

Ruby仮想マシンにおけるスタックの自動拡張

杉山 敬太^{1,a)} 笹田 耕一^{2,b)} テュールスト マーティン ヤコブ^{3,c)}

受付日 2018年2月4日, 採録日 2018年5月12日

概要: 近年, より需要の高まりつつある並行処理では, メモリの効率的な使用が大きな課題となっている. 本研究では, プログラミング言語 Ruby のリファレンス実装において, スレッドごとのスタックの動的拡張を実現した. バグを含まない実装のための開発手段を提案, 利用するとともに, 複数の実装のメモリ使用量と実行速度を比較し評価した. 実験の結果, 最も良かった実装では, 実行速度の低下を平均 6%に抑えつつ, 実用的なモデルでメモリ使用量を 30%程度削減することに成功した.

キーワード: スタック自動拡張, 仮想マシン, プログラミング言語 Ruby, 開発手段

Dynamic Extension of the Ruby Virtual Machine Stack

KEITA SUGIYAMA^{1,a)} KOICHI SASADA^{2,b)} MARTIN J. DÜRST^{3,c)}

Received: February 4, 2018, Accepted: May 12, 2018

Abstract: For concurrent processing, recently increasing in popularity, efficient memory use is important. This research succeeded in implementing dynamic extension of per-thread stacks for the reference implementation of the programming language Ruby. We propose two development methods crucial for achieving a production-quality implementation, and comparatively evaluate the memory use and execution speed of various implementations. Our best implementation on Linux achieves around 30% memory reduction for a realistic model, at the cost of an average speed decrease of only 6%.

Keywords: dynamic stack extension, virtual machine, Ruby programming language, development methods

1. 序論

近年, 並行処理や並列処理の需要の高まりを受け, プログラミング言語の設計面から実装面まで高度な並列処理への対応が問われている. 日本を中心に開発されているオブジェクト指向言語 Ruby は, ウェブアプリケーションや簡単なスクリプトなどのために世界中で幅広く使われてい

る. しかし, 並行・並列処理には課題がある.

そこで最近, 新しい並行実行モデルの導入 [1] から, 並行処理機能を追加するために必要な実装の修正まで, Ruby の並行・並列処理をプログラマにとって使いやすく効率のよいものにする取り組みが行われている. 本論文では, この一環としてマルチスレッドプログラムでのメモリ使用量削減を行う.

Ruby のリファレンス実装かつ言語開発の中心でもある Matz's Ruby Interpreter (以後 MRI) では, スレッドごとのスタックサイズは現時点で固定である. 本研究では, 仮想マシンのスタックの動的な拡張の実装により, スレッドごとに確保するスタック領域の無駄を減らすことで, メモリ使用量を抑えることに成功した.

スタックを新たな領域にコピーすることで動的に拡張する場合, スタック内を参照するポインタへの漏れのない対応とその確認のためのテスト手法は大きな課題である. そ

¹ 青山学院大学大学院理工学研究科理工学専攻知能情報コース
Intelligence and Information Course, Graduate School of Science and Engineering, Aoyama Gakuin University, Sagami-hara, Kanagawa 252-5258, Japan

² クックパッド株式会社
Cookpad Inc., Shibuya, Tokyo 150-6012, Japan

³ 青山学院大学理工学部情報テクノロジー学科
Department of Integrated Information Technology, College of Science and Engineering, Aoyama Gakuin University, Sagami-hara, Kanagawa 252-5258, Japan

a) sugiyama@sw.it.aoyama.ac.jp

b) ko1@cookpad.com

c) duerst@it.aoyama.ac.jp

の対策のため、2つの開発手段を提案し、有効性を確認する。効率の良いスタック拡張を実現するため、複数の手法を提案し、メモリ使用量の削減や実行速度に対する影響を評価する。

本論文の残りの構成は以下のとおりである。2章では本研究の対象であるMRIの仮想マシンスタックについて説明する。3章では、バグを含まない実装のために不可欠であった開発手段を紹介する。4章では、スタックの動的な拡張の設計について、5章ではその実装について説明する。6章では、メモリ使用量の削減や性能への影響を評価する。7章では、他言語の実装に用いられている仮想マシンスタックの拡張と比較し、8章に、まとめと今後の課題を述べる。

2. プログラミング言語 Ruby と Ruby 仮想マシンのスタック

本章では、プログラミング言語 Ruby とその実装の概要を述べる。加えて、Ruby の実装に用いられる仮想マシンのスタックについて説明する。

2.1 プログラミング言語 Ruby

Ruby [2] は、松本行弘によって開発が始められた。1995年に公開され、多くのプログラマから支持を集めている。Ruby のリファレンス実装は Matz's Ruby Interpreter (以後MRI) または CRuby と呼ばれる。MRI は C 言語で記述され、C 言語で拡張ライブラリを書くことができる。

MRI は仮想マシンとして笹田らが開発した YARV (Yet Another Ruby VM) [3] を使用する。YARV は Ruby 1.9 から Ruby に導入された [4]。本論文では、以後この仮想マシンのことを Ruby 仮想マシンと呼ぶ。

2.2 Ruby 仮想マシンのスタック

Ruby 仮想マシンのスタック領域は図 1 に示すように、2つのスタックからなる。Shaughnessy [5] に合わせて、スタック領域の始点からメモリアドレスが増加する方向に伸びるスタックを内部スタック、反対方向に伸びるスタックをコールスタックと呼ぶ。スタック領域へのポインタとそのサイズは実行コンテキスト構造体に記録され、必要に応じ構造体を通じて参照される。実行コンテキスト構造体とはスレッドごとに用意される実行情報を記録するための構造体である。

2.2.1 コールスタック

コールスタックはコントロールフレームと呼ばれるデータ構造を記録する。コントロールフレームは一定サイズの構造体である。現在のコントロールフレームへのポインタは実行コンテキスト構造体に記録される。各コントロールフレームはメソッドやブロックの呼び出しと対応して、フレームのスタックポインタ、環境ポインタなどを記録する。スタックポインタはフレームごとの内部スタックのスタック

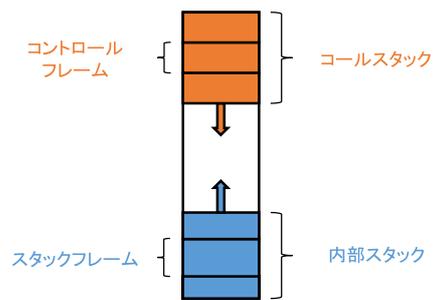


図 1 Ruby 仮想マシンスタックの構造

Fig. 1 Structure of Ruby virtual machine stack.

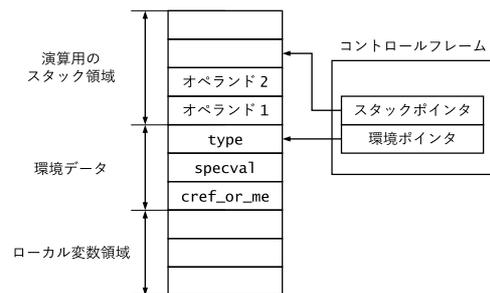


図 2 スタックフレームの構造

Fig. 2 Structure of stack frame.

クトップを指すポインタである。環境ポインタはフレームのローカル変数や環境データへのアクセスに使用される。コントロールフレームには、他にフレームのプログラムカウンタや self が記録されるが、これらは本研究とは関係しないため説明は省略する。

2.2.2 内部スタック

内部スタックには、各コントロールフレームと対応して図 2 のような構造を持つスタックフレームが構成される。各スタックフレームは、ローカル変数と環境データを記録する。環境データには、そのフレームが C 言語で定義されたメソッドか Ruby メソッドかどうかなどのフレームに関する情報や、ブロックのレキシカルスコープを実装するための情報が記録される。フレームの上部はスタック領域として用いられ、命令列を実行する際に、命令の被演算子や戻り値を記録するために使われる。

2.3 仮想マシンスタックの動作

本節では、仮想マシンスタックの動作を説明する。本研究に関連する内容を中心に、ローカル変数へのアクセス、メソッド呼び出し、ブロックの起動について説明する。

2.3.1 ローカル変数へのアクセス

Ruby のローカル変数のスコープはクラスやメソッドに限定されている。しかし、無名関数やクロージャを実現するブロックからは周りのスコープのローカル変数にアクセスできる。この実装のため、フレームは 2 種類に分類される。メソッドのようにローカル変数のスコープがそのフレームで完結するフレームを「ローカルである」と表現し、

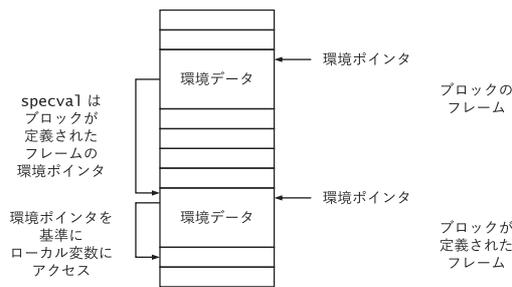


図 3 外側のスコープのローカル変数へのアクセス
Fig. 3 Access to local variables in outer scope.

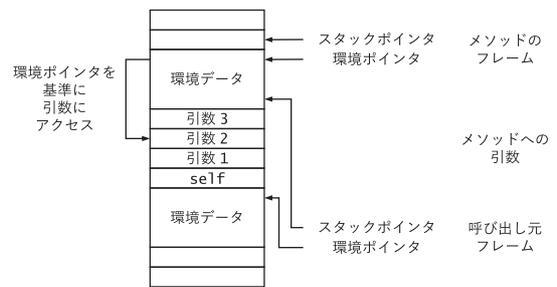


図 4 メソッドへの引数の配置
Fig. 4 Layout of arguments passed to a method.

ブロックのような、ローカル変数のスコープが以前のフレームに及ぶフレームは「ローカルでない」と表現する。

ローカル変数には、現在のコントロールフレームの環境ポインタを基準にインデックスとレベルでアクセスする。レベルはローカルでないフレームからフレーム外のローカル変数にアクセスする場合に、対象が何フレーム前に記録されているかを表す。環境データの1つである `specval` は、`type` に記録されているフレームの種類によって、2つの用途に兼用される。メソッドなどの場合に、`specval` にはそのメソッドに渡されたブロックが参照する環境が `block_handler` として記録される。これはコントロールフレームの一部または手続きオブジェクトへの参照である。手続きオブジェクトの構造をコントロールフレームの一部と同じにすることで区別なくアクセスできる。ブロックの場合には、`specval` には前のフレーム（すなわち、外のスコープ）の環境ポインタが記録されている。ブロック内から外のスコープのローカル変数にアクセスする場合の概要を図 3 に示す。

2.3.2 メソッド呼び出し

メソッド呼び出しでは、まず内部スタック内のスタックフレーム上部の領域に図 4 のようにメソッドへの引数を配置する。次に、メソッドのフレームとして、コントロールフレームとスタックフレームを確保し、実行コンテキスト構造体のコントロールフレームへのポインタを更新する。スタックフレームは配置された引数のうえに確保されるので引数はメソッドのフレームからローカル変数と同様にアクセスできる。メソッドはローカルなフレームなので、`specval` には現在のコントロールフレームへの参照を、`block_handler` として記録する。

2.3.3 ブロックの起動

ブロックを起動する際は、まず、メソッド呼び出し時と同様にスタックにブロックへの引数をスタックに積み、コントロールフレームとスタックフレームを確保する。このとき、`specval` にはブロックが定義されたフレームの環境ポインタが記録される必要がある。そのためにまず、呼び出し元のフレームの `specval` をローカルなフレームまでたどり、`block_handler` を得る。`block_handler` はコント

ロールフレームまたはそれと同じ構造のオブジェクトへの参照なので、メンバに環境ポインタを持つ。この環境ポインタをブロックのフレームの `specval` に記録する。

2.3.4 スタックオーバーフローの発生

MRI ではスタックオーバーフローの発生をマクロで確認する。確認はプッシュ操作毎ではなく、次の処理に必要なスタック領域が事前に計算可能な場合に限定されている。実際に確認が行われる箇所は実装上 8 カ所存在する。

例として、メソッド呼び出し時のフレームのプッシュがある。ここでは、メソッドの実行に十分な領域があるかどうかを判断し、そうでない場合スタックオーバーフローを呼び出す。各メソッドにおいてローカル変数や被演算子の記録に必要な内部スタック上の領域は、命令列のコンパイル時に算出され、記録される。そのほかにスタックオーバーフローの確認が行われる箇所としては、メソッドやブロックへの引数をスタック上に配置する処理がある。

2.4 仮想マシンのスタックサイズ

スレッドごとの仮想マシンのスタックサイズは仮想マシンの初期化時に決定される。サイズは環境変数 `RUBY_THREAD_VM_STACK_SIZE` で指定できる。標準のスタックサイズは 128 K ワードで、64 bit CPU の環境では 1 MB になる。

Ruby にはスレッドと別に Fiber [6] というユーザレベルスレッドが用意されている。Fiber のスタックの大きさは別の環境変数で指定できる。スタック関連を含め実装中の多くの箇所ですレッドと区別なく扱われるため、本論文ではスレッドと同等と考える。

3. バグを含まない実装のための開発手段

実装の詳細については 4 章で説明するが、スタックの拡張はスタックを新たな領域に移動することで行う。そのため、拡張により移動するスタック内へのポインタが問題になる。実行コンテキスト構造体や仮想マシンスタック内に記録されるポインタは、スタックの拡張時に修正できるため問題にならない。しかし、これら以外の箇所から参照しているポインタは、修正はもとより、発見すら困難である。

このような参照は特に MRI のような、国内外の多数の開発者で開発されているオープンソースプロジェクトの場合に顕著に発生する。たとえば仮想マシンの開発者はデバッガに関係したコードを把握しているとは限らないし、そのコードの責任者も明確でないかもしれない。

そこで、安定した実装を実現するための開発手段を2つ提案し、利用する。1つ目の手段は早期にポインタの修正漏れを発見することを可能にする。2つ目の手段はあらゆる条件でのスタック拡張の検証を可能にする。

3.1 移動前のスタックへのアクセス禁止

拡張前のスタックへのポインタが使用されると、多くの場合エラーが発生するまでに時間がかかり、エラーの原因の特定が困難になる。そこで、問題のあるアクセスでただちにエラーを発生させるために、移動前のスタックを単に free で解放せずに mprotect [7] 関数で PROT_NONE を指定してアクセスを禁止した。これにより、スタック拡張の前に保存されたポインタを拡張後に使用すると、セグメンテーション違反がただちに発生し、エラーの原因が特定しやすくなる。mprotect は OS 依存の関数であり、この手段は仮想メモリ空間を大量に消費するが、開発段階でのテストとしてのみ必要なため最終的に完成した実装には影響しない。

3.2 すべての箇所でのスタックの移動

一般のプログラムにおいて、スタックの拡張が起こることはまれである。また、スタック拡張をあらゆる条件下でテストするプログラムの作成も難しい。そこで、拡張の起こりうるすべての箇所疑似的なスタック拡張を行う手段を導入する。疑似的な拡張とは、スタックのサイズを変化させずに移動することを指す。前述の手段と組み合わせることで、実装中でスタックが拡張される可能性のある処理を越えてスタック内へのポインタが使用される箇所を早期に見出せる。これにより、スタックがいつ拡張されても正常に動作するような実装を目指す。

4. スタック拡張の手法

本研究では、仮想マシンのスタック拡張を2つの手法で実装する。本章では、それぞれの手法について説明する。

4.1 手法1：両スタックのコピー

スタック拡張の1つ目の手法（以後手法1）ではスタックの構造をそのまま再利用する。スタック領域には、元の実装と同様に、上端と下端にコールスタックと内部スタックをそれぞれ配置する。スタックオーバーフローが発生する場合には、図5に示すように拡張する。使用中のスタックより大きなメモリ領域を新たに確保し、そこへ使用中のスタックをコピーする。

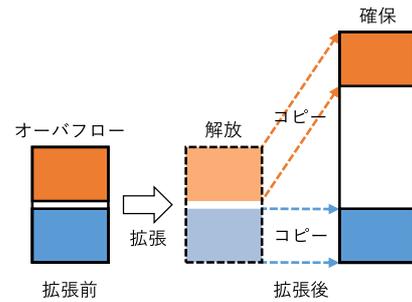


図5 手法1: スタック領域全体の再確保
Fig. 5 Method 1: Overall stack relocation.

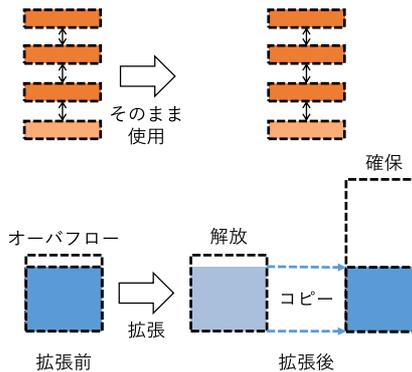


図6 手法2: コントロールフレームのリスト化
Fig. 6 Method 2: Using a linked list for control frames.

この手法では、両方のスタックが移動するため、両スタックへのアクセス方法を変更する必要がある。また、それぞれのスタックは別々に移動するため、スタックへのポインタを区別して扱う必要がある。

4.2 手法2：コールスタックのリスト化

スタック拡張の2つ目の手法（以後手法2）では、コールスタックをコントロールフレームの連結リストとして実現し、内部スタックのみを図6に示すように拡張する。

コントロールフレームのサイズは一定であるため、リストでの実装は容易である。コントロールフレームは必要に応じて新たに確保しリストに追加するため、内部スタックの拡張時には処理が発生しない。スタック外からは主にコントロールフレームを利用してスタックにアクセスするため、多くの箇所ではスタックへの参照方法を変更する必要がない。

内部スタックは元の構造のまま利用する。内部スタックをスタックフレームの連結リストとして実装しても、引数の配置処理で発生するスタックオーバーフローではスタックフレームを拡張する必要があるため、内部スタックへのポインタを元の実装のまま使用することはできない。そのため、内部スタックは元の構造のまま連続領域に確保する。

5. スタック拡張の実装

本章では、スタック拡張処理の詳細を説明する。特にポ

インタによるスタックへの参照に対する対応方法について述べる。

5.1 スタックの拡張処理

スタックオーバフローの検知処理を変更し、スタックオーバフロー発生の可能性がある場合、スタック拡張処理を呼び出す。手法1でのスタックオーバフローの検知処理は元の実装と同様に、コールスタックの頂点と内部スタックの頂点との間に、処理の継続に十分な領域があるかどうかを確認する。手法2では内部スタックの頂点とスタック領域上端の間を確認する。

スタック拡張処理ではまず、拡張後のスタックサイズを決定し、新たな領域にスタック領域を移動する。次に、スタックおよび実行コンテキスト構造体内に記録された、元のスタック内へのポインタを新しいスタックを指すように修正する。最後に、不要な古いスタック領域を解放する。

5.2 スタックサイズの決定

スタックの拡張では拡張ごとにスタックサイズを2倍にする。ただし、無限再帰のプログラムを発見できるようにするためにスタックには最大サイズを設定する。それを越えるサイズが必要な場合には、元の実装のスタックオーバフロー時と同様に例外を発生させる。スタックサイズは最大サイズを超えないように決定し、2倍より大きい領域が必要な場合は必要な分だけ拡張する。

従来の仮想マシンのスタックサイズを示す環境変数を変更し、スタックの最大サイズを示すようにした。また、スタックの初期サイズのために新しい環境変数 `RUBY_THREAD_VM_STACK_INITIAL_SIZE` を導入する。

5.3 スタック内の相互参照の修正

スタック領域内のポインタは実行コンテキスト内のポインタと違い、必ずしもスタックへのポインタとは限らない。そのため、その値や環境データ内の情報を元に判別して修正する必要がある。各コントロールフレームのスタックポインタはつねに内部スタックへのポインタであるが、環境ポインタはスタック内へのポインタとは限らない。MRIではブロックをProcオブジェクトに変換するときなどに、スタックフレームをスタックからヒープ内にコピーする。このとき環境ポインタはヒープ内の対応する箇所を指すように変更されるため、環境ポインタは内部スタック内またはスタック外へのポインタになる。

環境データ内の `specval` はフレームがローカルかどうかで修正方法を変更する。ローカルなフレームでは環境ポインタとしての値が記録されているため、内部スタック内を指すかどうかを判定する必要がある。ローカルでないフレームでは、`block_handler` としての値が記録されている。そこで、MRIに用意されている関数を用いて

`block_handler` のタイプを判定する。この結果がRubyオブジェクトでない場合は、ポインタの値がコールスタック内かを判定し修正する。

5.4 スタック外からの参照への対応

MRIでは実装においてスタック内へのポインタを一部の構造体やCレベルのローカル変数に保存して使用している。たとえば、Rubyレベルでの例外の発生時にはポインタとして保存されたコントロールフレームまでコントロールフレームを巻き戻す。また、メソッドやブロックへの引数を内部スタック上に配置するとき、引数部分へのポインタを利用する。したがって、すでに解放されたスタックへのポインタの再利用を防ぐため、状況や実行速度を考慮しながら「間接参照の利用」、「メソッドへの引数のコピー」、「実行コンテキスト構造体の利用」の3つの方法で対応する。

本研究における実装では、スタックへのポインタの扱いについて、元となるMRIの構造を大きく変更せずに、問題になるポインタそれぞれへの局所的な変更にとどめる。スタック拡張によって問題になるポインタを使用する箇所は3章に述べた2つの手段を利用して発見できる。仮想マシンの内部構造は拡張ライブラリやほとんどのクラスからは隠されているため、変更が必要になるのは主に仮想マシンの処理に関する箇所である。

5.4.1 間接参照の利用

拡張処理前に保存したスタックへのポインタを拡張処理後に再び利用する箇所では、保存時にスタックの始点からのオフセットを保存し、利用時に各スタックの始点とオフセットから参照先のアドレスを計算するように変更する。実行コンテキスト構造体内のコールスタックおよび内部スタックの始点へのポインタは拡張時に修正されるので、それらのポインタを基準にオフセットを計算する。

手法1で問題になるコールスタックへのポインタには、コントロールフレームへのポインタとして用いられるものと、クロージャの実装で `block_handler` として用いられるものがある。コントロールフレームへのポインタについては前後のコントロールフレームへのアクセスを容易にするため、コントロールフレーム構造体単位でオフセットを計算する。コントロールフレームへのポインタはメソッド呼び出しに関する多くの関数に引数として渡されるほか、命令ディスパッチ処理において、レジスタに保存して用いられる。`block_handler` は必ずしもコールスタックへのポインタとは限らないため、ローカル変数や関数の引数として利用する場合には `block_handler` がオフセットに変換されたものかどうかを判別するための変数を合わせて利用する。

両手法で問題になる内部スタックへのポインタが利用されている箇所は、メソッドやブロックへの引数をスタックに配置する処理である。このような処理では、内部スタック

ク上の処理中の箇所に対するポインタを一時的にローカル変数に保存して使用する。一方で、命令ディスパッチ処理においては、スタックポインタや環境ポインタとしてコントロールフレームのメンバを参照するため問題にならない。

5.4.2 メソッドへの引数のコピー

MRI では Ruby プログラムから C 言語で定義された Ruby メソッドを呼び出す場合、呼び出すメソッドに引数の配列を渡す。このとき、引数の配列として内部スタック上の引数部分へのポインタを用いる。呼び出されたメソッド中で別のメソッドを呼び出したときにスタックの拡張が発生する可能性があるため、その後引数配列を参照するべきではない。

呼び出されるメソッドは仮想マシンの内部構造には関知しないので、メソッド中でこのポインタへの対応を行うことはできない。そのため、メソッドへの引数を `alloca` 関数により確保した領域へコピーしてわたす。

5.4.3 実行コンテキスト構造体の利用

メソッドへの引数を内部スタック上に配置する処理では、スタック内へのポインタが繰り返し用いられる。オフセットとポインタの変換を繰り返すのは非効率であるため、この処理で用いるポインタを実行コンテキスト構造体内に記録するように変更する。これにより、スタックの拡張時にそれらのポインタを修正できるため、ポインタとオフセット間で繰り返し変換する必要がなくなる。

6. 評価実験

本章では本研究による実装を元の実装と比較して評価する。以後、本研究による手法 1 と手法 2 の 2 つの実装と元の実装をまとめて 3 実装と表現する。3 実装のメモリ使用量について、スレッド数による変化と、初期スタックサイズによる変化を評価する。さらに、3 実装の実行速度を比較する。

6.1 評価環境

実験は Intel x86_64 CPU Core i7-6500U 2.50 GHz、メモリ 16 GB の Gentoo Linux 4.12.5 上で行った。処理系のコンパイルには GCC (Gentoo 6.4.0 p1.1) 6.4.0 を使用した。比較には、本研究による実装の元とした ruby 2.5.0dev (trunk 60436) (以下 trunk) を用いた。

6.2 互換性の評価

3 実装それぞれについて MRI のユニットテストを実行し、本研究による実装が元の実装との互換性を十分に保っていることを確かめた。本研究による 2 つの実装へのテストは 3 章で説明した開発機能を有効にした場合と、無効にした場合のそれぞれで実行した。

17,429 個のテスト、2,232,108 個のアサーションが実行され、元の実装ではすべてのテストに成功した。開発機能

```
(0..ARGV[0].to_i).map do |i|
  Thread.new { sleep 100 }
end
```

図 7 メモリ使用量の評価のためのプログラム
Fig. 7 Program for evaluation of memory usage.

表 1 スレッド数を変化させたときのメモリ使用量 (初期スタックサイズ 1KB)

Table 1 Memory usage per number of threads.

スレッド数	最大物理メモリ使用量 [MB]		
	手法 1	手法 2	trunk
0	6.2	6.1	6.1
2,000	34.6	34.9	55.8
4,000	63.5	64.1	105.8
6,000	92.4	93.4	155.9
8,000	121.2	122.5	206.1
10,000	150.1	152.0	256.5

を無効にして行ったテストではすべてのテストに成功した。開発機能を用いた場合では、複数のテストで制限時間の超過やメモリリークが検出された。また、仮想メモリサイズを制限したプロセスを利用するテストでは、スタックの確保に失敗したことがあった。メモリリークの原因は、仮想マシンスタックを解放しなかったことである。テストの制限時間はあくまで無限ループや外部要因による大幅な遅延の感知のためのものであるため、制限時間を開発機能による遅延を考慮して適切に延長した場合、テストは成功した。この結果から、本研究による 2 つの実装が元の実装と十分な互換性を保つことが確認できた。

また、本研究の前段階として前年に同様の実装を試みたが、今回取り入れた開発手段がなかったため実装が困難であり、部分的な実装にとどまった。このことから、本論文における実装者の個人的な感想として、3 章で提案した開発手段は非常に有効であったと考える。

6.3 メモリ使用量の削減

新実装によるメモリ使用量の削減の度合を検証するために、3 実装それぞれで図 7 に示すプログラムを実行した。GNU time 1.7 (shell 組込の time 命令とは別) で最大物理メモリ使用量を計測した。プログラムは 3 回ずつ実行し、最も良い結果を採用した。

6.3.1 生成するスレッドの数とメモリ使用量の関係

生成するスレッド数を 0 から 10,000 まで変化させてメモリ使用量を計測した。手法 1 と手法 2 については、初期スタックサイズを 1KB に設定した。元の実装は標準のスタックサイズである 1MB を使用した。得られた結果を表 1 に示す。

スレッド数とメモリ使用量はおおむね線形な関係にある。スレッド 1 つあたりのメモリ増加量が元の実装での

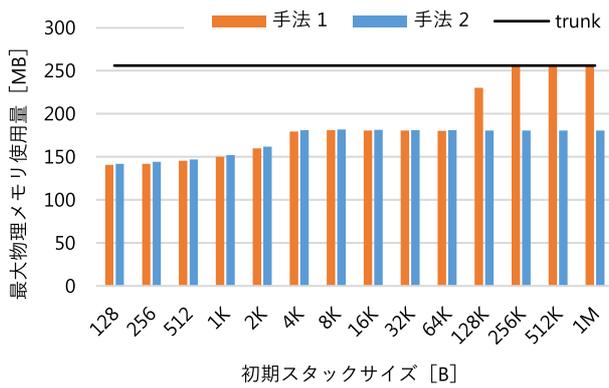


図 8 初期スタックサイズを変化させたときのメモリ使用量 (スレッド数 10,000)

Fig. 8 Memory usage for different initial stack sizes (10,000 threads).

25.7KB から手法 1 で 14.8KB, 手法 2 で 15.0KB に減少した。したがって, スレッド 1 つあたりのメモリ使用量は手法 1 で 42.4%, 手法 2 で 41.7%削減されたといえる。

また, 元の実装で 10,000 個のスレッドを生成した場合, 仮想マシンスタックには 10,000 MB の領域が使用されることになるが, 実際に消費された物理メモリは 257 MBにとどまる。これは, Linux ではアロケーションによって仮想メモリ領域を割り当てて, 実際に使用されるまで物理メモリにマッピングしない [8] ためと考えられる。よって, 元の実装のメモリ使用量と新実装による削減は OS によって変化する可能性がある。

6.3.2 初期スタックサイズとの関係

初期スタックサイズを 128B から 1MB まで変化させてメモリ使用量を計測した。生成するスレッド数は 10,000 に固定した。実験により得られたメモリ使用量を図 8 に示す。

全体的に, 初期スタックサイズの縮小がメモリ使用量の削減に有効だと分かる。初期スタックサイズによるメモリ使用量の変化が一定でない理由は, 割り当てられた領域の大きさにより, OS が物理メモリをマッピングする際の振る舞いが変わるためと考えられる。4KB はページサイズと同等で, 128KB 以上はメモリ管理のデータ構造の深さが変わることが想像できる。

また, 初期スタックサイズが元の実装のスタックサイズである 1MB と近いとき, 手法 2 のメモリ使用量は元の実装よりも少なくなった。これは, 元の実装や手法 1 と違って, 手法 2 ではスタック領域の両端ではなく, 下端のみがマッピングされたためだと考えられる。

6.4 より現実的なモデルでの評価

前節では, 単に sleep するスレッドを多数生成するプログラムを作成し, 評価に用いた。本来は, より実社会で使われているものと近いプログラムによる実験が好ましい

```
def recurse(n)
  if n > 0
    recurse(n - 1)
  else
    sleep
  end
end
```

図 9 スタックの深さを制御する再帰メソッド
Fig. 9 Recursive method to control stack depth.

が, たとえばネットワーク処理を含む実験などでは不確定な要素が多く, 再現性を保って評価実験を行うことが困難である。そこで, 現実に近いがコントロール可能な実験設定として, スレッドのスタックの深さの最大値に関する分布のモデルを 2 つ作成し, メモリ使用量の変化を比較し, 評価した。片方のモデルではスタックの深さは線形に分布する。これを線形モデルと呼ぶ。もう片方のモデルではスレッドの数が半減するにつれ, スタックの深さが倍増する。これを指数モデルと呼ぶ。

スタックの深さを制御するため, 図 9 に示す指定した回数再帰するメソッドを利用する。このメソッドは, 1 回再帰するごとにスタック上にコントロールフレーム 1 個 (6 ワード, 64 bit CPU の場合 48 B) とスタックフレーム 1 個 (5 ワード, 64 bit CPU の場合 40 B) を追加する。

評価に用いたプログラムではスレッドを 1,024 個生成し, それぞれのスレッドで再帰が終了することを確実にするため各スレッド生成後に 0.1 秒スリープさせた。それにより, 一定数のスレッドとそのスタックの最大の深さの分布の場合の最大メモリ使用量を評価した。

線形モデルでは i 番目のスレッド t_i における再帰回数が $10i$ 回になるように実行した。スレッド数は 1,024 個であるため再帰回数は 10 回から 10,240 回となり, 全体での平均的な再帰回数は 5,125 回である。64 bit CPU においてはスタック全体の深さは手法 1 と trunk では最低 880 B, 最大 901,120 B で, 手法 2 では最低で最低 400 B, 最大 409,600 B となる。結果, スタックの拡張が起こらない大きさからスタックオーバーフローが発生する直前まで様々なスタックの深さでの実験となる。

指数モデルでは 512 スレッドで 10 回再帰, 256 スレッドで 20 回再帰, 128 スレッドで 40 回再帰のように再帰回数を決定して実行した。最大の再帰回数は 1 スレッドでの 10,240 回, 平均の再帰回数は 60 回である。実際の応用では浅いスタックのスレッドが頻繁に生成され, 深いスタックのスレッドがまれにしか使われないことを反映したモデルである。

線形モデルおよび指数モデルでの結果を表 2 に表す。結果から, 線形モデルでは, スタック拡張を実装した場合にメモリ使用量が増加したことが分かる。まず, 手法 1 での増加の原因は, スタック拡張処理時に一時的に 2 つの領域

表 2 スタック拡張をともなうプログラムでのメモリ使用量

Table 2 Memory usage for programs with stack extension.

再帰回数の分布	最大物理メモリ使用量 [MB]		
	手法 1	手法 2	trunk
線形モデル	375.5	550.4	348.0
指数モデル	24.3	24.6	34.9

を必要とするためであると考えられる。また、手法 2 ではコントロールフレームを動的に確保するため、コントロールフレーム数が多くなったときにメモリの無駄が増加すると考えられる。

指数モデルでは、本研究による 2 つの実装について、元の実装に比べおよそ 30% のメモリ使用量の削減が確認された。これはスタックの浅いスレッドが多かったため線形モデルで問題になった要素の影響が小さく、全体でのメモリ使用量が削減されたためだと考えられる。このことから、実用的な応用において特によく見られるスタックの浅いスレッドの多い場合に、本研究で提案するスタックの自動拡張がメモリ使用量の削減に有効であると考えられる。

6.5 実行時間

MRI のベンチマークにより実行時間を計測した。3 実装それぞれについて 3 回ずつ実行し、最も短かった実行時間を結果として用いる。

6.5.1 実装の変更による実行速度への影響

幅広く用いられる実装において新規機能の導入や一部の応用のために新実装を行う場合、従来どおりの使用（今回はシングルスレッド）への影響が懸念される。そのために初期スタックサイズを元の実装と同様の 1 MB に設定し、スタック拡張が発生しない状態で実装の変更による実行速度への影響を評価した。

評価には Ruby 付属のベンチマークを使用した。様々な機能や応用にわたって計 197 のベンチマークを行ったが、紙面の都合上、一部の結果のみ記載する。それぞれの実行結果について、ベンチマーク名のプレフィックスごとにグラフを作成した。グラフ中ではベンチマーク名のプレフィックスを省略している。縦軸は元の実装との実行時間比であり、値が大きいほど実行速度が遅くなったことを示す。プレフィックスが app のベンチマークの結果を図 10 に示す。これは様々な応用処理についてのベンチマークである。また項目名が vm, vm1 で始まるベンチマークは、様々な基本処理について実行時間を計測するものである。結果をそれぞれ図 11, 図 12 に示す。項目名が vm で始まるベンチマークは主にスレッドに関する項目である。

ベンチマーク全体での実行時間比の最大値は、手法 1 では 1.63, 手法 2 では 1.35 だった。また、全体での実行時間比の平均は手法 1 では 1.18, 手法 2 では 1.06 だった。

本研究による 2 つの実装では、元の実装と比較するとほ

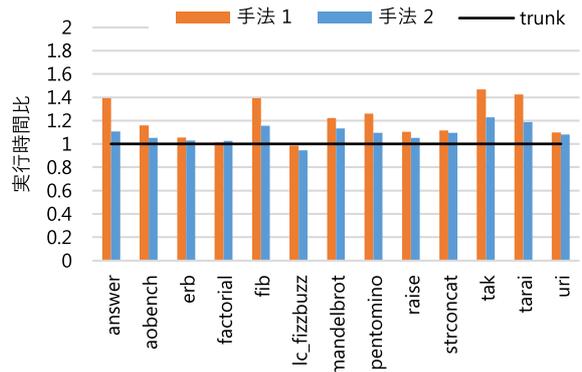


図 10 app 系のベンチマークの実行時間比

Fig. 10 app-related relative benchmark times.

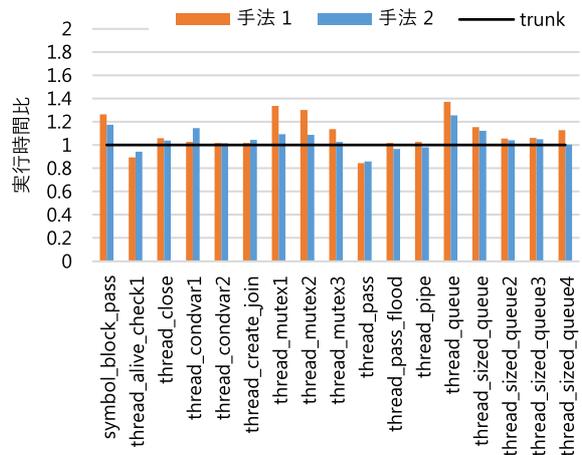


図 11 vm 系のベンチマークの実行時間比

Fig. 11 vm-related relative benchmark times.

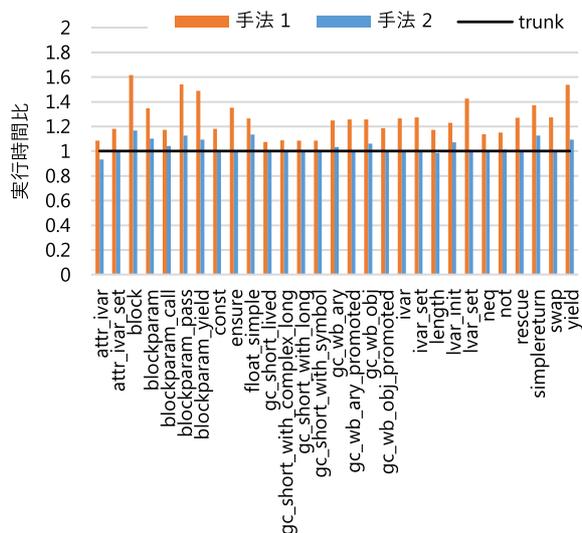


図 12 vm1 系のベンチマークの実行時間比

Fig. 12 vm1-related relative benchmark times.

とんどのベンチマークで実行速度が低下した。主要な要因として、スタックへの参照方法を一部の箇所を変更したことや、メソッドやブロックに対する引数を別の領域にコピーすることが考えられる。

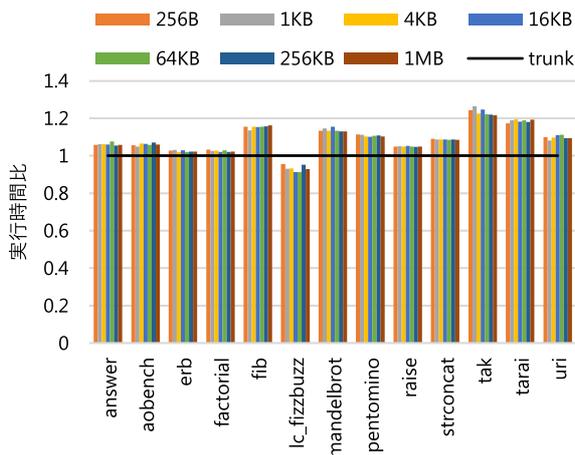


図 13 スタックサイズによるベンチマークの実行時間比の変化 (手法 2)

Fig. 13 Variation of relative benchmark time for different stack sizes (Method 2).

また、手法 2 と比較して手法 1 の実行速度が遅いことが確認された。図 11 から、blockparam_pass や block などのブロックに関するベンチマークで特に差が大きいことが分かる。この原因は、手法 1 でブロックに関連する処理で使用されるコールスタックへのポインタを一部の箇所の間接参照に変更したためだと考えられる。

6.5.2 初期スタックサイズによる実行速度への影響

初期スタックサイズを 256 B から 1 MB まで変化させたときの実行時間をベンチマークで計測した。手法 2 について得られた結果のうち、ベンチマーク名が app で始まるものの結果を図 13 に示す。

グラフから、手法 2 での実行速度には初期スタックサイズによって明らかな差が生じないことが分かる。スタック拡張ではスタックサイズを毎回 2 倍するため、拡張が起こる回数が少なく、スタックの拡張処理自体による実行速度への影響が観察されなかったものと思われる。このことから、初期スタックサイズは任意に小さくしても問題ないといえる。また、スタック拡張の実装による速度低下の主な原因は、スタックへのアクセス方法の変更にあると考えられる。

7. 他言語との比較

本章では、他言語や他実装における仮想マシンのスタック拡張について比較する。

7.1 mruby

mruby [9] は Ruby の実装の 1 つである。組み込み用途を想定し、Ruby の作者である松本らによって開発された。Ruby の ISO 規格 [10] の一部に準拠し、文法は Ruby 1.9 と互換性を保つ。MRI と同様に mruby の仮想マシンは主に 2 つのスタックを使用するが、これらは別の配列として

管理され、動的に拡張される。

MRI のコールスタックと対応するスタックの初期サイズは、標準でおおよそ 2 KB である。このスタックは、フレームをプッシュする際に十分な領域がないとき 2 倍に拡張される。もう一方のスタックは MRI における内部スタックと同様に用いられ、初期サイズは標準で 1 KB である。このスタックはメソッドへの引数をスタックに配置するために十分な領域がないときに拡張される。スタックサイズを毎回 2 倍にする方法と線形増加の方法を処理系のコンパイル時に選択できる。

スタックへのアクセスは実行コンテキスト構造体を通して行うため、スタックの拡張時にこれらのポインタを修正することで、新しいスタックを参照するように変更できる。

7.2 Go

Go [11] は Google で開発され、2009 年に公開されたプログラミング言語である [12]。goroutine と呼ばれる仕組みを用いて並行処理を容易に記述できる。コンパイラ言語であるため仮想マシンは存在しないが、runtime^{*1}がスタックを管理する。

goroutine ごとのスタックの初期サイズは最小で 2 KB であり、動的に拡張および縮小される。runtime は固定長 (2 KB, 4 KB, 8 KB, 16 KB) のスタックをプールで管理し、使用する。16 KB より大きなスタックが必要な場合には 2 倍のサイズのスタックを新たに確保する。goroutine と対応する構造体内に記録されたポインタについて、スタック内を指すものがあれば新たなスタックを指すように修正する。

7.3 Lua

現在の Lua [13] はブラジルの Pontifical Catholic University of Rio (PUC-Rio) で Ierusalimschy らによって開発された軽量で高速なインタプリタ言語である。他言語との連携が容易なため、1993 年に開発が始められて以来、多くのアプリケーションで使われている。

Lua の実装は、レジスタ型仮想マシンを用いる。MRI におけるコントロールフレームと対応するデータ構造は連結リストで管理され、仮想マシンの実装にはスタックが用いられる。スタックの初期サイズは標準で 480 B で、動的に 2 倍に拡張される。スタック内へのポインタが、スタック拡張を起こす可能性がある処理を越えて使用される場合は、オフセットに変換して保存する。

8. 結論

本研究では、プログラミング言語 Ruby の並行・並列処理への応用の拡大のために、仮想マシンにおけるスタック

*1 <https://golang.org/pkg/runtime/>

の自動拡張の実装し、評価した。安定した実装のために、「拡張前のスタック領域へのアクセス禁止」、および「すべてのオーバーフローチェックでのスタック移動」の2つの開発手段を提案し、利用した。これらの開発手段なしでは、現在の高い安定性を保つての実装はほぼ不可能であった。

拡張手法を2つ提案し、性能評価では手法2が手法1に実行速度の面で勝った。手法2ではコールスタックを連結リストで実装し、内部スタックのみ大きい領域へのコピーによって拡張する。Linuxにおいて、実用的なスタックサイズの分布のモデルの下で、メモリ使用量を30%程度削減し、ベンチマークの平均計算時間の増加を6%に抑えることに成功した。また、スタックの拡張処理自体による実行速度への影響はほとんどないことを確かめた。

本研究ではスタックの縮小を実装しなかったが、スタックの縮小によりさらなるメモリ使用量の削減が期待できる。たとえば、長く使用されるスレッドのスタックが一時的な処理のために拡張された後、ほとんど使用されずにメモリを占有しつづける可能性がある。

また、多数のスレッドを使用しないユーザにとっては、仮想マシンのスタックによるメモリ使用量は問題にならない。その場合に本実装が不利にならないようにするため、特にスタックの拡張が起きない場合、さらなる調整によって実行速度を改善することが望ましい。

MRIは多くの環境でコンパイル・実行可能である。そのため、実用に至るには他のOS、様々な設定下、様々な応用での評価と調整が必要になる。

謝辞 本研究は、2016年度に青山学院大学のDürst研究室で小池翔氏が行った研究に基づく。研究の進め方や論文の執筆について松原俊一氏と荘司慶行氏から多くの助言をいただいた。遠藤侑介氏には研究についての打ち合わせに参加していただき、助言を賜った。四氏に心から感謝する。

参考文献

- [1] 笹田耕一, 松本行弘: Ruby 3に向けた新しい並行実行モデルの提案, 情報処理学会論文誌プログラミング (PRO), Vol.10, No.3, p.16 (2017).
- [2] 松本行弘ほか: オブジェクト指向スクリプト言語 Ruby, 入手先 (<https://www.ruby-lang.org/ja/>) (参照 2018-04-23).
- [3] 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 (PRO), Vol.47, No.SIG2(PRO28), pp.57-73 (2006).
- [4] Thomas, D., Fowler, C. and Hunt, A., 松本行弘 (監訳), 田和 勝 (訳): プログラミング Ruby 1.9 言語編, chapter 25.6, オーム社 (2010).
- [5] Shaughnessy, P.: *Ruby Under a Microscope: An Illustrated Guide to Ruby Internals*, No Starch Press (2013).
- [6] 芝 哲史, 笹田耕一: Ruby1.9 での高速な Fiber の実装, 第 51 回プログラミング・シンポジウム予稿集, Vol.2010, pp.21-28 (2010).
- [7] Kerrisk, M., 千住治郎 (訳): Linux プログラミングインタフェース, オライリー・ジャパン (2012).
- [8] Gorman, M.: *Understanding the Linux Virtual Memory Manager*, Prentice Hall (2004).
- [9] 松本行弘ほか: mruby, 入手先 (<https://mruby.org>) (参照 2018-04-23).
- [10] International Organization for Standardization: *ISO/IEC 30170:2012 - Information technology - Programming languages - Ruby* (2012).
- [11] Donovan, A.A.A. and Kernighan, B.W.: *The Go Programming Language*, Addison-Wesley Professional (2015).
- [12] Pike, R.: Go at Google: Language Design in the Service of Software Engineering, available from (<https://talks.golang.org/2012/splash.article>)(accessed 2018-04-23).
- [13] Ierusalimsky, R., de Figueiredo, L.H. and Celes, W.: The Evolution of Lua, *Proc. 3rd ACM SIGPLAN Conference on History of Programming Languages*, pp.2-1-2-26, ACM (online), DOI: 10.1145/1238844.1238846 (2007).



杉山 敬太 (学生会員)

1995年生。2018年青山学院大学理工学部情報テクノロジー学科卒業。同大学大学院修士課程在籍。言語処理系に興味を持つ。



笹田 耕一 (正会員)

東京大学大学院情報理工学系研究科助手, 助教, 講師 (2006~2012年)。株式会社セールスフォース・ドットコム, Heroku, Inc. (2012~2017年)。現在はクックパッド株式会社にてRubyインタプリタ開発に従事 (2017年~)。言語処理系や並列処理システムに興味を持つ。



テュールスト マーティン ヤコブ (正会員)

チューリッヒ大学経済学部修士課程修了。1990年東京大学大学院理工学研究科情報科学専攻博士課程修了, 理学博士。チューリッヒ大学情報科学科主任助手, 慶應義塾大学政策・メディア研究科にて特別研究助教授, マサチューセッツ工科大学客員研究員を経て2005年青山学院大学理工学部情報テクノロジー学科着任。現教授。