# 2014 年度 Ruby アソシエーション助成金公募 最終報告書

# RuJIT: A Trace-based Just In Time Compiler for CRuby

2014 年 2 月 27 日

井出真広

## 1. 概要

本稿ではトレース方式 JIT コンパイラ RuJIT について，Ruby アソシエーション助成金のもと行われた開発成果と今後の課題についてまとめる．RuJIT は CRuby 処理系向け Just In Time コンパイラであり，本プロジェクトの目標を JIT コンパイラによる Ruby スクリプト実行の高速化として開発を行った．RuJIT は期間中に新たなコンパイラ用中間表現，トレース選択器の実装，そして既知のコンパイラ最適化手法の実装を行い，実行速度の改善を行った

## 2. 背景と目的

Ruby 言語は，高い生産性やプログラミングのしやすさから Web アプリケーション開発を中心に広く利用されている．Ruby 言語は他のスクリプト言語と同様に動的型付けやメタプログラミングの機能を備え，加えて数多くのフレームワーク，ライブラリを有する言語である．

Ruby 言語の参照実装である CRuby ではその評価器は高速化のため YARV(Yet Another RubyVM)をバイトコードインタプリタとして導入し Ruby 言語の処理性能を大幅に向上させてきた．しかし Ruby 言語の処理性能は Lua 言語の LuaJIT など高速な処理系と比較すると大きな性能差がある．

本プロジェクトでは，スクリプト言語の JIT コンパイル手法として用いられている手法の１つであるトレース方式 JIT コンパイル手法を Ruby 処理系上に構築し，CRuby 処理系の処理性能向上を目指す．本プロジェクトでは CRuby 処理系向け JIT コンパイラ，RuJIT の構築をゴールとし．以下 4 点について開発を行う．

1. Ruby プログラムから Hotspot となるコードの抽出機構の開発
2. Ruby プログラムを表現可能，かつ最適化が容易な命令セット(RuJIT 命令セット)と Ruby プログラムを RuJIT 命令セットへ変換するコンパイラの設計・開発
3. Ruby 言語向けコード最適化器の設計・開発
4. RuJIT 命令セットから機械語へ変換するコンパイラの設計・開発

## 3. 実装

我々は上記で述べた項目について開発を行い，項目 1，2 について開発を完了された．しかし，項目 3，4 については当初の計画の半分程度の達成率となった．以下では，それぞれの項目について開発内容を述べる．

### 3.1 HotSpot 抽出機

HotSpot 抽出機は Ruby プログラム中から頻繁に実行されるパスを抽出しコンパイル対象として選択する機構である．その抽出方法は，文献[1]を基本としインタプリ

タ中にプロファイルコードを埋め込み，そこから得られたプロファイル結果をもとにコンパイル対象の抽出を行っていた．しかしインタプリタに埋め込まれたプロファイルコードにより，特に RuJIT が JIT コンパイル対象にできていないコード，例えば再帰関数を含むアプリケーションの実行速度が YARV に比べ 20%程度低下するといった実行オーバヘッドの高いという問題があった．

RuJIT の導入による低速化を防ぐため我々は，実行時にプロファイル用コードを取り外し可能な実装を行い，プロファイルが不要な場合（JIT コンパイル対象にならない場合）の速度低下は平均 10%程度に低減した．

### 3.2 RuJIT 命令セットの設計

### 3.3 コード最適化器

コード最適化器は，コンパイル時に得られた仮定を元に Ruby バイトコード列から得られた RuJIT の中間表現の最適化を行う．RuJIT ではコンパイラ最適化として以下の最適化器が実装されている．

- Constant Propagation
- Common Sub-expression Elimination
- Dead Code Elimination
- Dead Store Elimination
- 限定的な Stack allocation

期間中に行ったベンチマークにより生成されたコードについて最適化の余地があることがプロファイリング結果から確認することができた．とくにコンパイル時に収集した仮定が実行時に正しいかを検査するガード命令は，実行オーバヘッドに加えて，生成されるコードの control flow graph を複雑化し，最適化を阻害することが確認できた．我々はガード命令について，既知のコンパイラ最適化としてループ外へのコード巻き上げ最適化を導入し，また冗長なガード命令の除去を行いマイクロベンチマークにて 1.2 倍から 2 倍の速度向上を確認できた．

### 3.4 コード生成器

最後に，コード生成器について現在の実装では実装コスト低減のためシステムに付属している C コンパイラ（例えば OSX では clang，Linux では gcc）を利用している．現時点で数行程度の小さなアプリケーションにおいては RuJIT を適応するとコード生成器の実行オーバヘッドによって速度向上が見られない，もしくは低速化することを確認できた．我々はコード生成のオーバヘッド低減のため，コード生成に用いるヘッダファイル，ランタイムコードをプリコンパイル済みヘッダに変換しオーバヘッド低減を行った．ただし，この改良を加えた場合でも，そのオーバヘッドが無視できない場合も存在するため今後高速なコード生成器を導入する必要があると考えられる．

## 4. 評価

本節では，マイクロベンチマークによる評価を行い，HotSpot 抽出のオーバヘッド，そしてコード生成器と本プロジェクトであらたに導入した最適化機による効果を示す．なお実験環境は OSX10.0，CPU Intel Corei5 1.3GHz，メモリ 8GB のものを使用

した.
まずRuJIT導入によるオーバヘッドを計測するためJITコンパイル機能を停止させたRuJITとCRuby処理系との実行時間を比較し，図1に示す．縦軸はCRuby処理系の実行時間を1とした時のRuJITでの実行時間を示しており，この数値が高いほどRuJIT導入によるオーバヘッドが高いことを示す.

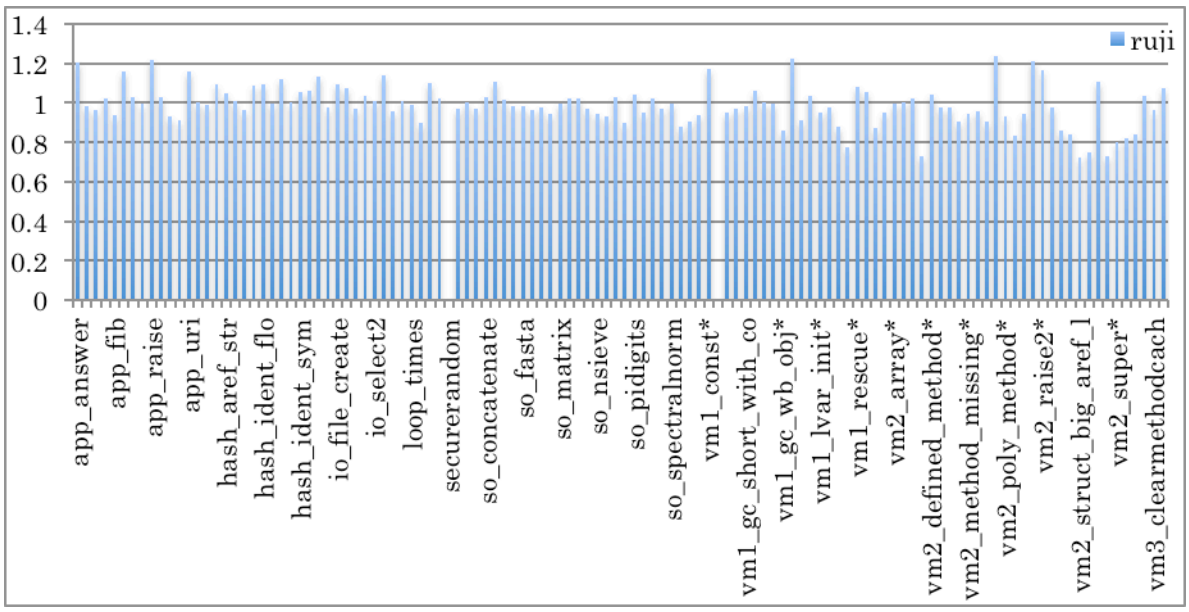
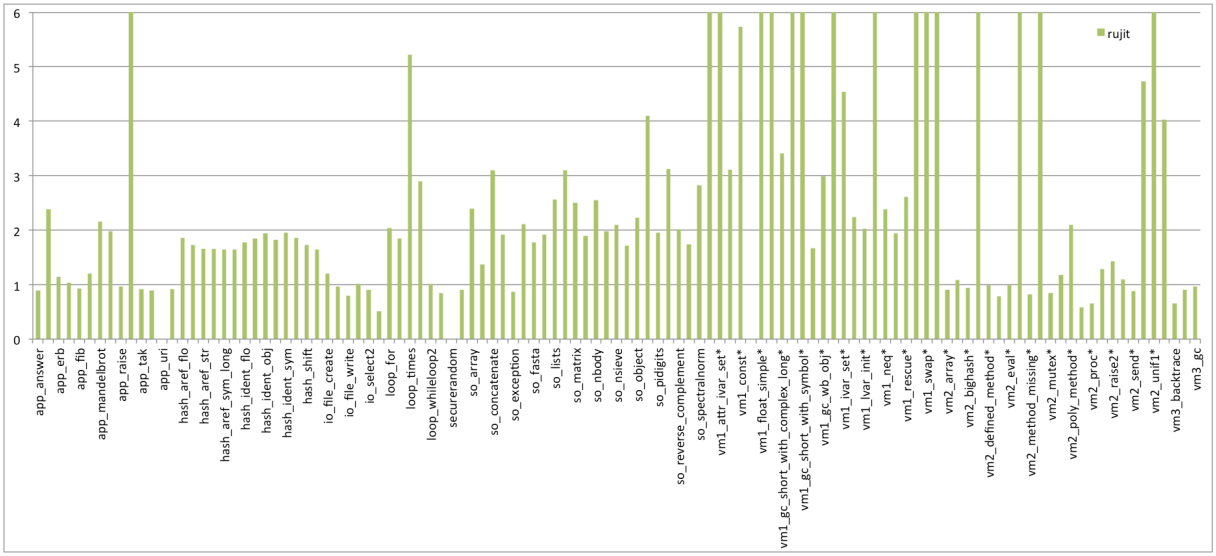



図 1 抽出機のオーバヘッドの評価

図1より多くの場合でHotSpot抽出機をインタプリタに導入した際のオーバヘッドは0~10%程度に収まっていることを確認した．なお，抽出機導入によるオーバヘッドが20%を超えるベンチマークについて，JIT コンパイルにより，そのオーバヘッドは無視できるものと考えられる.
次にRuJITのパフォーマンスを評価するため同様のベンチマークをRuJITとCRuby処理系にて行い実行時間の比較を行った（図 2）．縦軸は CRuby 処理系の実行時間を 1 とした時のRuJITでの実行時間を示しており，この数値が高いほどRuJITが高速であ



ることを示している．なお RuJIT の実行時間はトレース抽出にかかった時間，コンパイルにかかった時間を含む．
図 2 より，RuJIT のコンパイル対象を含む多くのベンチマークでは 2~5 倍の高速化を達成していることが確認できる．一方で io_file_read や app_fibo や app_lc_fizzbuzz といったベンチマークでは，実行時間の改善はほとんど見られなかった．これはファイル操作や再帰関数の実行，クロージャの実行といった，トレースに含むことができなかった処理を多く含むためと考えられる．

## 5. まとめと今後の課題

RuJIT は CRuby 処理系の高速化を目標にした JIT コンパイラである．マイクロベンチマークの結果多くの場合で高速化を達成した．
今後の課題として RuJIT の完成度を早急に向上させ Rails や RDoc など大規模アプリケーションにおける評価を行うことがあげられる．また一方で前述のとおり一部のアプリケーションで低速化もしくは YARV と同程度の実行速度のプログラムも確認している．これらプログラムも含めたさらなる高速化を実現するため LLVM を用いた高速なコード生成やコード最適化の導入について取り組む予定である．
　なお期間中に設計，実装を行った HotSpot 抽出機，RuJIT 命令セット，コンパイラ基盤についてはドキュメントを作成し，以下 URL から参照可能である．

http://imasahiro.hatenablog.com/entry/2014/12/03/000037
http://imasahiro.hatenablog.com/entry/2014/12/03/233000
http://imasahiro.hatenablog.com/entry/2014/12/05/055822
http://imasahiro.hatenablog.com/entry/2014/12/05/000000
http://imasahiro.hatenablog.com/entry/2014/12/06/233429
http://imasahiro.hatenablog.com/entry/2014/12/08/081323
http://imasahiro.hatenablog.com/entry/2014/12/09/074731