Ruby のメモリ管理の改善

○ 笹田　耕 †1

プログラミング言語 Ruby の処理系は，保守的マークアンドスイープガーベージコレクション（GC）による自動メモリ管理機構を有す．本稿では，より高性能，高効率なメモリ管理を実現する手法について検討する．まず，ヒープサイズをこれまでよりも大きくとることで GC オーバヘッドを削減する方針をとる．ヒープ拡大により，オブジェクトフラグメンテーションが発生するが，これを防ぐためにいくつかの手法について検討する．また，マーク時にマークカウンタを用いることで，スイープ時にブロック単位で領域を開放することを可能にし，スイープ時間の削減を目指す．また，他プロセスと協調したヒープサイズの調整を行うため，ページアウトを監視する手法についても述べる．また，これらを実現するために `mmap()` システムコールなど，OS 依存の機構を用いてメモリ管理機構を一新する．本発表では，これらの手法の検討結果および実装方法について述べ，予備評価の結果を示す．

Improving Memory Management for Ruby

○ Koichi Sasada †1

The interpreter of programming language Ruby has a conservative mark and sweep garbage collection (GC) mechanism. In this paper, we consider about performance and efficiency improvement about memory management on the Ruby interpreter. At first, we take a course to expand heap area to improve memory management. Solutions against object fragmentation problems due to heap expansion are considered. We also introduce mark counter technique to reduce a sweep overhead. To make cooperative heap-size adjustment with other processes in same system, page out information are used. We implement memory mangement system with mechanisms depend on OS such as `mmap()` system call. In this presentation, we show our proposals, implementations, and result of preliminary evaluation.

1. はじめに

オブジェクト指向スクリプト言語 Ruby[12)11)]（以降 Ruby）は，その使いやすさから世界中で利用されているプログラミング言語である[15)]．

Ruby は，開発開始時にすでに存在した Perl[7)] など，その他のスクリプト言語と同様に，簡単なテキスト処理を，より書きやすくするための言語といった位置づけであったが，最近ではウェブアプリケーション開発のためのプラットフォームとしての言語，サーバの構築[16)14)] での利用といった，多くの用途に使われるようになってきた．とくに，Ruby on Rails[6)] の爆発的な普及から，Ruby をウェブアプリケーション作成用プログラミング言語として利用することが多くなってきた．

Ruby は，多くの現代的なプログラミング言語と同様，ガーベージコレクション（GC）による自動的なメモリ管理をサポートしている．GC により，Ruby スクリプトを記述するプログラマは煩雑なメモリ管理を行わなくてもよい．

我々が開発を進めている C 言語による Ruby 処理系（以降，Ruby 処理系）[17)] は，GC に保守的マークアンドスイープ GC を用いて実装されている[11)]．保守的 GC を用いているのは，C 言語による拡張機能（以降，拡張ライブラリ）の実装を容易にするためである．具体的には，全スレッドのマシンスタックをマークを行うためのルートとする．この工夫により，拡張ライブラリの実装者は JNI[4)] での Java 仮想マシンを拡張するときなどで求められる，冗長なオブジェクトの処理を記述しなくてもよいという利点がある．ただし，保守的 GC を用いているため，コピーアルゴリズムなどを単純に用いることができないため，マークアンドスイープアルゴリズムを用いている．

このような Ruby 処理系の GC 実装は，他の GC アルゴリズムを採用した他の言語処理系に比べると効率が悪いが，当初の簡単なテキスト処理といったアプリケーションでは問題とならなかった．しかし，Ruby が多様な目的に利用されるようになったため，GC を含めたメモリ管理について，速度や効率などが問題となるようになってきた．

この問題を解決するための手法として，GC アルゴリズムを変更することが考えられる．すでに，Ruby のメモリ管理を改善するためにいくつかの手法が提案されている[13),21),22)]が，しかしこれらの研究は Ruby 処理系のソースコードレベルでの互換性，バイナリレベル

†1 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo

での互換性を保つことができない．

そこで，本稿では実際的な Ruby 処理系開発者の見地から，既存の Ruby 処理系のソースコード互換性，バイナリ互換性を保ちつつ，メモリ管理を改善する方式について検討する．方針としては，ヒープサイズをできる限り大きくとることで GC のオーバヘッドを削減する方法を検討する．また，これを実現するために，従来の Ruby 処理系では用いられてこなかった，OS に依存する機能を利用した性能改善を行う．ただし，例えば Linux のみでしか利用できない，といった機能は極力使わず，POSIX などの規格レベルでの，なるべく多くのプラットフォームで実装可能となる範囲に絞る．もちろん，OS を改変しなければならないような機能は使わず，既存の OS で動作するようにする．

具体的には，(1) ページアウトを契機としたヒープサイズの自動調整，(2) ブロックごとにスイープ操作を行うことによるスイープ時間の削減を行う．また，これを実現するために (3) メモリマネージャと連携したヒープの管理を Ruby 処理系に実装し，予備評価を行った．また，さらにオブジェクトの寿命を用いたオブジェクトフラグメンテーションの阻止方式について検討を行った．

本研究は，抜本的なアーキテクチャの改善を行う多くのメモリ管理改善手法の提案とは異なり，互換性のために多くの制約を抱えた Ruby 処理系という言語処理系に対して，既存のプログラム資産を有効に活用可能としながら改善を行うところに特徴がある．

本稿の構成は以下の通りである．2 章で既存の Ruby 処理系のメモリ管理について説明する．3 章で改善手法について検討し，提案する．4 章で実装について述べ，5 章で予備評価を行う．6 章で関連研究について述べ，7 章でまとめる．

2. 現在の Ruby 処理系でのメモリ管理

本章では Ruby 処理系のメモリ管理手法，オブジェクト管理手法について述べる．なお，本章で述べる内容は詳細は[11),18)] に詳しい．

2.1 ヒープ管理

Ruby オブジェクトは，RVALUE という 5 マシンワードの固定長構造体によって表現される[⋆1]．文字列や配列など，RVALUE に格納しきれない場合は，その都度 malloc() によりその領域を確保する．このように malloc() により管理する箇所については，Ruby 処理系のメモリ管理の一部ではあるが，本稿ではこれ以上議論しない．つまり，本稿が対象とするメモリ管理とは，オブジェクトの管理，つまりこの RVALUE の管理と言い換えることができる．

⋆1 Fixnum オブジェクトなど，一部のオブジェクトはタグにより即値で表現される．

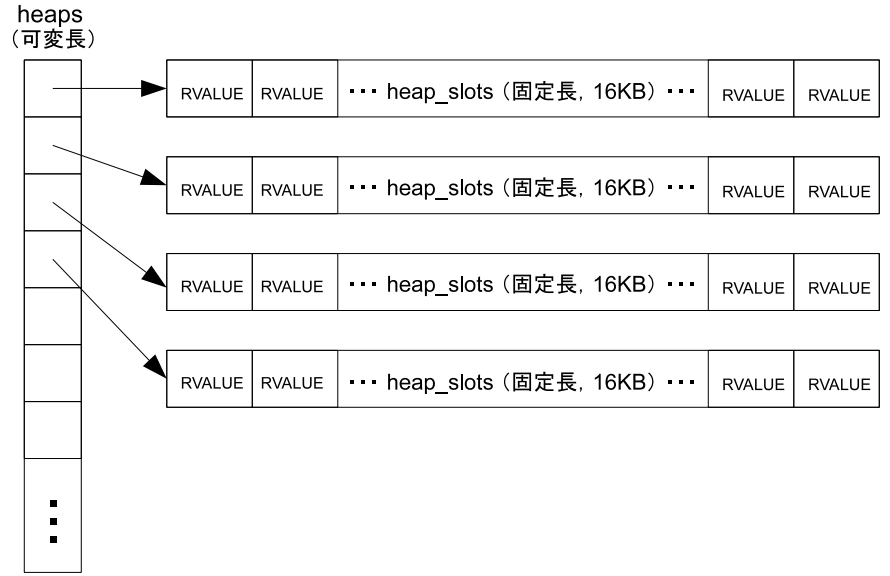

図 1 現在の Ruby 処理系のメモリ管理

RVALUE は heap_slots という固定長の配列（現在は 16KB）によって表現される．さらに，heap_slots は heaps という可変長の配列によって管理される．生成可能なオブジェクトの数は，heaps のエントリ数 heap_slots のサイズをかけあわせたものとなる（**図 1**）．新たにヒープを追加するときは，heap_slots を追加し，heaps からそれを指すようにして行う．

2.2 ガーベージコレクション

1 章で述べたように，Ruby 処理系では拡張ライブラリの記述の容易性を優先するために保守的マークアンドスイープアルゴリズムを採用している．

まず，マークフェーズではマシンスタック，マシンレジスタ，仮想マシン（VM）スタックなどをルートとして，辿れるオブジェクトをマークする．マークするときには，あるポインタが RVALUE を指すポインタであるかどうかを判断するための is_pointer_to_heap() 関数を用いてマーク対象であるかどうかを判別し，そうであれば RVALUE のヘッダ部にある

マークビット領域をセットする．

スイープフェーズでは，heaps から，heap_slots 経由ですべての RVALUE をたどり，マークビットがセットされていればそれをクリアし，されていなければ使われていないと判断しフリーリストにつなぐ．この操作により，フリーリストは利用されていない RVALUE の集合となる．

オブジェクトを新規作成するときは，フリーリストから RVALUE を取り出すことで行う．フリーリストが空だった場合，GC を起動する．GC を起動しても十分な数の RVALUE が得られない場合，heap_slots を新たに追加する．具体的には，存在する heap_slots の数の 1.8 倍となるように調整される．

heap_slots に生きているオブジェクトがない場合，heap_slots は解放できる．逆に，1 つでも生きているオブジェクトが存在する場合，heap_slots は解放できない．このように，少数のオブジェクトが残ってしまうために heap_slots を解放できない問題をオブジェクトフラグメンテーション問題と呼ぶことにする．Ruby 1.8 処理系では，heap_slots のサイズを可変長とし，割り当てごとに大きくしていく設計だったため，オブジェクトフラグメンテーションが顕著であった．この問題を解決するため，Ruby 1.9 処理系（本研究が対象とする Ruby 処理系）では，heap_slots の大きさを固定長（16KB）とすることでこの問題を緩和している．

is_pointer_to_heap() は，ポインタが heaps からたどることができる heap_slots の一部かどうかをテストすることで判断する．この処理は，heaps のエントリ数が少なければ問題無いが，多くなると線形探索で行うには効率が悪い．例えば，100MB のヒープサイズを実現するためには，heap_slots は 6400 個必要である．そこで，Ruby 処理系では heap_slots の集合を，そのアドレスで整列して管理しており，is_pointer_to_heap() 実行時にはバイナリ探索を行うことで効率的な検索を実現している．ただし，heap_slots の追加時には整列させなければならないなど，若干のオーバヘッドが存在する．

2.3 Ruby 処理系のメモリ管理の制約

本節では本稿が対象とする，現在の Ruby 処理系のメモリ管理機構が抱える制約について述べる．

まず，保守的なガーベージコレクションを行う関係上，オブジェクトの移動ができないという制限がある．コピー GC もしくはコンパクションを行うためにオブジェクトの格納位置を変更しようとすると，存在するすべての，そのオブジェクトへのポインタ，つまり RVALUE へのポインタを書き換える必要がある．しかし，保守的 GC アルゴリズムを採用しているため書き換え可能かどうか判断ができない．そのため，オブジェクトのコピーを利用するメモリ管理改善はできない．

また，拡張ライブラリの記述の負荷を下げるため，ライトバリアなどを書かせないというポリシとなっており，実際にそのようなポリシの元で多くの拡張ライブラリが開発されている．このため，ライトバリアを利用する GC アルゴリズムを利用することはできない．ハードウェアおよび OS が提供するメモリ保護機構を使ったライトバリア機構も考えられるが，Ruby ではユーザ定義データに RVALUE へのポインタを格納することができるので，正しくハードウェアライトバリアを実現するためには全メモリ領域にライトプロテクトをかける必要が生じてしまうため現実的ではない．

これらの制約は既存の Ruby 処理系の実装方法によって生じる制約である．この制約を緩めるためには Ruby 処理系の実装方針を変更することで実現できる．しかし，すでに多くの拡張ライブラリが現在の Ruby 処理系の実装方針で記述されているため，実際的な立場から，これを変更するのは容易ではない．

そこで，本稿ではこれらの制約を前提として議論を進めていく．

3. 検　　討

本章では，Ruby のメモリ管理を改善するための手法について検討する．改善のための前提として，実用的な観点から，過去の Ruby 処理系および拡張ライブラリについて，ソースコードレベルでの互換性およびバイナリレベルでの互換性は堅持するものとする．

3.1 改善手法の検討

前章で見たとおり，既存の Ruby 処理系および拡張ライブラリとの互換性を保つことを前提とした場合に生じるこれらの制約の上で，メモリ管理のオーバヘッドを削減しメモリ利用効率を向上する方法を考える．

まずは，GC のオーバヘッドについて考える．保守的 GC であるため，オブジェクトの移動が出来ないということからコピーアルゴリズムの導入は出来ない．つまり，マークアンドスイープアルゴリズムをこのまま利用することが前提となる．世代別 GC とするためには，ライトバリア挿入が難しいという点が問題となり導入できない．従って，既存の保守的マークアンドスイープアルゴリズムを高速化する方法について検討する．

ここで，あるアプリケーション A の総 GC 時間（A が終了するまでに行った GC の実行時間の合計）T_{gc} について，単純化したモデルを考えてみる．A は終了までに合計 N_A のオブジェクトを生成するとし，ヒープサイズ H で実行すると考える．なお，ここでいうヒー

プサイズとは，同時に存在可能なオブジェクト数の最大値を指す．また，ヒープサイズ H はアプリケーション A を実行するのに十分なサイズであるとする．

マークアンドスイープアルゴリズムでの 1 回の GC にかかる時間は，マークフェーズの実行時間 T_m とスイープフェーズの実行時間 T_s の合計である．ここで，i 回目の GC 起動時に生きているオブジェクトの数を L_i とすると，i 回目の GC でのマーク時間 T_{m_i} は，生きているオブジェクトの数に比例した時間がかかるので $L_i W_m$ となる（W_m は 1 つのオブジェクトのマークにかかる時間）．ここで，簡単のために $L_i = L$ と固定する．つまり，GC のタイミングでは一定個数の生きているオブジェクトが存在するとする．ここから，一回のマークフェーズの実行時間は $T_m = LW_m$ となる．同様に，スイープフェーズの実行時間 T_s はヒープサイズ H に比例するため HW_s となる（W_s は 1 つのオブジェクトについてスイープするために必要な時間）．$H - L$ 個のオブジェクトを生成するたびに GC が起動することになるため，アプリケーション A では合計 $\frac{N_A}{H-L}$ 回の GC が行われる．

このモデルのもとで T_{gc} を求めると，次のような式が得られる．

$$
\begin{aligned}
T_{gc} &= \sum_{i=1}^{\frac{N_A}{H-L}} (T_{m_i} + T_{s_i}) = \sum_{i=1}^{\frac{N_A}{H-L}} (LW_m + HW_s) = \frac{N_A}{H-L}(LW_m + HW_s) \\
&= \frac{N_A}{1 - L/H}(\frac{L}{H}W_m + W_s)
\end{aligned}
$$

この単純なモデルから，H が大きければ T_{gc} は減少するということがわかる．回りくどく書いたが，ヒープサイズを大きく取れば，総 GC 実行時間は短くなるという，直感的な話である．

そこで，GC のスループットを向上するために，(1) ヒープサイズ H を出来るだけ大きくしてマーク時間，そして T_{gc} を小さくする，(2) スイープを工夫することで W_s を小さくし，スイープ時間を短くする，という 2 つの戦略を検討する．

3.2 ヒープサイズをできるだけ大きくとる手法の検討

ヒープサイズ H を大きくするためには，システム上で動作するアプリケーションが 1 つしかなければ，実メモリが許す限りヒープサイズを大きくすればよい．しかし，マルチプロセス環境のシステムでは，そのような仮定は成り立たない．

そこで，システムが提供するメモリ管理に関する情報を利用し，できるだけ大きなヒープサイズ H を利用する方法について検討する．

3.2.1 ヒープサイズ決定手法の検討

他のプロセスのメモリ利用状況を，既存の OS の機能を利用して正しく把握するのは困難である．そこで，本研究では自プロセスのメモリのページアウト回数を監視することで，他のプロセスのメモリ利用状況を推測するという手法を提案する．

まず，従来の Ruby 処理系にくらべ，多くのヒープ領域を確保し，それを利用してプログラムの実行を進める．従来よりも多くのヒープ領域を利用するため，ガーベージコレクションの起動回数が減少し，オーバヘッドを削減することができる．

もし，自プロセスページアウトが 1 度でも発生すると，システムでメモリ利用状況が逼迫しているという推測ができる．そこで，自プロセスにおいて一度でもページアウトの発生を検知すると，解放可能なメモリを解放することにより，システム全体のスループット向上を目指す．このとき，従来の Ruby 処理系が行っていたメモリ管理方式に戻る．つまり，本手法は，最悪でも従来のメモリ管理効率は保つことができる．

多くの POSIX 環境において `getrusage()` システムコールを用いてページアウト回数を得ることができる（Linux などでいうメジャーフォールト回数）．そのため，本研究ではこのシステムコールを利用して実装を行うことにする．

3.2.2 オブジェクトフラグメンテーション対策の検討

ヒープサイズをただ大きくするだけでは，オブジェクトフラグメンテーションによって必要なときにメモリを解放することができない．つまり，ページアウトが発生してメモリを返却する必要が生じたとき，`heap_slots` に 1 つでも生きているオブジェクトが存在すると `heap_slots` を解放することができず，システムにメモリを返却することができない．オブジェクトの移動ができればコンパクションを行うことが可能であるが，オブジェクトの移動ができないという制約を前提としているため，それも出来ない．

そこで，オブジェクトフラグメンテーションの対策として 2 つ検討した．

まず，現在 `heap_slots` のサイズが 16KB となっているが，これを 4KB にすることを検討した．これで，オブジェクトフラグメンテーションの確率が若干下がることになる．

次に，オブジェクトの寿命予測を利用して，短寿命オブジェクトを一箇所に集中させる方式を検討した．経験則からプログラムが生成するほとんどのオブジェクトは短寿命オブジェクトであるため，短寿命オブジェクトを一箇所に固めることにより，その箇所がフラグメンテーションされる可能性は低下する．

最も単純な寿命予測は，オブジェクトの種類により分別する方式である．例えば，クラスやモジュールといったオブジェクトはプログラムの初期化時に生成され，終了時まで生きて

いる可能性が高い．逆に，文字列や浮動小数点数オブジェクトなどは，すぐに不要になるテンポラリオブジェクトであることが多い．そこで，種類によって短寿命，長寿命と分ける方法が考えられる．ただし，もちろん精度は低くなる．

また，プロファイルベースによる寿命予測[1),2),8)]が知られている．これは，プロファイラを事前，もしくはオンラインで実行して，プログラムのどの箇所でどのような寿命のオブジェクトを生成するか，という情報を用いて寿命予測を行う方式である．プロファイルベースの寿命予測は世代別 GC において最初からオブジェクトを殿堂入りさせる場面などで利用する．本研究では，逆に短寿命であるかどうかを得る必要がある．

ただし，検討した 2 方式とも，あくまで確率によるフラグメンテーション防止策であるため，完全なフラグメンテーション防止にはならない．対策としては，オブジェクトの寿命予測をかなり保守的に行い，ほぼ短寿命であることが確実であるオブジェクト以外を短寿命オブジェクトとして扱わない，といった方針が考えられる．

また，ウェブアプリケーションなどの比較的ステートレスなアプリケーションにおいては，定期的にアプリケーションを終了することにより，ヒープを空にしてフラグメンテーションを解消するという方式も考えられる．ただし，ネットワークコネクションなどを保持し続けなければならないデーモンプログラムなどは，それは難しい．

なお，時間の関係から寿命予測については検討のみとし，`heap_slots` のサイズの縮小のみ，実装と予備評価を行った．

3.3 ブロック単位の解放によるスイープ時間削減

スイープ時間がオブジェクトの数 N_A に比例するのは，オブジェクトの後始末，フリーリストの構築，マークビットのクリアなどのために各オブジェクトを辿る必要があるためである．

そこで，可能ならばスイープ動作をスキップする方式を検討する．

まず，`heap_slots` ごとにマークカウンタと，ファイナライザカウンタという 2 つのカウンタを用意する．オブジェクトを生成したとき，回収時に後始末が必要なオブジェクトであれば，そのオブジェクトが属する `heap_slots` のファイナライザカウンタを 1 つ増やす．GC のマークフェーズにてオブジェクトをマークするとき，そのオブジェクトが属する `heap_slots` のマークカウンタを 1 増やす．

スイープ時，`heap_slots` のマークカウンタおよびファイナライザカウンタが 0 であれば，各 `RVALUE` を辿らなくてもその `heap_slots` が解放可能であることがわかるので，即座に解放対象と判別できる．つまり，`RVALUE` ごとに辿る必要はなくなり，スイープ時間が削減できる．なお，0 でなければマークカウンタを 0 にして従来通りの後始末を行う．

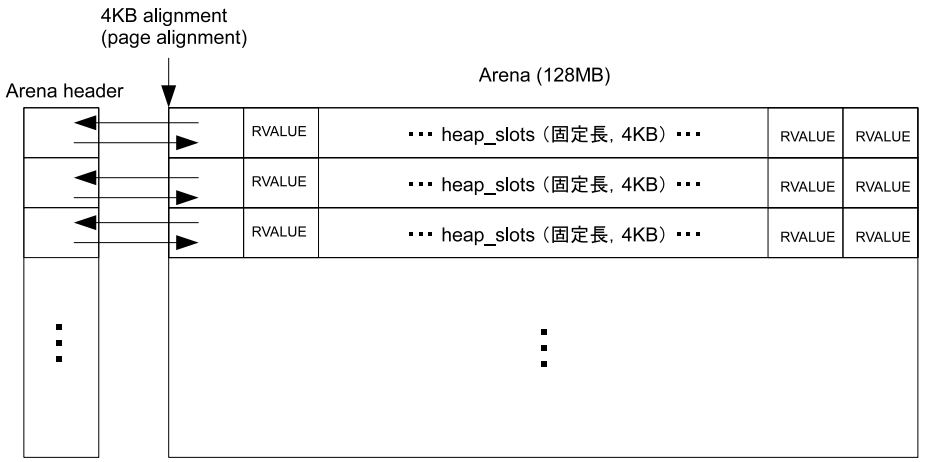

図 2 mmap(), madvise() を利用したメモリ管理

マークカウンタを処理するため，マーク操作に若干のオーバヘッドが追加されるが，生きているオブジェクトの数はヒープサイズよりも小さいため，このオーバヘッドは相対的に小さいと考えられる．

今回はマークカウンタとファイナライザカウンタをカウンタとして実装したが，その `heap_slots` を即座に解放してよいかどうかを示すためのものであるため，フラグとして実装してもよい．

なお，時間の関係からファイナライザカウンタは未実装であり，本稿での評価は行わない．

4. 実　　装

3 章で検討したメモリ改善手法の実装を解説する．

4.1 mmap(), madvise() による明示的なメモリ管理

まず，`heap_slots` を 4KB で効率良く管理するために，従来の `malloc()`，`free()` による管理から，`mmap()`，`madvise()` による管理に変更した[⋆1]（**図 2**）．

まず，`mmap()` により大きな領域（今回の実装では 128MB のメモリ）を確保する．この

⋆1 Microsoft Windows の場合は `VirtualAlloc()` を利用する．

領域を Arena と呼ぶ．mmap() で確保したメモリ領域はページ境界（今回は 4KB と仮定）でアラインメントされている．また，アクセスされない限り，OS は実際にはメモリを割り当てないため，128MB を即座に消費するわけではない[⋆1]．

ここから，4KB ずつ切り出して heap_slots として利用する．128MB を 4KB ずつ利用するため，Arena は 32768 個の heap_slots を用意することができる．どの領域が heap_slots として利用中であるかどうかは，利用状況を示すビットマップを別途用意することによって実現する．つまり，空き領域を探すときは，このビットマップを探索すればよい．また，空き heap_slots リストを用意することで heap_slots の新規割り当てを高速化している．

不要になった heap_slots は，サイズがページ単位であるため madvise() システムコールにより解放することができる．Linux の場合は MADV_DONTNEED オプションを指定し，BSD 系の OS では MADV_FREE を指定することで，その領域が必要ないことを OS に直接通知することができる．もし，不要領域がページアウトされていた場合は，そのページを触ることなく解放することができるため，余計なページインが存在しない．

なお，短寿命オブジェクトを隔離[⋆2]することで，1 度の GC により大量の heap_slots の解放を行う可能性がある．大量の解放は，システムコールの発行，および OS のページテーブル書き換えによりオーバヘッドが発生する．

そこで，ページアウトを検出していない場合は madvise() による解放を行わないこととし，ページアウトを検出した段階で madvise() によりページ解放を行うこととする．そのため，未使用の heap_slots には madvise() による解放済みのものと，madvise() で未解放の 2 種類があり，それらを区別できるようにしている．また，heap_slots の割り当て時は，madvise() によって解放していないものを優先的に割り当てることとしている．

4.2 GC 時に必要な操作の変更

ページサイズ（4KB）アラインメントになったため，いくつかの処理が軽量に実装できるようになった．

まず，is_pointer_to_heap() は，求めるポインタが線形にどの Arena に属しているかを探索する．該当する Arena があれば，その heap_slots が生きているかどうか，ビットマップによりチェックすることで実現できる．

次に，RVALUE へのポインタから，マークカウンタ，ファイナライザカウンタを効率的に探し出すため，heap_slots の先頭に Arena ヘッダという，heap_slots ごとに用意する領域へのポインタを格納する．Arena ヘッダには，マークカウンタ，ファイナライザカウンタを含め，各 heap_slots に必要な情報が格納されている．

heap_slots の先頭はページサイズアラインメントであるため，この領域へのアクセスは定数時間のビット操作で行うことができる．この操作は，ビットマップマーキング[20] などに容易に応用できるため，今後確認していきたい．

4.3 ページアウトを検出する仕組み

自プロセスのメモリのページアウトの検出は，getrusage() システムコールによって行う．rusage 構造体の ru_majflt によって実際にページアウトが何回行われたか，ということを示すメジャーフォールトの回数が得られる．つまり，それ以前に保存していた値よりも大きければ，ページアウトが発生したということがわかる．

この検出は，新たな heap_slots の割り当て要求のたびに行う．つまり，4KB 分のオブジェクト割り当てが発生するたびに行う．

もしページアウトを検出したら，GC を起動して不要となったオブジェクトを回収し，解放可能な heap_slots を madvise() によって解放し，OS にメモリを直接返却する．

5. 予 備 評 価

本稿で提案したいくつかの点について評価を行う．執筆時点では，本稿で提案している手法について，実装がすべて終了していないため，予備評価という位置づけとする．

5.1 ベンチマークプログラムと評価環境

本稿執筆時点では，短寿命オブジェクトの予測による隔離が実装できていない．また，ファイナライザカウンタを用意していないので，後始末が必要なオブジェクトで実験を行えない．

そこで，プログラムでは意図的に短寿命オブジェクトが大量に生成する**図 3** で示すマイクロベンチマークプログラムを用いることにした．(2) の 0.5 + 0.7 という計算には意味がないが，1.2 という短寿命の浮動小数点数オブジェクトを大量に生成するプログラムである．また，浮動小数点数は，解放時の後始末が必要ないため，ファイナライザカウンタがなくても正しく処理を行うことができる．(1) の部分で配列を作成することで，プログラム開始か

⋆1 では，どのような時にメモリを消費するか，は OS の実装による．書き込みが行われたときに実メモリ割り当てが行われるのは当然である．しかし，メモリ割り当てされていないページに対して読み込みが行われたとき，ゼロページ（0 で埋められた，書き込み禁止の特殊なメモリ領域）をマップするだけで新しい割り当てを行わない OS もあれば，読み込みが行われた時点で実メモリを割り当てる OS もある．一般的には前者が自然と思われるが，Linux 2.6 の一部のバージョンでは後者の動作を取るようだ．当初，未使用のページかどうかを調べるために，前者の動作を期待してメモリ読み出しを行い，値が 0 かどうかで判断するように設計したが，後者の動作となる環境があったのでビットマップによる明示的な管理を行うことにした．

⋆2 本稿の実験では未実装．

```
size = (ARGV.shift || 10_000).to_i
ary = Array.new(size){|i| i.to_s}  # (1)

50_000_000.times do
  0.5 + 0.7                        # (2)
end
```

図 3 マイクロベンチマークプログラム

ら終了まで生きているオブジェクト（長寿命オブジェクト）の数を任意に設定することができる．指定しなければ 1 万個の長寿命オブジェクトを作成する．

評価は，Intel Xeon CPU E5335 2.00GHz（4 core）のプロセッサを 2 個搭載する，合計 8 コアの CPU，2GB のメモリを搭載したマシンで行った．利用したソフトウェアは次の通りである．OS は GNU/Linux 2.6.26-2-amd64，コンパイラは gcc version 4.3.2 (Debian 4.3.2-1.1) を利用した．比較対象とする Ruby 処理系は ruby 1.9.2dev (2009-10-26 trunk 25491) [x86_64-linux] と，それに提案手法を実装したものである．

5.2 生きているオブジェクトの数の影響

まず，長寿命オブジェクトの数を変更してマイクロベンチマークを実行した結果を**図 4** に示す．X 軸が長寿命オブジェクトの数，Y 軸が実行時間である．

図 4 の proposed が提案手法，current は従来の Ruby 処理系である．discard は，自プロセスのメモリのページアウトの有無とは無関係に，必ず `madvise()` システムコールにより，空き領域を OS に返却するようにしたものである．

評価の結果，proposed の実行時間が安定して短いことがわかる．これは，単一プロセスしか走らないため，ページアウトが発生せず，生きているオブジェクトの数よりも大きなヒープサイズ（この場合，128MB，約 41 万オブジェクト格納可能）を使い続けることができたからだと思われる．

discard は，空き領域を毎回 OS へ返却するため，そのオーバヘッドが観測できている．なお，`time` コマンドでシステム時間を計測して，proposed との実行時間の差がシステム時間であることが確認出来た．

current は実行時間にばらつきがある．これは，現在の Ruby 処理系の，生きているオブジェクト数に対するヒープサイズの選択戦略のパラメータのためだと思われる．

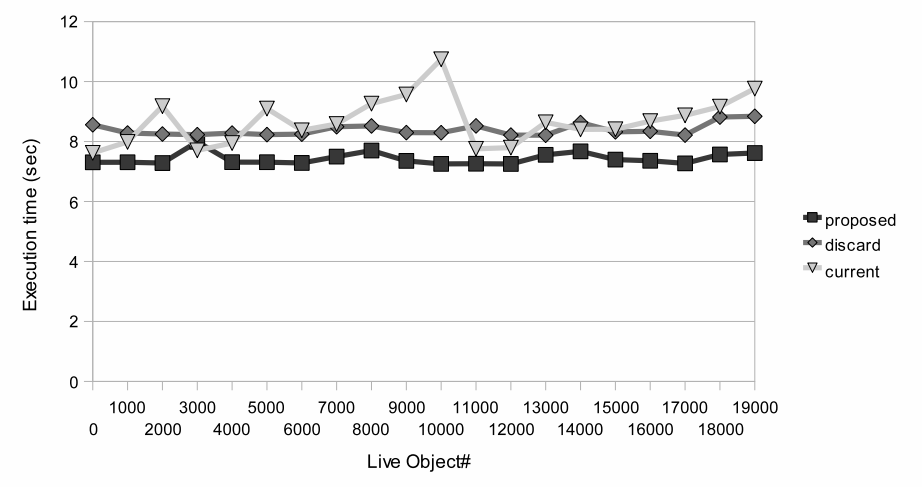

図 4 生きているオブジェクトの数を変更しての実行時間

5.3 複数プロセス実行時の評価

次に，マイクロベンチマークを複数プロセス実行させ，ページアウト監視によるヒープサイズ調整がうまく機能するかどうかを調査した．結果を**図 5**，**図 6** に示す．図 5 はすべてのプロセスが終了するまでの時間，図 6 がその合計時間をプロセス数で割った時間，すなわち 1 プロセスの実行時間の平均である．8 プロセスまでは総実行時間が平ら，つまり 1 プロセスあたりの実行時間が減少しているのは，CPU が 8 コアあり，最大 8 並列同時実行可能だからである．

current と比較して，proposed の実行時間が有意に短い傾向が見られる．これは，1 プロセス実行では proposed のほうが速い傾向が，プロセス数を増加させても維持されているともいえる．最低 128MB の領域を Arena として利用するため，32 プロセス同時実行時には 4096MB の領域を `mmap()` で確保することになるが，ページアウトを検知して `madvise()` により，正しく実メモリを解放できていることがわかる．また，この結果から，自プロセスのメモリのページアウトの検知によりヒープサイズのフィードバックがうまく働いていることがわかる．

実は，当初の予想では，同時実行プロセス数が多くなればメモリを多く利用しようとする proposed よりも，メモリをあまり消費しない（1 プロセス 4MB 程度，top コマンドで確

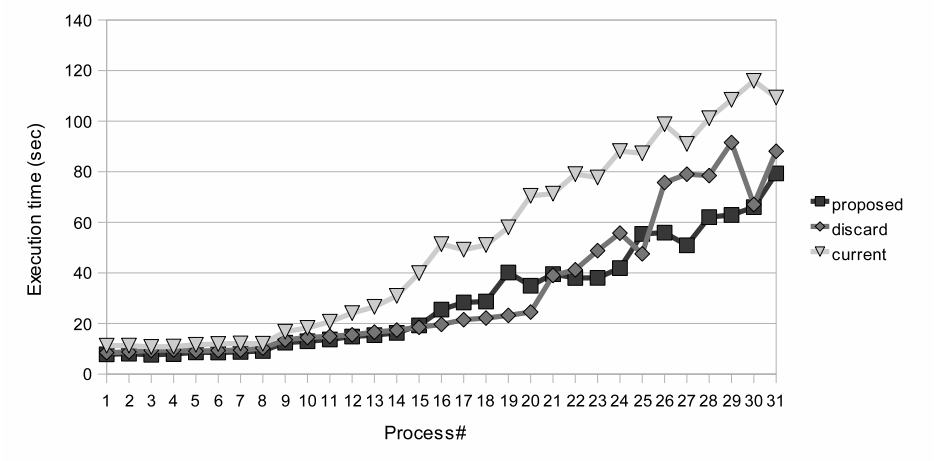

図 5 同時実行プロセス数を増加させたときの実行時間（合計）

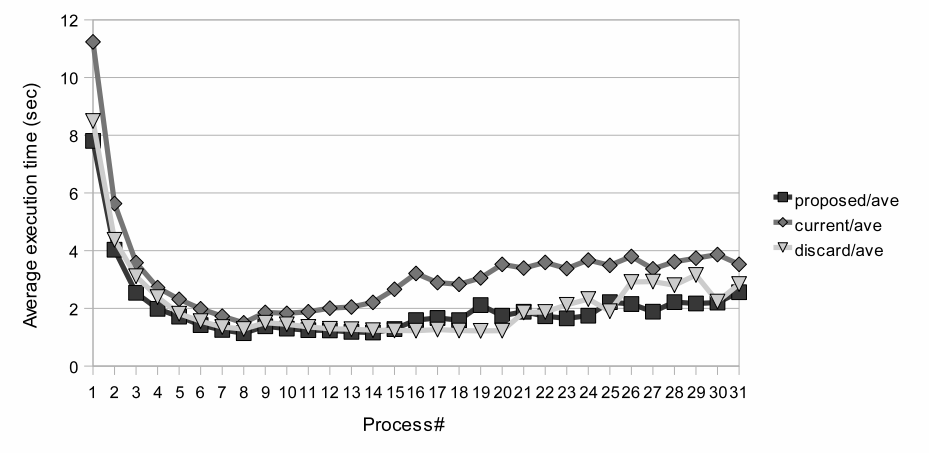

図 6 同時実行プロセス数を増加させたときの実行時間（平均）

認）current が有利になると考えていた．しかし，結果は proposed が概ね高速であるという結果が得られた．これは予測であるが，current では GC のたびにメモリバンド幅を消費するが，proposed ではページアウトでの I/O 処理により実行がストールすることで，その他のプロセスがメモリバンド幅を有効に活用できるからではないかと思われる．

途中，discard が proposed を抜かしているのは，頻繁にメモリを返却することでのシステム全体のメモリ利用効率向上による性能向上が，システムコールおよびページテーブル操作のオーバヘッドを越えたためだと思われる．

なお，並列実行を許可しない，シングルコアでの実験も試みたかったが，手元にシングルコアマシンがなかったこと，シングルコア向け OS をインストールしていなかったことから今回は行わなかった．

5.4 aobench による評価

マイクロベンチマークよりも，もう少し現実的な評価を取るため，同じく浮動小数点数を多く用いるレンダリングプログラムである aobench[3] をベンチマークとして実行した．この結果を**図 7** に示す．

proposed，discard について 5 プロセス以上のデータがないのは，メモリ消費量が大きくなりすぎてしまったためか（OOM Killer 機構か）Linux が異常終了してしまったためである．終了直前の状況を見ると，各プロセスが 128MB の Arena を 4 つ（500MB ほど）確保して実行していることが確認できた．これは，Arena 追加アルゴリズムにバグがあるからではないかと思われる．

1〜5 プロセスでの実行結果を見ると，proposed が高速という結果が得られている．バグを直して，再度検証したい．

6. 関 連 研 究

Ruby のメモリ管理を改善する研究はいくつか存在する[13),19),21),22)] が，どれも既存のソフトウェアとの互換性を考慮していないため，現実的な観点から問題である．なお，本稿で述べた `madvise()` を用いて実メモリを解放するアイデアは，文献 13) で紹介されていたものであり，これからヒントを得て本研究に着手した．

オブジェクトの寿命予測は，効率的なメモリ管理の研究から世代別ガーベージコレクションでの事前昇格などに利用されている[1),2),8)]．しかし，とくに短寿命オブジェクトを利用しようとする研究はない．

システム全体のメモリ資源を見ながらヒープサイズを決めるための仕組みも多く研究されている[5),9),10)] が，従来の研究では OS やハードウェアにメモリ状況を得るための専用の仕組みを組み込むことで実現している．本稿では，単純な `getrusage()` システムコールのみを使って実現している点が異なる．ただし，評価が十分でないため，本当にこの単純な仕

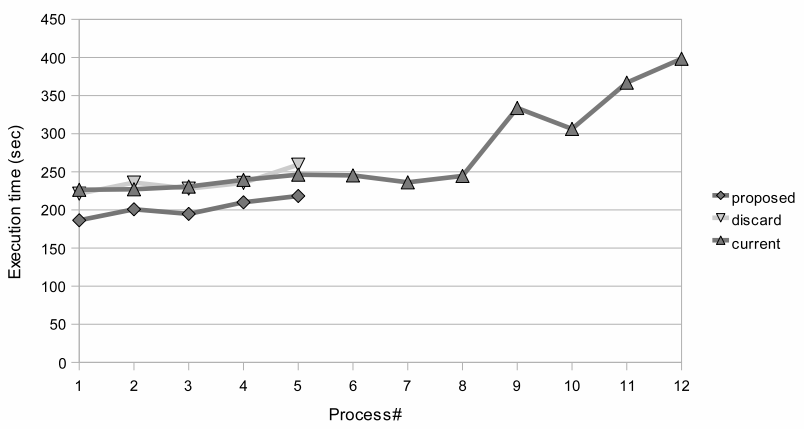

図 7 aobench で同時実行プロセス数を増加させたときの実行時間（合計）

組みで他のプロセスと協調動作できているかを確認する必要があり，今後の課題としたい．とくに，OS が管理するファイルキャッシュ（ページキャッシュ）と実メモリの競合が発生したときにどのような動作になるかは詳しく検討していく必要がある．

7. ま と め

本稿では Ruby 処理系のメモリ管理について，互換性という制約の下で，いかにして効率化するかという点について考察した．

まず，互換性の観点から，保守的マークアンドスイープ以外の GC アルゴリズムを採用することができないことを確認した．その上で，GC オーバヘッド削減にはヒープサイズをできるだけ大きくとることで性能向上が可能であることを示した．ヒープサイズ拡大には (1) 他のプロセスとの協調が必要であること，(2) オブジェクトの移動ができないため，オブジェクトフラグメンテーションを解決する必要があること，について検討した．その結果，(1) は自プロセスのページアウトを監視することでヒープサイズを調整する手法について提案した．(2) は，まず `heap_slots` を 4KB にすることで，フラグメンテーション発生確率を減少させた．さらに，短寿命オブジェクトをまとめることで，オブジェクトフラグメンテーションを防ぐ手法について検討した．ただし，短寿命オブジェクトを確定させる手法については既存研究のオブジェクト寿命予測について検討するだけで，本稿では実装まではしていない．さらに，(3) マークカウントを用いることでブロック単位でスイープすることでスイープの実行時間を削減する手法を提案した．

(1)，(2)，(3) を実現するために，`mmap()`，`madvise()` システムコールによる明示的なメモリ管理機構を行い，また，Arena という新しいメモリ管理の枠組みを導入した．`mmap()`，`madvise()` を利用することで，OS のメモリ管理機構と直接連携することができるようになった．そして，`getrusage()` によりページアウト監視を実現した．

提案手法のうちいくつかを実装し，簡単なベンチマークによる予備評価を行った結果，実行時間を若干短縮することができた．また，ページアウト監視によるヒープサイズの自動調整が機能することを確認した．しかし，実装に問題があり，現実的な規模のプログラムでの効果を確認することはできなかった．

本稿では，いくつか重要な機能の実装がまだ出来ていないため，提案手法全般が現実的な問題に対して有効であるか，まだ確認できていない．とくに，オブジェクトの寿命予測を用いたオブジェクトフラグメンテーションの抑止について実装が間に合わなかったのが残念である．寿命予測にはプロファイルベースで行う場合，プロファイリングオーバヘッド，およびオブジェクトアロケーション時の寿命予測のオーバヘッドがかかるため，全体として性能改善につながるかは興味深い．また，ファイナライザカウンタの実装を行っていないのも問題である．マークカウンタによる `heap_slots` 単位での解放は，後始末するべきオブジェクトがないことが前提であるため，Ruby オブジェクトのデータ構造の再検討も必要になるかもしれない．この未確認部分の検証が今後の課題となる．

なお，本稿で実装した OS のメモリ管理機構と直結する，Arena によるメモリ管理機構は，ビットマップマーキングへの拡張など，本研究をこえて今後の研究開発にとって有用である．今後，さらに Ruby 処理系のメモリ管理手法について発展させていきたい．

謝　　辞

本稿執筆にあたり，ネットワーク応用通信研究所のまつもとゆきひろ氏を初めとした Ruby 開発者の方々，電気通信大学の鵜川始陽氏に助言を頂きました．Linux カーネルのメモリ管理について，富士通株式会社の小崎資広氏に助言を頂きました．本稿で掲載したメモリ管理実装の一部は，プログラミング&セキュリティキャンプ 2009 の参加者によって試験実装を行ってもらいました．協力して下さった方々にこの場を借りて感謝いたします．

本研究の一部は，日本学術振興会科学研究費補助金若手研究（若手（B）），課題番号 21700024 の助成を得て行いました．

参 考 文 献

1) DavidA. Barrett and BenjaminG. Zorn. Using lifetime predictors to improve memory allocation performance. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pp. 187–196, New York, NY, USA, 1993. ACM.
2) StephenM. Blackburn, Matthew Hertz, KathrynS. Mckinley, J.EliotB. Moss, and Ting Yang. Profile-based pretenuring. *ACM Trans. Program. Lang. Syst.*, Vol.29, No.1, p.2, 2007.
3) Syoyo Fujita. Ao bench. http://lucille.atso-net.jp/aobench/.
4) Rob Gordon, 林秀幸 (訳). JNI:Java Native Interface プログラミング. ソフトバンククリエイティブ, 10 1998.
5) Chris Grzegorczyk, Sunil Soman, Chandra Krintz, and Rich Wolski. Isla vista heap sizing: Using feedback to avoid paging. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pp. 325–340, Washington, DC, USA, 2007. IEEE Computer Society.
6) DavidHeinemeier Hansson. Ruby on rails: Web development that doesn't hurt. http://www.rubyonrails.org.
7) O'Reilly Media, Inc. Perl.com: The source for perl – perl development, perl conferences. http://www.perl.com/.
8) MatthewL. Seidl and BenjaminG. Zorn. Segregating heap objects by reference behavior and lifetime. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pp. 12–23, New York, NY, USA, 1998. ACM.
9) Ting Yang, Matthew Hertz, EmeryD. Berger, ScottF. Kaplan, and J.EliotB. Moss. Automatic heap sizing: taking real memory into account. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pp. 61–72, New York, NY, USA, 2004. ACM.
10) Weiming Zhao and Zhenlin Wang. Dynamic memory balancing for virtual machines. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pp. 21–30, New York, NY, USA, 2009. ACM.
11) まつもとゆきひろ, 石塚圭樹. オブジェクト指向スクリプト言語 Ruby. 株式会社アスキー, 1999.
12) まつもとゆきひろ他. オブジェクト指向スクリプト言語 ruby. http://www.ruby-lang.org/ja/.
13) 鵜川始陽. Ruby における mostly-copying gc の実装. 情報処理学会論文誌. プログラミング, Vol.2, No.2, pp. 1–12, 2009.
14) 江渡浩一郎, 高林哲, 増井俊之. qwikweb：メーリングリストと wiki を統合したコミュニケーション・システム. 情報処理学会研究報告. HI, ヒューマンインタフェース研究会報告, Vol. 2004, No. 115, pp. 5–11, 20041111.
15) 松本行弘. Ruby の真実. 情報処理, Vol.44, No.5, pp. 515–521, 2003.
16) 哲高林, 俊之増井. Quickml:手軽なグループコミュニケーションツール. 情報処理学会論文誌, Vol.44, No.11, pp. 2608–2616, 2003.
17) 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 (PRO), Vol.47, No. SIG 2(PRO28), pp. 57–73, 2 2006.
18) 青木峰郎. Ruby ソースコード完全解説. インプレス, 2002.
19) 相川光, 笹田耕一, 本位田真一. Ruby 処理系へのスナップショット gc の実装. 情報処理学会論文誌. プログラミング, Vol.2, No.2, pp. 177–177, 2009.
20) 中村成洋, 松本行弘. 効率的なビットマップマーキングを用いた ruby 用ごみ集め実装. 情報処理学会論文誌. プログラミング, Vol.2, No.2, pp. 176–176, 2009.
21) 木山真人. オブジェクト指向スクリプト言語 ruby への世代別ごみ集めの実装と評価. 情報処理学会論文誌. プログラミング, Vol.42, No.3, pp. 40–48, 2001.
22) 木山真人, 佐原大輔, 津田孝夫. オブジェクト指向スクリプト言語 ruby への世代別ごみ集め実装手法の改良とその評価. 情報処理学会論文誌. プログラミング, Vol.43, No.1, pp. 25–35, 2002.