先頭（2 行目，図 1 の 2 行目にあたる部分）で`_setjmp()` をする．`_setjmp()` の呼び出し時には，返戻値は 0 であるため，`if` 文の中に入り通常処理を行う．そして，4 行目で `function()` 関数を呼び出し，呼び出し先の 15 行目（図 1 の 14 行目，`raise` を行うところ）に進んだら，`_longjmp()` によって例外を投げる．すると大域ジャンプして，制御が`_setjmp()` した位置（2 行目）に戻る．このとき`_setjmp()` の戻り値が 0 以外になるため，制御が 5 行目の `else` 節の中（図 1 の 5 行目にある `rescue` に相当する処理）に移る．
なお，`setjmp()/longjmp()` の実装にはいくつかバリエーションがある場合がある．POSIX は `setjmp()/longjmp()` の際にシグナルコンテキストの退避復帰を行うか否かを規定してないが，`setjmp()/longjmp()` の実装でシグナルコンテキストを保存する環境では，シグナルコンテキストを保存しないものを`_setjmp()/_longjmp()` という名称で提供することがある．また，シグナルマスクの退避復帰を行うか否かをプログラム上に表記可能にした版に `sigsetjmp()/siglongjmp()` がある．本論文では実験に用


© 2011 Information Processing Society of Japan

いた Ruby 1.9.1-p0 が例外処理の際にシグナルマスクの退避や復帰をしないことから，文中に現れる `setjmp()/longjmp()` の名称を `_setjmp()/_longjmp()` に統一する．

これら 3 つの実装方法の中で，一般的な状況，つまり例外の発生頻度が低い状況で最も高速に動作するのは表引き法である．なぜなら表引き法では他の実装方法と異なり，例外の発生に備えた比較や分岐といった処理が不要だからである．例外処理の実装にあたっては実行効率への配慮から表引き法を採用することが多いが，実装上の問題から表引き法を採用できないケースもあり，そういった場合には他の手段を利用することになる．

表引き法を利用できないケースの 1 つに Ruby から C 言語で記述した関数（C ライブラリと記述する）を呼び出し，そこから Ruby の例外を投げる場合がある．Ruby では，C ライブラリで例外を投げる場合には，Ruby が C ライブラリに提供する Application Programming Interface（C API と略記する）の 1 つである `rb_raise()` を使うよう規定している．`rb_raise()` は呼び出されると何らかの手段によって例外ハンドラに制御を移すが，ここで 2 返戻値法や表引き法を利用することはできない．なぜなら，このケースでは，`rb_raise()` を呼び出すのは C ライブラリだが，例外ハンドラは C ライブラリの呼び出し元にあるので，例外ハンドラに到達するためには，まず C ライブラリが C スタックに積んだフレームを除去しなくてはならないが，2 返戻値法や表引き法のように，C スタックのフレームを順次破棄してゆく方法では，次の理由から，C ライブラリが積んだフレームを除去できないからである．

- 2 返戻値法でフレームを除去するのはメソッドから返戻する処理なので，2 返戻値法で C ライブラリが積んだフレームを除去するためには，C ライブラリの中に 2 返戻値法のためのコード，つまり例外の発生を検知して，検知したら呼び出し元に返戻するコードを挿入しておかなければならない．しかしながら Ruby の言語仕様は C ライブラリの中にそのようなコードを挿入せよとは規定しておらず，したがって C ライブラリの中にそのようなコードは入っていないから，2 返戻値法で C ライブラリが積んだフレームを除去することはできない．
- 表引き法を利用するにはプログラムカウンタと例外ハンドラの対応表が必要になるが，C コンパイラはこの対応表を出力しないから，C ライブラリ中の関数に表引き法を適用することはできない．表がなかった場合には C ライブラリ中の関数と見なし，その中に例外ハンドラはないから対応するフレームを破棄する，という実装も現実的でない．なぜなら C ライブラリ中の関数が積んだフレームを破棄するには，その大きさを知る必要があるが，C ライブラリ中の関数がそのような情報を提供するとは限らないからである．

これらの問題を解決するため，Ruby の実装の 1 つである CRuby では setjmp 法を利用している．具体的には，Ruby 仮想機械の内部で `_setjmp()` を実施しておき，C ライブラリ経由で `rb_raise()` が呼び出された場合には，そこから `_longjmp()` で Ruby 仮想機械に戻って例外ハンドラを探索する．なお，CRuby では Ruby 仮想機械内での例外ハンドラの探索，つまり Ruby で記述したプログラム内での例外ハンドラの探索には表引き法を利用している．

さて，我々は CRuby 向けに動的コンパイラの開発を行っており，我々が実装した CRuby 向け動的コンパイラでも，パフォーマンスへの配慮から，CRuby と同様にコンパイル済みコード内で発生した例外は表引き法で処理し，C ライブラリで発生した例外を受け取る場合や Ruby インタプリタからコンパイル済みコードに例外を渡す際に限って setjmp 法で処理するが，頻繁に `_setjmp()` を実行すると，その実行オーバヘッドが問題になる．そこで `_setjmp()` のオーバヘッドを軽減するために，2 つの技法を開発した．開発した技法の 1 つは `_setjmp()` の呼び出し元と呼び出し先にある命令のうち，C ライブラリが投げる例外の捕捉に不要なものを除外する技法であり，もう 1 つは，`_setjmp()` で初期化したバッファを使いまわすことで，`_setjmp()` が必要になる箇所を減らす技法である．

本論文ではこれらの技法について詳述し，その効果を示す．本論文の構成は次に示すとおりである．まず，次章で関連研究について述べ，3 章で CRuby 向け動的コンパイラとその例外処理の実装について述べる．4 章で `_setjmp()` のオーバヘッド削減する提案技法について述べる．5 章では，提案技法が実行速度に与える影響を調査した結果を示す．最後に 6 章で結論を述べる．

## 2. 関 連 研 究

例外処理の実現方法が仮想機械の内部と外部（C 言語で実装したライブラリ）で異なるのは CRuby だけでなく，Java の HotSpot 仮想機械[5),6)] も同様であり，これらの仮想機械には，外部で発生した例外に対応する仕組みが実装されている．たとえば Java では，C 言語で実装したメソッド（ネイティブメソッド）の中で例外が発生した場合には，発生した例外を 2 返戻値法で仮想機械に引き渡すよう規定している．このため，HotSpot 仮想機械はネイティブメソッド呼び出しから仮想機械に制御が戻るたびに例外が発生しているかチェックし，発生していたら仮想機械内にある表引き法の例外処理ルーチンに例外を引き渡す．2 返戻値法のチェックにかかるコストはネイティブメソッド呼び出しにともなうその他の処理に



© 2011 Information Processing Society of Japan

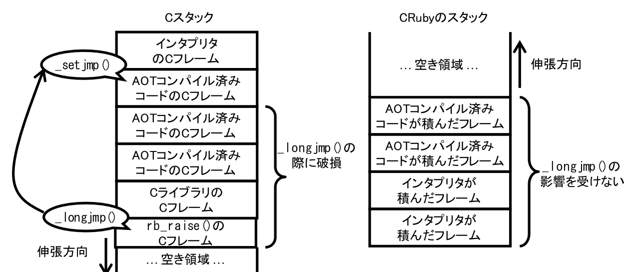



**図 3**　C ライブラリから例外を投げた際に CRuby インタプリタのスタックが受ける影響
Fig. 3　CRuby interpreter stacks affected by an exception thrown in C library.

かかるコストに比べて小さいので，ネイティブメソッド呼び出しのたびにチェックを実施してもパフォーマンスに与える影響は大きくない．

一方，Ruby が提供する例外発生のための C API である `rb_raise()` は setjmp 法以外での実装が容易でないが，`_setjmp()` の実行にかかるオーバヘッドは必ずしも小さくないので，たとえば C ライブラリを呼び出すたびに実行するとパフォーマンスに無視できない影響を与える．Ruby でも言語仕様を変更すれば Java と同様に 2 返戻値法で C ライブラリ内の例外処理を実現できるが，言語仕様を変更すると多くの C ライブラリの改修が必要になる．プロセッサの中には `_setjmp()` に相当する処理のオーバヘッドを軽減する機能を持つものもあるが[15]，そういった機能を持たないプロセッサもある．特定のプロセッサに依存することなくこの問題を回避する手段として，五嶋らは，CRuby 用 AOT コンパイラの実装にあたって，`_setjmp()` の呼び出し箇所を初めてコンパイル済みコードを呼び出したときに限る技法を提案した[16]．五嶋らの技法ではコンパイル済みコードから C ライブラリを呼び出し，そこで例外を投げるために `_longjmp()` すると C スタックにおいてあるコンパイル済みコードの C フレームが破壊されてしまうが（**図 3** 参照），それに備えて，実行コンテキストを原則として C スタックに配置せず，かわりに CRuby インタプリタのスタックに配置する．そして，`_longjmp()` によって C フレームが破壊された場合には，ヒープ上にある CRuby インタプリタのスタックにある情報を使って C フレームを復元し，実行を継続する．

なお，我々の動的コンパイラは，五嶋らの実装とは異なり，実行コンテキストを C フレームに保存するため，`_longjmp()` によって C フレームが破壊されるとプログラムの実行を継続できなくなる．そのため，我々の動的コンパイラに五嶋らの技法をそのまま適用することはできない．我々は五嶋らのアイデアを応用できないか考察し，その結果，4.1 節で述べるように，動的コンパイラが生成したコードでは `_setjmp()` を実行せずに済ます技法を考案したが，試作，評価の結果から採用を見送った．

例外処理向けの最適化技法には，Cierniak らが提案した冗長な例外生成を省略する技法[17]や Ogasawara らが提案した例外が通過するパスをインライン展開して局所ジャンプによる例外処理を実施可能にする技法[18]がある．これらの技法は Java 向けに開発されたものだが，Ruby 向けにも有用である可能性はある．ただし我々の例外処理の実装ではまだこれらの技法をとりいれてはいない．

## 3. CRuby 向け動的コンパイラ

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 節で動的コンパイラの起動から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 節で例外処理の実装について述べる．

### 3.1 動的コンパイラの起動から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，CRuby（version 1.9.1-p0）用のライブラリとして実装した．CRuby は，インタプリタを利用してアプリケーションを実行するが，我々は，このインタプリタに動的コンパイルするライブラリを呼び出す処理の追加と動的コンパイル済みコードを呼び出す処理を加え，実装したライブラリで動的コンパイルするようにした．具体的には，インタプリタのメソッド・ブロック呼び出しを監視し，呼び出し回数が閾値を超えたメソッドまたはブロックをコンパイルする．そして，次に呼び出す際に，コンパイル済みコードを呼び出すようにする．コンパイル済みコードを呼び出すときは，呼び出し規約の違いがあるため，呼び出し規約を調整するスタブを経由して呼び出す．スタブでは，引数の積直しや例外の伝搬処理などを行う．また，コンパイル済みコードからインタプリタのメソッドまたは，C ライブラリの関数を呼び出す場合にも，別のスタブがあり，スタブを経由して呼び出す．

### 3.2 例 外 処 理

CRuby インタプリタでは，例外処理の実現に表引き法と setjmp 法を併用しており，表引き法は CRuby インタプリタ内の例外を処理し，setjmp 法は CRuby インタプリタと C ライブラリの間で例外を処理するために利用している．我々のコンパイル済みコードでも同様に，コンパイル済みコードでは高速に動作する表引き法，コンパイル済みコードと C ライブラリまたは，CRuby インタプリタとの間では，発生した例外を `_longjmp()` で引き渡す．

`_longjmp()` で引き渡される例外を受け取るためには `_setjmp()` を実行する必要があるが，



© 2011 Information Processing Society of Japan

_setjmp() の実行コストが大きいことを考えると，同時に_setjmp() がパフォーマンスに与える悪影響を軽減する対策も必要になる．次章ではこの対策について考察する．

## 4. 提案技法

本章では_setjmp() のオーバヘッドを軽減する方法について次の 3 つの観点から考察する．考察の過程で，それぞれにどのような対策を施しうるかを示すが，我々が採用した対策は，(2)，(3) 向けの技法のみである．

(1)　_setjmp() を実施するタイミング
(2)　_setjmp() とその呼び出し元にある冗長な処理
(3)　冗長な_setjmp() 呼び出し

### 4.1 _setjmp() を実施するタイミング

C ライブラリで例外を投げる手段として_longjmp() を使う場合に考えなければならないことは，_longjmp() を行うと，C スタックのうち，_longjmp() を行った C フレームから_setjmp() した C フレームの次の C フレームまでが破壊されてしまうことである．このため，コンパイル済みコードが C フレームに実行コンテキストを配置する場合，**図 4** に示すように C ライブラリを呼び出す直前のフレームで_setjmp() するなどして，_longjmp() によって C フレームが破壊されないようにする必要がある．図 4 はインタプリタからコンパイル済みコードを呼び出し，さらにコンパイル済みコードから C ライブラリを呼び出した時点での C スタックの様子を表している．図 4 中に記載した I2C スタブ，C2N スタブはそれぞれ，インタプリタからコンパイル済みコードを呼び出す際に使うスタブと，コンパイル済みコードから C ライブラリを呼び出す際に使うスタブである．

図 4 のようにコンパイル済みコードから C ライブラリを呼び出す手前で_setjmp() を実施することの問題点は，_setjmp() を呼び出す頻度がある程度高くなってしまうことである．この問題を回避する手段として，**図 5** のように，C スタックを 2 本用意する方法が考えられる．すなわち，2 本の C スタックの一方をコンパイル済みコード専用にすれば，_longjmp() によるコンパイル済みコードの実行コンテキストの破壊を回避できる．図 5 は，図 4 と同じく，インタプリタからコンパイル済みコードを呼び出し，そこからさらに C ライブラリを呼び出した時点における C スタックの様子を表しているが，ここで C ライブラリで例外を投げるために_longjmp() をしてもコンパイル済みコードの C フレームは壊れない．このため，インタプリタの例外ハンドラを改造して，例外をつかまえた際に動的コンパイル済みコードを呼び出し中かチェックし，呼び出し中だったらコンパイル済みコードから実行を再開するようにすれば，コンパイル済みコードから C ライブラリを呼び出すたびに_setjmp() をしなくても C ライブラリが投げる例外を処理可能になる．

もっとも図 5 の方法が図 4 の方法より優れているとは限らない．その理由を次に示す．

- 図 5 の方法を使うためには I2C スタブや C2N スタブで C スタックを切り替える必要があり，そこからオーバヘッドが生じる．
- C2N スタブで callee save レジスタを保存しなければならないことは図 4 の方法と変わりない．ここで callee save レジスタとは呼び出し規約において内容を保存する責任が呼び出し先にあると定めているレジスタのことである．図 5 では C2N スタブがコンパ

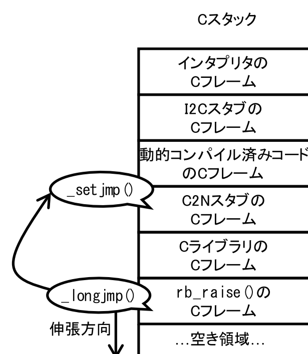



**図 4**　1 本の C スタックを使用した例外処理の実装
Fig. 4　An implementaion of exception handling using a single C stack.

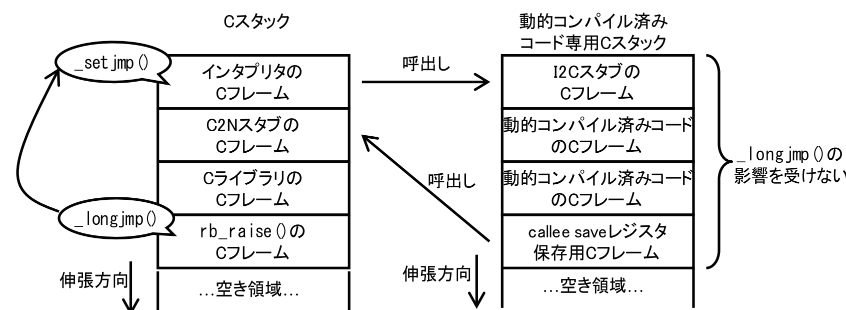



**図 5**　2 本の C スタックを使用した実装
Fig. 5　An implementaion of exception handling using two C stacks.


© 2011 Information Processing Society of Japan

イル済みコードの呼び出し規約における callee save レジスタを動的コンパイル済み専用 C スタックに退避し，C ライブラリ呼び出しから戻る際に復帰できるようにしている（例外が発生した状態，していない状態のどちらで戻った際にも）．なお，図 4 の方法では callee save レジスタの保存を _setjmp() の一環として実施できる．

我々が図 4，図 5 の 2 つの方法のうち，図 5 のスタックを切り替える処理を試作し，後述する提案技法を加えた図 4 の方法と比較するために実験をした．実験は各方法の _setjmp() に相当する処理のみを 10 億回実行した時間を計測した．計測の結果，図 4 の方法のほうが，実行時間が 33%少ないことが分かった．そこで我々は図 5 の技法の採用はとりやめた．

### 4.2　_setjmp() とその呼び出し元にある冗長な処理

setjmp 法の準備に当たる処理（図 1 の 2 行目）は，次に示す 3 つの処理からなる．

（1）　_setjmp() の呼び出し処理
（2）　_setjmp() の呼び出し元における比較処理と分岐処理
（3）　_setjmp() のレジスタの値を退避する処理

これらの個々の処理向けのオーバヘッドを軽減する技法を順次示す．_setjmp() の呼び出し処理については，_setjmp() を呼び出し元にインライン展開すれば，呼び出しのオーバヘッドを回避できる．展開後の呼び出し元のコードを**図 6** に示す．図 6 のコードは Linux/IA32 環境において _setjmp() を展開したもので，1〜15 行目が展開した _setjmp() に，16〜19 行目が _setjmp() の呼び出し元における比較処理と分岐処理にあたる．

_setjmp() の呼び出し元における比較処理と分岐処理については，_setjmp() で保存する _longjmp() のジャンプ先を変更すれば不要になり，削除できる．具体的には，_setjmp() では _longjmp() のジャンプ先として _setjmp() の戻りアドレスを保存するが（図 6 の 10〜13 行目），ここでジャンプ先を例外ハンドラに変更すれば，例外が発生した際には直接例外ハンドラにジャンプするようになるので，_setjmp() の呼び出し元で返戻値を比較したり（14 行目），比較結果に応じて分岐したり（15〜17 行目）する必要がなくなり，これらの処理を削除可能になる．また，比較判定用レジスタの初期化処理（1 行目）も削除可能になる．

最後に，_setjmp() のレジスタの値を退避する処理については，C 言語用の _setjmp() を使うと呼び出し規約の違いから無駄が出るのでカスタマイズにより無駄を省くことができる．図 6 では，ebx，edi，esi，eip，esp，ebp の 6 本のレジスタの値を退避している．退避する理由は，Linux/IA32 環境における C 言語の呼び出し規約によって，関数呼び出しから戻る際にこれらのレジスタ内容を復元するように求められているためだが，我々はコンパイル済みコード内の呼び出し規約を定める際に，esp，eip 以外のレジスタの内容の復元を求めないことにした．そのため，esp，eip 以外の 4 本のレジスタの値を退避する命令を削減できる．また，15 行目のシグナルマスクの保存判定用のフィールドを初期化する処理では，シグナルマスクの保存は行わないため，_setjmp() 用バッファの確保時に行うことで削減できる．よって，図 6 のコードでは，3，4，5，14，15 行目が削減できる．これらの技法を行ったコードを**図 7** に示す．図 7 と図 6 のコードを比較すると，これらの技法によって命令数が 18 命令から 9 命令に減っていることが分かる．

```
1:   xorl %eax, %eax
2:   movl JMPBUF, %edx
3:   movl %ebx, (JB_BX*4)(%edx) ; ebx 退避
4:   movl %esi, (JB_SI*4)(%edx) ; esi 退避
5:   movl %edi, (JB_DI*4)(%edx) ; edi 退避
6:   leal (%esp), %ecx
7:   xor %gs:0x18, %ecx ; マングル処理
8:   rol $0x9, %ecx ; マングル処理
9:   movl %ecx, (JB_SP*4)(%edx)
10:  movl, (16 行目のアドレス), %ecx ; eip 退避
11:  xor %gs:0x18, %ecx ; マングル処理
12:  rol $0x9, %ecx ; マングル処理
13:  movl %ecx, (JB_PC*4)(%edx) ; esp 退避
14:  movl %ebp, (JB_BP*4)(%edx) ; ebp 退避
15:  movl %eax, JB_SIZE(%edx) ; シグナルマスク判定フィールド初期化
16:  testl %eax, %eax
17:  jz not_exception
18:  jmp exception_handler
19:  not_exception:
```

**図 6**　インライン展開を適用した setjmp 法の準備のコード
Fig. 6　_setjmp() call site after inlining _setjmp().

なお，setjmp 法の準備処理を図 7 のコード列にするまでの過程で適用した技法のうち，次に示すものは Linux/IA32 環境に固有で他の環境に適用できるとは限らない．

- 呼び出し規約の違いを利用したレジスタの値を退避する命令の省略
- シグナルマスクが保存されているか否かを示すフィールドを初期化する処理の移動

シグナルマスクが保存されているか否かを示すフィールドを初期化する処理，つまり図 7 の 13 行目の命令が必要になるのは glibc 2.10.1 において，_longjmp() が siglongjmp() と実装を共有しており，siglongjmp() がシグナルマスクの復帰を行うか否かを決めるためにこのフィールドを参照するためだが，もし _longjmp() が実装を siglongjmp() と共有し


© 2011 Information Processing Society of Japan

```
1:   movl JMPBUF, %edx
2:   leal (%esp), %ecx
3:   xor %gs:0x18, %ecx ; マングル処理
4:   rol $0x9, %ecx ; マングル処理
5:   movl %ecx, (JB_SP*4)(%edx)
6:   movl (例外ハンドラのアドレス), %ecx ; eip 退避
7:   xor %gs:0x18, %ecx ; マングル処理
8:   rol $0x9, %ecx ; マングル処理
9:   movl %ecx, (JB_PC*4)(%edx) ; esp 退避
```

**図 7**　提案技法を適用した setjmp 法の準備処理のコード
Fig. 7　_setjmp() call site after proposed techniques.

ていなければ，この初期化処理自体が不要になる．

なお，我々は今のところ例外処理の高速化のために CRuby を修正してはいないが，シグナルマスクが保存されているか否かを示すフィールドの初期化処理を省略し，それでも CRuby の rb_raise() が正常に動作するように，rb_raise() が呼び出す_longjmp() を rb_raise() に特化したものにして，そこではシグナルマスクが保存されているか否か確認しないことにすることも可能である．さらに，CRuby が呼び出す_setjmp() と_longjmp() の双方を改変するなら，図 7 の 3，4，7，8 行目にあるポインタのマングル処理も省略可能になる．マングル処理は jmpbuf 中に保存するポインタのセキュリティ確保のためにあるが，我々の実装では例外処理のための jmpbuf を C スタックを配置しており，我々が C スタック上に保存している他のポインタ（返戻先番地など）にマングル処理を施していないことを考慮すると，例外処理のための jmpbuf に保存するポインタに限ってマングル処理を施すことの意味は少ないといえる．しかし，_longjmp() の呼び出し回数は少ないことや，メモリの操作命令に比べ，レジスタの操作命令は実行時間が少ないことから，あまり大きな効果が見込めないと考えられるため，今回は採用を見送った．

### 4.3　冗長な_setjmp() 呼び出し

4.1 節で述べたように，1 つの C スタックを利用する場合，_longjmp() の実行時に C スタックが破壊されるのを防ぐ対策が必要になる．具体的な対策の 1 つは，C ライブラリを呼び出すたびに，_setjmp() を実行することだが，それでは_setjmp() を呼び出す頻度が高くなってしまう．この問題を解決する手段として，本節では実行プロファイルを使って_setjmp() を呼び出す回数を減らす技法を提案する．

提案技法が対象とするのは_setjmp() を複数回呼び出しうるメソッドである．_setjmp() を呼び出すケースは，C ライブラリの呼び出し前に呼び出すケースと_setjmp() を呼び出すスタブを呼び出すケースである．このようなメソッドについて，_setjmp() の実施箇所をメソッドの先頭に移し退避されたバッファを使いまわすことで，_setjmp() の呼び出しをメソッド呼び出し 1 回につき 1 回にする．この技法は個々の C ライブラリ呼び出し前で_setjmp() を実行した場合に_setjmp() が保存する内容が異なると使えない．保存する内容は，例外ハンドラのアドレスとスタックポインタ esp の値のみだが，ここで esp の値はメソッドの実行中に変化しないので 1 度保存しておけば問題ない．

保存すべき例外ハンドラのアドレスが呼び出し元ごとに異なる場合には，Ruby のプログラムカウンタから例外ハンドラを求めてジャンプする汎用的な例外ハンドラのアドレスを保存しておく．なお，Ruby のプログラムカウンタは現在の我々の動的コンパイラの実装がバックトレースの生成や脱最適化のサポートなどを目的としてメソッド呼び出しのたびに CRuby の制御フレームスタックに保存しているので，そこから取得できる⋆1．

なお，この方法には，_setjmp() を行わないメソッドに適用すると_setjmp() の実行回数が逆に増加しまうことがあるという問題がある．この問題を解決するため，プロファイラを用いて最適化の有効または無効にするべきかの情報を取得し，その情報をもとに最適化すべきメソッドでは最適化し，逆に最適化すべきでないメソッドでは，最適化を適用しないようにした．具体的には，メソッドの実行時に_setjmp() の実行回数が平均 1 回以上であれば最適化を適用し，逆に実行回数の平均が 1 未満であれば最適化を適用しないこととした．

## 5. 評　　価

本章では，4 章で提案した技法が実行速度に与える影響を評価した結果を示す．5.1 節では，setjmp 法における準備の実行速度に与える影響について評価した結果を示す．5.2 節では，Ruby アプリケーションの実行速度に与える影響について評価した結果について述べる．評価に利用した計算機およびソフトウェアは，**表 1** のとおりである．

### 5.1　setjmp 法の準備処理

4.2 節で述べた技法が setjmp 法の準備処理の実行速度に与える影響を評価した結果を**図 8**

⋆1 Ruby のプログラムカウンタは，C スタックに保存されているコンパイル済みコードへの戻り番地から算出できる．そこで我々は将来的には，メソッド呼び出しのたびに Ruby のプログラムカウンタを保存する処理を省略する予定だが，この省略を行ったとしても提案技法は利用可能である．ただしその場合 C スタック上の戻り番地から Ruby のプログラムカウンタを計算することになるので，_longjmp() の際に C スタック上の戻り番地を破壊してしまうことがないよう，_setjmp() の際に保存する C スタックポインタの値を調整する必要が生じる．


© 2011 Information Processing Society of Japan

**表 1**　評価環境
Table 1　Evaluation environment.

| | |
|---|---|
| CPU | Intel Xeon 3060（2.40 GHz） |
| Memory | 2GB |
| OS | Ubuntu-9.10（32bit） |
| Linux Kernel | 2.6.31.14 |
| Compiler | gcc-4.4.1 |

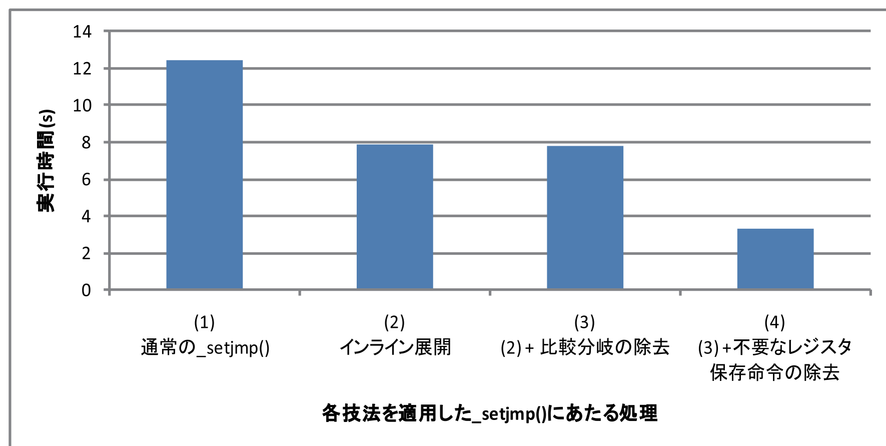



**図 8**　setjmp 法の準備処理の実行結果
Fig. 8　An evaluation of techniques for `_setjmp()` call site.

に示す．評価は，次のパターンの setjmp 法の準備処理のコードのみを 10 億回実行する際にかかる時間を計測することで行った．

(1)　提案技法未適応（図 2 の 2 行目にあたる処理）
(2)　(1) にインライン展開したコード（図 6）
(3)　(2) から比較と分岐命令を除去したコード
(4)　(3) から不要なレジスタの値の退避命令を除去したコード（図 7）

図 8 を見ると，命令数の減少により高速化していることが分かる．オーバヘッドが比較的大きくなる比較分岐命令については，あまり速度の向上がみられていないが，測定プログラムが単純なプログラムであるため CPU の分岐予測による影響だと思われる．すべての技法を適用した (4) では，(1) と比べ，実行時間を 73%削減できていていることが分かった．

**表 2**　今回使用したベンチマークプログラム一覧
Table 2　Bench mark program list used this time.

| 種別 | ベンチマークプログラム | 略称 |
|---|---|---|
| macro-benchmarks | bm_cal.rb | cal |
| | bm_gzip.rb | gzip |
| | bm_hilbert_matrix.rb | hilbert |
| | bm_list.rb | list |
| | bm_mpart.rb | mpart |
| | bm_norvig_spelling.rb | norvig |
| | bm_observ.rb | observ |
| | bm_parse_log.rb | parse |
| | bm_pi.rb | pi |
| | bm_rcs.rb | rcs |
| | bm_sudoku.rb | sudoku |
| rails | bm_substruct_create_variations.rb | create |
| | bm_substruct_load_products.rb | load |
| | bm_substruct_request_root_30x.rb | request |
| | bm_substruct_request_root_same_session_30x.rb | same |

### 5.2　Ruby アプリケーション

本節では，4 章で述べた提案技法が Ruby アプリケーションの実行時間に与える影響を，Ruby Benchmark Suite [19] を使って，評価した結果を示す．Ruby Benchmark Suite とは，Ruby のベンチマークプログラムセットである．今回の計測に使用したベンチマークプログラムの一覧を**表 2** に示す．本論文では，個々のベンチマーク項目を表 2 の略称で記述する．評価にあたっては，提案技法を適用しない場合に比べ，次の技法を適用するとどれだけ実行時間に影響があるかを調査した．

評価するケースは，次の 5 つのパターンである．

(1)　インライン展開（図 6）
(2)　(1) から比較と分岐命令の除去
(3)　(2) から不要なレジスタの値の退避命令の除去（図 7）
(4)　(3) + `_setjmp()` の呼び出しをメソッド呼び出し 1 回につき 1 回にする技法
(5)　プロファイラを用いて (4) の適応箇所を最適化する技法

調査にあたっては，各プログラムを 100 回ずつ実行し，平均時間を計測した．なお，計測プログラムは，事前に十分に実行されすべてのコードがコンパイル済みコードであるものとした．また，プロファイラを用いた最適化についても，同様にプロファイル情報を収集し


© 2011 Information Processing Society of Japan

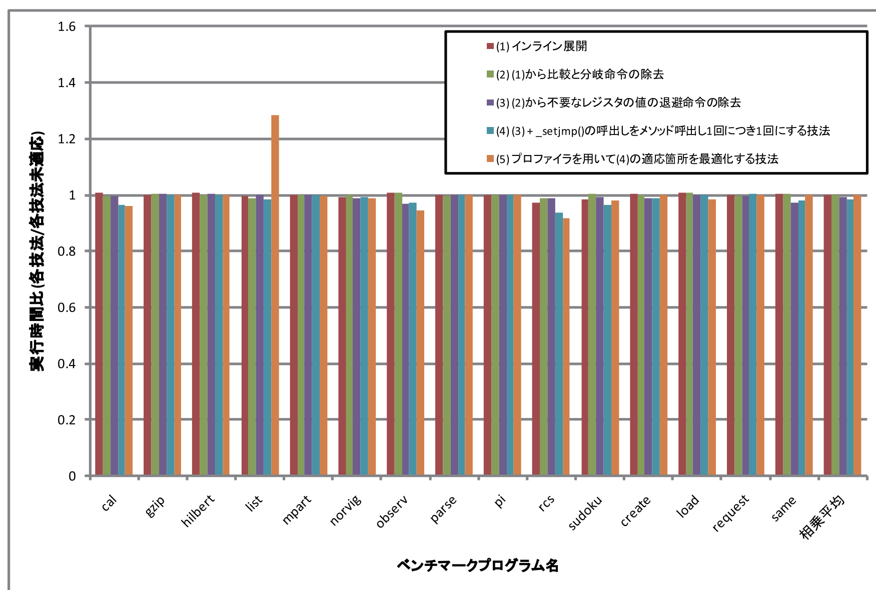



**図 9**　提案技法が Ruby Benchmark Suite の実行性能にあたえる影響
Fig. 9　Proposed techniques effect measured using Ruby Benchmark Suite.

た後，再コンパイルしたものとした．評価結果を**図 9** に示す．

図 9 の計測結果を見ると，個々のベンチマーク項目の結果の中には提案技法を適用するとかえって実行時間が長くなるケースがままあるものの，相乗平均の項目を見るとより多くの技法を適用した方が基本的には性能を改善できる傾向があることが分かる．ベンチマークの実行時間が一定せず，1%程度のぶれが珍しくないことから，相乗平均で性能を改善できているので提案技法はおおむね有効ではないかと考える．

なお，(5) に限っては相乗平均でも (4) と比較して性能劣化しているが，これは list の測定結果による影響が大きく，list を除いた相乗平均では (4) より実行時間を 0.2%短縮できていた．list については，GC の実行時間が (5) だけ 43%増加していたことが分かっており，それが list で (5) の性能が劣化した原因であると考える．list も含めたベンチマークプログラムすべての相乗平均では，(4) のケースが最も速く，実行時間を 1.4%削減できることが分かった．

(5) と (4) で適用した技法の効果を調査するために，冗長な呼び出しの除去を適用しない場合に比べ，適用するとどれだけ呼び出し回数が減少するか調査した結果を**図 10** に示す．

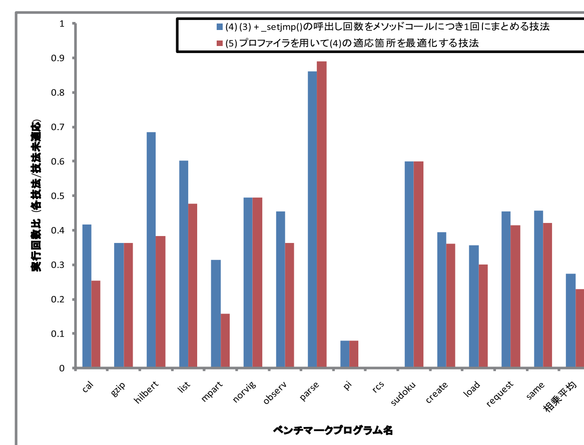



**図 10**　提案技法が Ruby Benchmark Suite の`_setjmp` の呼び出し回数にあたえる影響
Fig. 10　Proposed techniques effect measured call frequency of setjmp of Ruby Benchmark Suite.

図 10 をみるとすべての項目で呼び出し回数を削減できていることが分かり，また，プロファイルを用いる方がより多く削減できていることが分かる．`parse` の項目に限ってはプロファイルを用いない方が多く削減できているが，調査の結果，その原因はプロファイルによる予測のミスであることが分かった．

## 6. 結　　論

本論文では，CRuby 向け動的コンパイラにおける例外処理を実現するうえで問題となる点について指摘し，その問題点を改善する方法を提案，実装した．例外処理は，表引き法と setjmp 法を用いて実装したが，setjmp 法を用いることで発生する性能劣化が問題となるため，`_setjmp()` による性能劣化を軽減する 2 つの技法を開発した．開発した技法の 1 つは`_setjmp()` の呼び出し元と呼び出し先にある命令のうち，C ライブラリが投げる例外の捕捉に不要なものを除外する技法であり，もう 1 つは，`_setjmp()` で初期化したバッファを使いまわすことで，`_setjmp()` が必要になる箇所を減らす技法である．これらの技法により，setjmp 法の準備処理の実行時間を 73%削減でき，Ruby Benchmark Suite で評価したところ，Ruby アプリケーションの実行時間を相乗平均で 1.4%削減できることが分かった．


© 2011 Information Processing Society of Japan

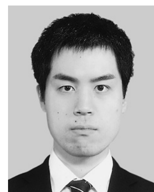
## 参 考 文 献


1) 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).
2) まつもとゆきひろほか：オブジェクト指向スクリプト言語 Ruby. http://www.ruby-lang.org/
3) Schilling, J.L.: Optimizing away C++ exception handling, *SIGPLAN Not.*, Vol.33, No.8, pp.40–47 (1998).
4) Andreas, K. and Mark, P.: Monitors and Exceptions: How to implement Java efficiently, *Proc. ACM 1998 Workshop on Java for High-Performance Network Computing*, pp.15–24, ACM (1998).
5) Paleczny, M., Vick, C. and Click, C.: The Java Hotspot(tm) Server Compiler, *Proc. USENIX Java Virtual Machine Research and Technology Symposium*, pp.1–12 (2001).
6) 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).
7) Dean, J., DeFouw, G., Grove, D., Litvinov, V. and Chambers, C.: Vortex: An optimizing compiler for object-oriented languages, *SIGPLAN Not.*, Vol.31, No.10, pp.83–100 (1996).
8) Krall, A. and Grafl, R.: CACAO — A 64-bit JavaVM just-in-time compiler, *Concurrency: Practice and Experience*, Vol.9, No.11, pp.1017–1030 (1998).
9) 千葉雄司：Java2C トランスレータにおける例外処理の実現，情報処理学会論文誌：プログラミング，Vol.42, No.11, pp.14–24 (2001).
10) 千葉雄司：組込み機器向け Java2C トランスレータにおける 2 返戻値法を使った例外処理の実現，情報処理学会論文誌：プログラミング，Vol.43, No.1, pp.85–96 (2002).
11) Jung, D.-H., Park, J., Bae, S.-H., Lee, J. and Moon, S.-M.: Efficient exception handling in Java bytecode-to-C ahead-of-time compiler for embedded systems, *Comput. Lang. Syst. Struct.*, Vol.34, No.4, pp.170–183 (2008).
12) Cameron, D., Faust, P., Lenkov, D. and Mehta, M.: A portable implementation of C++ exception handling, *Proc. C++ Conference*, pp.225–243 (1992).
13) de Dinechin, C.: C++ Exception Handling, *IEEE Concurrency*, Vol.8, No.4, pp.72–79 (2000).
14) 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).
15) de Dinechin, C.: C++ exception handling for IA-64, *Proc. 1st conference on Industrial Experiences with Systems Software* (*WIESS'00*), Berkeley, CA, USA, p.8, USENIX Association (2000).
16) 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, p.21 (2009).
17) Cierniak, M., Lueh, G.-Y. and Stichnoth, J.M.: Practicing JUDO: Java under dynamic optimizations, *PLDI '00: Proc. ACM SIGPLAN 2000 conference on Programming language design and implementation*, New York, NY, USA, pp.13–26, ACM (2000).
18) Ogasawara, T., Komatsu, H. and Nakatani, T.: EDO: Exception-directed optimization in java, *ACM Trans. Program. Lang. Syst.*, Vol.28, No.1, pp.70–105 (2006).
19) Antonio, C.: Ruby Benchmark Suite. http://antoniocangiano.com/2008/06/01/help-me-create-the-ruby-benchmark-suite/
20) Rails Core Team: Ruby on Rails. http://rubyonrails.org/


(平成 22 年 7 月 5 日受付)
(平成 22 年 11 月 17 日採録)

**村田　俊哉**

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



**石井　直也**

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



© 2011 Information Processing Society of Japan

**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，文部科学省 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（S3FIRE）」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，特定非営利活動法人日本セキュリティ監査協会会長，ホワイトスペース推進会議会長，ブロードバンドワイヤレスフォーラム会長，ACM 日本支部長等．専門は計算機科学および情報セキュリティ．日本学術会議連携会員．



© 2011 Information Processing Society of Japan