情報処理学会論文誌　プログラミング　Vol. 2　No. 2　1–12 (Mar. 2009)

Ruby における Mostly-Copying GC の実装

鵜　川　始　陽[†1]

本研究では Ruby VM にコピー GC を実装した．従来の Ruby VM は，メモリ管理に保守的マークスイープ GC を用いていた．その理由は，Ruby には多くの C 言語で記述された拡張モジュールがあり，これらの関数が扱う領域のどこに Ruby オブジェクトへのポインタがあるかが精確には分からないためである．我々は，Bartlett の mostly-copying GC を応用しコピー GC を実現した．基本的なアイデアは，ポインタかどうか分からない値の含まれる「あいまいなルート」から指されるオブジェクトは移動させず，それ以外のオブジェクトのみを移動させるというものである．しかし Bartlett のアルゴリズムは，あいまいなルートが多くのポインタを含む場合に回収できないごみが多くなる．そのため改善したアルゴリズムを提案する．提案アルゴリズムでは，マークスイープ GC を組み合わせることにより，このような場合もごみを回収できるようにした．これによりヒープのコンパクションが可能になり，生きているオブジェクトの減少に合わせて広がったヒープを縮小できるようになった．本論文では，そのアルゴリズムの詳細を述べ，それを実装した VM の性能評価を行う．

An Implementation of Mostly-copying GC on Ruby

Tomoharu Ugawa[†1]

We implemented a copying GC on a Ruby VM. The Ruby VM has been employed a conservative mark-sweep GC. This is because Ruby has many extension libraries written in C and we cannot find pointers to Ruby objects accurately in memory used by them. We implemented a copying GC on the Ruby VM using the idea of mostly-copying GC of Bartlett. The basic idea is that the collector pins the objects pointed by the "ambiguous root", where we cannot find pointers accurately, and moves only the others. However, Bartlett's algorithm retains many garbages when ambiguous root has many pointers. Thus we propose a variant of the algorithm. The algorithm collects garbages by using the technique of mark-sweep GC under such an environment. The algorithm allows us to compact the heap. As a result, we can shrink the heap when the number of live objects is decreased. In this script, we show the details of the algorithm and evaluate the performance of the VM with the algorithm.

1. はじめに

近年，計算機の性能向上により，手軽にプログラミングができ生産性の高いスクリプト言語が重要な役割を果たしている．Ruby 言語[1] はシンプルかつ強力なオブジェクト指向スクリプト言語で，強力な文字列処理機能と，例外処理やイテレータなどの高度な言語機能を備える．その手軽さと強力さから，簡単なテキスト処理だけでなく，ウェブアプリケーションやユーザインタフェース，さらには，ゲームや大規模なアプリケーション全体など，様々な用途に利用されている．

それを支えている Ruby の特徴の 1 つが，拡張の容易さである．Ruby では，比較的簡単に C 言語を使って拡張モジュールを作ることができる．Ruby では C 言語のポインタをカプセル化するデータ型が用意されており，これを用いて任意の C 言語のデータを Ruby 言語に持ち込むことができる．また，C 言語の関数中で Ruby オブジェクトへの参照を直接扱うこともでき，Ruby オブジェクトへの参照を C 言語のデータ中に書き込むことも許されている．

一方で，C 言語で記述された拡張モジュール内で自由に Ruby オブジェクトを操作できることが，システム全体のメモリ管理に制約を付けている．Ruby は他のスクリプト言語と同様にごみ集め（以下 GC）を備え，どこからも参照されなくなったオブジェクトの領域を自動的に回収し再利用する．拡張モジュール中の C 言語の関数はマシンスタックを使って実行されるため，マシンスタックのどこかから Ruby オブジェクトが参照される可能性がある．しかし，マシンスタックのどこに Ruby オブジェクトへの参照があるか精確には分からない．また，C 言語のデータ中についても，Ruby オブジェクトへの参照とそうでない値が混在していて，どこにポインタがあるか精確に分からない，あるいはどこにあるか管理しておくのが大変な場合がある．そこで Ruby 処理系では保守的 GC[2] が採用されている．

保守的 GC では，オブジェクトへの参照かそれ以外の値（たとえば整数値）か分からない値は，保守的にオブジェクトへの参照として扱い，参照されているオブジェクトを回収しないようにする．しかし，「参照らしき値」は実際には整数値かもしれないため，参照されているオブジェクトを移動させることはできない．もし移動させた場合，「参照らしき値」を移動先を指すように書き換えなければならないため，もし実際には参照でなかった場合

†1 電気通信大学情報工学科
Department of Computer Science, The University of Electro-Communications

© 2009 Information Processing Society of Japan

には値が変化してしまうことになる．そこで，保守的 GC はオブジェクトを移動させないマークスイープ方式と組み合わされることが多い．Ruby も保守的マークスイープ GC[2] を採用している．

オブジェクトを移動させないメモリ管理では，メモリフラグメンテーションが問題となる．Ruby では，すべてのオブジェクトを同じサイズに揃えることで，フラグメンテーションに対処している．オブジェクトのサイズが揃っていると，回収された領域には必ず別のオブジェクトを割り当てることができるので，

- 空き領域の合計は十分にあるが，オブジェクトを配置するのに十分な連続メモリが確保できない，
- 十分な大きさの領域を探すのに時間がかかり，メモリ割当てが遅くなる，

という問題は起こらない．しかし，

- 生きているオブジェクトがヒープ全体に広がってしまい，ワーキングセットが大きくなる，

という問題は残る．サーバマシンなどで，複数の Ruby プロセスを長時間実行するような場合，この問題は深刻である．1 度生きているオブジェクトの数が増えてヒープを広げてしまうと，その後，生きているオブジェクトが減っても，広がったヒープ全体にアクセスしてしまい，必要以上に物理メモリを割り当てなければならない．複数の Ruby プロセスでこのようなことが起こると，サーバマシン全体で RAM メモリの利用効率が悪くなり，同時に実行できる Ruby プロセスの数が減ってしまう．

保守的マークスイープに対し，Bartlett は mostly-copying GC を提案している[3]．Mostly-copying GC では，「ポインタらしき値」から直接指されているオブジェクトを移動させることはあきらめ，それ以外のオブジェクトを対象にコピー GC を行う．「ポインタらしき値」が少ないときは，この方法でも十分なコンパクションの効果が期待できる．

我々は，Ruby VM に mostly-copying GC を実装しフラグメンテーションの解消を試みた．しかし，Ruby では「ポインタらしき値」が比較的多いことが分かった．Bartlett のアルゴリズムは，このような環境では回収できないごみが多く残ってしまう．そこでこのような環境でもごみを回収できるようなアルゴリズムを提案する．

このアルゴリズムを Ruby VM である YARV[4] に実装して，コンパクションの効果を測定した．実装にあたっては，拡張モジュール中の C 言語で書かれたプログラムの変更を最小限にとどめるように注意した．

以下，2 章では Ruby VM のメモリ管理について，3 章で Bartlett の mostly-copying GC について説明する．次に 4 章で Ruby VM 向けの mostly-copying GC を設計し，5 章でそれを実装する．6 章では実装した処理系の性能を測定し，評価する．その後 7 章で関連研究を示し，8 章で今後の課題を述べ，9 章でまとめる．

2. Ruby VM のメモリ管理

この章では Ruby VM である YARV のメモリ管理について説明する．まずヒープの構成を示し，その後 GC について説明する．

2.1 ヒープの構成

YARV のオブジェクトはすべて 5 ワードで構成されている．そのため，オブジェクトによってはデータが収まりきらないことがある．このような場合，malloc によりメモリを確保し，オブジェクトの外部にデータを保存する．オブジェクトの外部のデータは，ヒープの外に割り付けられることになる．たとえば文字列オブジェクトは 32 ビット環境の場合 11 文字以上になると外部に文字列を保存する．外部のデータは，GC により Ruby オブジェクトが回収されるときに解放される．

ヒープは 5 ワードのオブジェクトしか格納しないため，5 ワードの区画に区切られている．オブジェクトに割り当てられていない区画は 1 本のフリーリストにつながれている．オブジェクトを生成する際には，フリーリストの先頭をとりだして割り当てる．

フリーリストが空になると GC が起動され，回収された区画がフリーリストに戻される．十分な数の区画が回収されなかったときは，ヒープが拡張される．ヒープは，malloc により新たに 5 ワードの区画の配列を確保することで拡張される．malloc により確保されたそれぞれの配列を**ヒープチャンク**と呼ぶことにする．確保されたヒープチャンクのすべての区画はフリーリストにつながれ，利用可能になる．

一方，GC 終了後に生きているオブジェクトを 1 つも含まないヒープチャンクがあれば，そのチャンクの区画はフリーリストから除き，ヒープチャンクのメモリを解放する．

2.2 ごみ集め

ごみ集めには保守的マークスイープ GC[2] が使われている．GC のルート集合には，各スレッドのマシンスタック，レジスタ，クラステーブルやシンボルテーブルなどの各種テーブルなどがある．このうち一部は，どこにポインタがあるか分からない**あいまいなルート**である．あいまいなルートに含まれる「ポインタらしき値」は，ヒープ中のオブジェクト境界を指しているかどうかでポインタかどうか判断する．YARV では，オブジェクトの先頭を 5 ワードの倍数のアドレスに配置することになっているため，いずれかのヒープチャンクの

© 2009 Information Processing Society of Japan

アドレスの中を指しており，かつ 20 の倍数（32 ビット環境の場合）になっている値をポインタとして扱う．

マーク処理ではポインタを深さ優先でたどる．このとき，マークを付けられたオブジェクトの持つポインタをたどって次のオブジェクトを探す．一般の Ruby 組み込みの型を持つオブジェクトはオブジェクト先頭のタグでどこにポインタがあるか分かる．C 言語で記述された拡張モジュールのデータをカプセル化したオブジェクトは，ガベージコレクタからカプセル化されたデータのどこに Ruby オブジェクトへのポインタがあるかが分からないため，拡張モジュールはオブジェクトのデータが持つ Ruby オブジェクトへのポインタをガベージコレクタに教える関数を提供することになっている．この関数は，データが含むポインタを精確に教えてもよいが，データをあいまいなルートとして扱って，あいまいなルートのアドレスを教えてもよい．

スイープ処理では，ヒープをヒープチャンクごとにスイープする．マークの付いていないオブジェクトを見つけたら，そのオブジェクトが外部に保持しているデータの領域（malloc された領域）を解放してから，フリーリストにつなぐ．

2.3 問　題　点

VM 起動時に確保されるヒープチャンクの大きさは 10,000 オブジェクト分で，拡張するたびに前回の 1.8 倍の大きさのヒープチャンクを確保する．ヒープチャンクは非常に大きく，解放できる条件は，そこにオブジェクトが 1 つも存在しないことなので，1 度ヒープを拡張してしまうと，ヒープチャンクを解放できることは非常にまれである．Ruby 1.9 に統合された YARV では，ヒープチャンクを 16 KB 程度（約 800 オブジェクト分）と小さくすることで解放の機会を増やそうとしているが，オブジェクトを移動しないため，様々な寿命を持つオブジェクトが同じヒープチャンクに割り当てられてしまうと，ヒープチャンクは解放できなくなってしまう．

これは，プロセスのワーキングセットが大きくなってしまうという問題につながる．たとえば，一時的に多くのデータを扱うようなプログラムでは，その多くのデータがすべて同時にごみになるような場合を除いて，ヒープチャンクを解放できない．その結果，解放されていないヒープチャンクにはオブジェクトが割り付けられ，プロセスは広い範囲をアクセスすることになる．プロセスのワーキングセット増大は，計算機全体の RAM メモリを圧迫するため問題である．本研究では，mostly-copying GC によりヒープのコンパクションを試みる．

3. Bartlett のアルゴリズム

Mostly-copying GC[3] は Bartlett が Scheme 言語[5] から C 言語へのコンパイラ[6] のために考案した，保守的なコピー GC である．コンパイル先が C 言語であるため，スタックやレジスタなどのルート集合はあいまいなルートとなる．一方，ヒープには型の分かっている Scheme オブジェクトしか割り当てられないため，どこにポインタが含まれているか精確に分かる．ただし，Scheme 言語では継続を一級オブジェクトとして扱えるが，継続オブジェトの持つスタックやレジスタのコピーについては，どこにポインタが含まれているか精確に分からない．このように，

- ルート集合はあいまいなルートとなっている，
- ヒープ中のほとんどのオブジェクトは，そのどこにポインタがあるかが精確に分かる，
- そうでないオブジェクト（これらもあいまいなルートと呼ぶ）があってもよい，

ということを前提としている．このような前提で，あいまいなルートから指されていないオブジェクトのみをコピーする．

以降で，このアルゴリズムの詳細を述べる．

Mostly-copying GC では，ヒープを均一なサイズのブロックに分割する．メモリ割当ては 2 段階で行われる．ブロック中のメモリを割り当てる際には，空き領域の先頭を指すポインタを進めるだけでよい．しかし，ブロック内に必要な大きさのメモリが割り当てられないときは，次の空きブロックを探し，以降はそのブロックのメモリを割り当てる．空きブロックの割当ての際に，すでに割り当てたブロックがヒープを構成する全ブロック数の半分に達していれば，次の空きブロックを探す前に GC を行う．

次に mostly-copying GC の手順を示す．まず，**図 1** の (a) に GC 前のヒープの状態を示す．オブジェクトが割り付けられたブロックが from 空間となっている．

(1) スタックやレジスタなどのルート集合をスキャンする．そこで見つかった「ポインタらしき値」により指されているオブジェクトが含まれるブロックには ***promoted* フラグ**をセットし**昇格**させる．図 1 の (b) では，A の含まれるブロックが昇格している．昇格したブロックは，最終的に to 空間と見なされ，その中のオブジェクトは生き残る．

(2) 空きブロックから to 空間を確保し，そこに昇格したブロックに含まれるオブジェクトをすべてコピーする（図 1 の (c)）．このときコピー元のオブジェクトには，コピー先を指す**フォワーディングポインタ**を残しておく．

© 2009 Information Processing Society of Japan

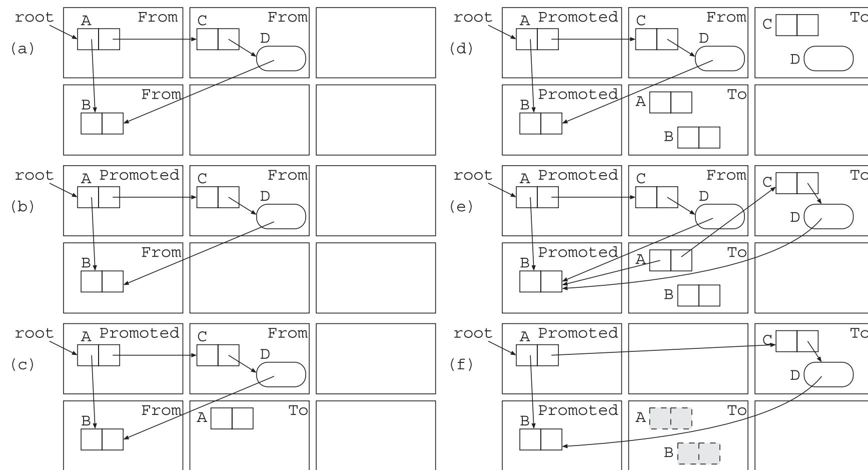

図 1　Mostly-Copying GC
Fig. 1　Mostly-copying GC.

(3)　Cheney の 2 空間コピー GC [7] の要領で to 空間をスキャンし，to 空間に含まれているオブジェクトから指されているオブジェクトを幅優先でコピーする．ただし，コピー GC と違い，to 空間のオブジェクトの持つポインタはまだ更新しない（図 1 の (d)）．ヒープ内にも，あいまいなルートを持つオブジェクト（図 1 では D）がある可能性がある．あいまいなルートを持つオブジェクトも，他のオブジェクトと同様にコピーしスキャンするが，スキャンする際に，あいまいなルートから指されるオブジェクト（図 1 では B）が含まれるブロックも昇格させる．To 空間をすべてスキャンし終わると，すべてのあいまいなルートから指されるオブジェクトが含まれるブロックが昇格したことになる．

(4)　To 空間に属するブロックをすべてもう 1 度スキャンし，ポインタを更新する（図 1 の (e)）．ただし，昇格したブロック中を指すポインタは更新しない．図 1 では，A および D のコピーが持つ B へのポインタは更新せずに残す．ポインタの更新には，コピー元のオブジェクトに残されたフォワーディングポインタを使う．

(5)　昇格したブロックをスキャンし，フォワーディングポインタを使って，コピー先のオブジェクトの内容を to 空間から書き戻す．図 1 の (f) では，A と B が書き戻しの対象になる．コピー先のオブジェクトのポインタは手順 (4) で適切に更新されているため，これを書き戻すことで，昇格したブロックに含まれるポインタが正しいオブジェクトを参照するようになる．

以上の手順の後，from 空間を空き領域として再利用し GC を終了する．

このアルゴリズムには，

- あいまいなルートから指されるブロックに含まれるごみや，そこから指されるごみは回収できない，
- あいまいなルートから指されるブロックに含まれるオブジェクトがいったんコピーされるが，そのコピーはすぐにごみとなり（図 1 の (f) では，影付きの A と B），次に GC が呼び出されるまで回収されない，

という欠点はあるが，これらはブロックのサイズを適切に決めることで軽減することができる．文献 3) の実験では，512 バイトのブロックが適切であるという結果になっている．

4. 設　　計

この章では，YARV のための mostly-copying GC の設計について述べる．

4.1　プロセスサイズの縮小

GC により生きているオブジェクトが減ったときに，プロセスの使っているメモリを縮小できるようにする．

サーバマシンやデスクトップマシンでは通常，ページングによる仮想メモリを利用している．このような環境では，物理メモリの容量を超えるメモリを要求されたとき，いずれかのフレーム（物理ページ）の内容を二次記憶にスワップアウトし，そのフレームを再利用する．スワップアウトするフレームには，最近どのプロセスもアクセスしていないフレームを選ぶのが一般的である．OS からメモリを確保していても，ページ単位でいっさいアクセスしなければ，そのページのフレームは別の用途で再利用される．したがって，減ったときにいっさいアクセスしないページを増やすことで，実質的なプロセスサイズを減らすことができる．

本研究では，GC 後に十分に大きな空き領域が得られたとき，いくつかの空きページに対して，以降オブジェクトを割り付けないようにすることで，いっさいアクセスしないページを増やす．空きページには生きているオブジェクトが含まれていないので，そこにオブジェクトを割り当てない限り，そのページにアクセスすることはない．これらのページに割り当てられているフレームは，計算機全体のメモリが不足したときに再利用される．

Bartlett のアルゴリズムでは，空きブロックは必ずしも連続しない．そのため，同一ページに複数のブロックが含まれる場合，すべてのブロックが空きブロックとならない限り空き

© 2009 Information Processing Society of Japan

ページとはならない．そこで，ヒープをハードウェアのページと 1 対 1 に対応するようにアライメントとサイズを揃えて分割する．これを Bartlett のアルゴリズムでいうところのブロックとして扱う．そのうえで，GC の後に空きブロックがあまりにも多いとき，その一部を空きブロックのリストから外し，**解放されたブロック**として扱う．

ところで，OS はページに割り当てられたフレームを再利用する前に，ページの内容をスワップアウトする．しかし，オブジェクトを割り付けないようにした空きページは有効なデータを含んでいないため，本来このスワップアウトは不要で，できれば避けたい．それには，Linux では madvise システムコールでそのページに `MADV_DONTNEED` というヒントを与えればよい．これにより，ページの内容は保証されなくなるが，OS がスワップアウトすることなく割り当てられたフレームを再利用する．

4.2　あいまいなルートの探索

Bartlett のアルゴリズムでは，あいまいなルートから指されているオブジェクトであってもいったん to 空間にコピーし，コピー先でオブジェクトに含まれるポインタを更新した後に，元の位置に書き戻している．あいまいなルートが非常に少なく，ほとんどのブロックが昇格しないという前提であればこれでも問題はない．しかし，あいまいなルートから指されているオブジェクトが増えると，一時的なコピーが大きな領域を占め，次の GC のために割り当てることができるメモリが減ってしまう．その結果，GC が頻繁に起こり性能が低下することが予想される．

Ruby では，拡張モジュールの作り方によっては，拡張モジュールのデータがあいまいなルートになる．さらに，5.3 節で詳細を述べるが，拡張モジュールを極力変更しないために，拡張モジュールのデータが持つポインタをすべてあいまいなルートからのポインタとして扱う．また，シンボルオブジェクトはシンボル表のキーとして利用されるため，移動させることができない．このような事情で，YARV ではあいまいなルートが多くのポインタを含むことになる．実際，YARV のソースコードと同時に配布されている小さなベンチマークプログラムを使って実験したところ，**表 1** に示すように，ほとんどのベンチマークで GC で生き残ったオブジェクトのうち 40%以上が移動できないオブジェクトであった．表 1 では，毎回の GC 終了時に，生きているオブジェクト (A) と，あいまいなルートから直接指されていて移動できないオブジェクト (B) を数え，その累計を示している．なお，GC が起こらなかったベンチマークプログラムと，エラーで実行できなかったプログラム[★1]の結果は省略した．大量のオブジェクトを扱うプログラムでは，相対的に移動できないオブジェクトの割合が減ると考えられるが，スクリプト言語である Ruby は，それほど大量のオブジェクトを扱わないプログラムにも利用されるため，移動できないオブジェクトが多いときに性能が低下しては困る．

表 1　移動できないオブジェクトの割合
Table 1　Ratio of immovable objects.

program	live (A)	immovable (B)	B/A
factorial	106,227	45,364	42.7%
mandelbrot	1,703,568	486,381	28.6%
raise	1,035,464	441,782	42.7%
strconcat	867,424	367,951	42.4%
concatenate	60,697	25,877	42.6%
count_words	93,841	39,849	42.5%
exception	976,963	414,355	42.4%
lists	54,791	23,226	42.4%
matrix	22,666	9,564	42.2%
object	898,651	383,726	42.7%
random	1,448,571	614,894	42.4%
array	1,724,623	731,028	42.4%
regexp	1,721,709	731,029	42.5%
send	1,728,599	735,270	42.5%
thread_create_join	430,465	183,247	42.6%

そこで，提案方式では，あいまいなルートから指されているオブジェクトをいったんコピーする方式は採用せず，代わりにオブジェクトを動かさずにポインタをたどることで，あいまいなルートを探すことにした．これはマークスイープ GC のマーキングと同様の処理になる．マーキングでオブジェクトに到達したとき，オブジェクトには到達したことを示すマーク（***live* マーク**と呼ぶことにする）以外に，あいまいなルートからのポインタをたどって到達した場合には ***immovable* マーク**を付ける．同時に，そのオブジェクトを含んでいるブロックに *promoted* フラグをセットする．このようにして，余分なスペースを利用することなく移動できないオブジェクトをすべて見つけ出す．

4.3　フリーリストの併用

一般的に使われている計算機では，ページサイズは 4096 バイトになっている．Bartlett の実験ではブロックサイズは 512 バイトが適切で，ブロックサイズを大きくすると，昇格したブロックに割り当てられることになるオブジェクトが増え，ごみの回収率が低下するという結果が得られている[3)]．

★1 計測用の修正を施す前の状態でもエラーとなった．

© 2009 Information Processing Society of Japan

これは，昇格したブロックに含まれるオブジェクトはすべて生きているものとして扱うためであるが，Bartlett の処理系[6)]が様々なサイズのオブジェクトを扱う必要があったことに由来していると考えられる．この場合，動かせないオブジェクト以外の領域を回収して再利用しようとすると，ブロック内の空き領域でフラグメンテーションの問題が発生する．つまり，小さな空き領域が多く作られ，メモリ割当てに時間がかかるようになる恐れがある．それに対して，YARV のヒープには 5 ワードの Ruby オブジェクトしか割り当てられない．そのため，動かせないオブジェクト以外の領域をフリーリストで管理しても，割当て速度の低下はほとんどない．

そこで提案手法ではフリーリストを併用する．GC の際には，昇格したブロックをスイープして，生きているオブジェクト以外の領域を回収しフリーリストにつなぐ．4.2 節でマーキングを行うことにしたので，どのオブジェクトが生きているかはオブジェクトのマークビットで判断できる．

スイープの際には，できるだけ連続領域への割当てができるように，フリーリストにはオブジェクト 1 個分の領域ごとにつなぐのではなく，連続する空き領域をまとめて 1 つのチャンクとしてつなぐ．メモリ割当てにおいて空きブロックを割り当てる際に，フリーリストにチャンクが残っていれば，空きブロックではなく，フリーリストの先頭のチャンクを割り当てる．

ところで，昇格したブロックに含まれる生きているオブジェクトで，あいまいなルートから指されていないオブジェクトは，GC の際に空きページに移動させたほうが，チャンクが大きくなると期待できる．**図 2** では，フリーリストに 4 つのチャンクが含まれているが，あいまいなルートから指されていないオブジェクト A と C を移動させれば，大きな 2 つのチャンクにまとめられる．

しかし，昇格したブロックに含まれるオブジェクトを移動させると，ほとんどのオブジェクトが生き残る場合や，昇格するブロックが多い場合に，GC を行うと空きブロックが減少してしまう危険がある．GC 開始時と終了時で，あいまいなルートから指されるブロックの数は変化しない．それ以外の使用中ブロックの数は，回収されたオブジェクト数に比例して

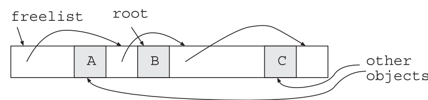

図 2　フリーリスト
Fig. 2　Freelist.

減少する．もし，昇格したブロックに含まれるオブジェクトを移動させると，移動先のブロックは使用中ブロックとなるので，その分だけ使用中ブロックが増える．この増加分が GC による減少分を上回ると，GC によって空きブロック数は減少する．このような事態を避けるために，昇格したブロックに含まれる生きているオブジェクトはすべて動かさないことにした．

4.4　アルゴリズム

以上の設計をまとめたアルゴリズムを示す．

ヒープはハードウェアのページに対応するブロックの配列で構成される．ブロックは次のいずれかの状態を持つ．

- 一杯のブロック（full）
- 部分的に空いたブロック（partial）
- 空きブロック（free）
- 解放されたブロック（released）

ここで，部分的に空いたブロックはフリーリストを持ち，ブロック内の空き領域を管理する．メモリの割当ては 2 段階で行う．通常は空き領域の先頭ポインタをずらして領域を割り当てる．連続空き領域（空きチャンク）の末尾まで割り当てると，次の空きチャンクを用意する．部分的に空いたブロックがある限りそのブロックのフリーリストの先頭のチャンクが次の空きチャンクとなる．部分的に空いたブロックがなくなると，空きブロックが 1 つとり出され，ブロック全体が次の空きチャンクとなる．

空きブロックをとり出すとき，使用中のブロックが解放されたブロック以外のブロックの半数に達していれば GC を呼び出す．GC をしても十分な空きブロックが作られなかったときは，解放されたブロックがある場合，解放されたブロックを空きブロックに加える．解放されたブロックもなければ，malloc によりブロックを確保してヒープを拡張する．

一方，GC 後に空きブロックの割合が多すぎると空きブロックの一部を解放されたブロックにする．このとき，madvise システムコールなどにより，OS にそのブロックに有効なデータがないことを宣言する．

以下に GC の手順を示す．GC ではフリーリストで管理されたブロックがないことを前提としている．しかし，プログラムが強制的に GC を起動した場合などではフリーリストで管理されたブロックが残っているので，このときは空き領域に**空き領域オブジェクト**を詰めてから GC を開始する．

(1)　一杯のブロックを from 空間とする．このときのヒープを**図 3** の (a) に示す．

© 2009 Information Processing Society of Japan

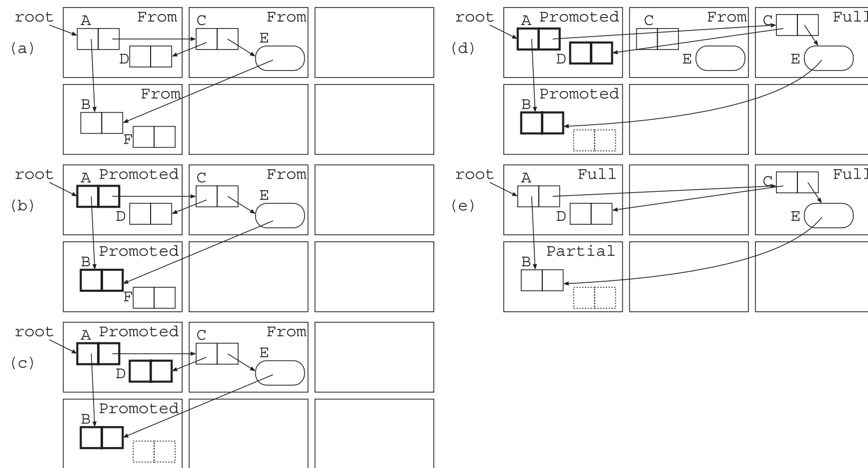

図 3　YARV のための Mostly-Copying GC
Fig. 3　Mostly-copying GC for YARV.

（2）　マークスイープ GC と同様にルートを起点としてポインタをたどり，到達できたオブジェクトに *live* マークを付ける．さらに，あいまいなルートからのポインタをたどってオブジェクトに到達したときは，*immovable* マークも付け，同時に，そのオブジェクトを含んでいるブロックを昇格させる．この様子が図 3 の (b) である．太線になっているオブジェクトが *immovable* マークが付いたオブジェクトで，*live* マークは省略している．

（3）　昇格したブロックをスイープし，*live* マークが付いているオブジェクトには *immovable* マークも付ける．これにより，昇格したブロック中のオブジェクトは移動できなくなる．図 3 の (c) では D に *immovable* マークが付けられている．*Live* マークが付いていないオブジェクト（図 3 では F）はごみなので，オブジェクト外部のデータを解放し，そのブロックのフリーリストにつなぐ．

（4）　昇格したブロック中のオブジェクトをルートとして，Cheney のコピー GC を行う．ただし，*immovable* フラグがセットされたオブジェクトに到達したときは，それをコピーしない．コピーと同時に *live* マークは消す．図 3 の (d) では，C と E が右上のブロックにコピーされ，右上のブロックは「full」の状態になっている．

（5）　昇格したブロックを再度スキャンし，オブジェクトの *live* マークや *immovable* マークを消す．ブロックに付いた *promoted* フラグもリセットする．図 3 の (e) では左上のブロックはごみが回収されなかったので「full」のままになっており，左下のブロックでは F が回収されたので「partial」になっている．

（6）　昇格しなかった from 空間のブロックをスキャンし，ごみになったオブジェクトが外部に持つデータを解放する．その後 from 空間のブロックは空きブロックにする．

以上で GC が完了する．

このアルゴリズムでは，移動できないオブジェクトが増えるとフリーリストで管理するブロックが増え，コンパクションはできなくなる．しかし，ごみは回収できており，メモリ効率が悪化することはない．GC 開始条件を，使用中のブロックが全ブロックの半数に達したら開始としているため，通常のマークスイープ GC の 2 倍のヒープは必要になるが，それ以上は必要としない．

5. 実　　装

4 章のアルゴリズムを YARV に実装する際に，YARV の構造に合わせて拡張した．この章では，その詳細について述べる．

5.1　安定なオブジェクト

YARV のシンボルオブジェクトは，あいまいなルートであるシンボル表からつねに指されている．そのため，シンボルオブジェクトを 1 つでも含んでいるブロックはコピー GC の対象にならない．このようなブロックはできるだけ減らしたい．そのためには，つねにあいまいなルートから指されているオブジェクトや，あいまいなルートから指されている可能性が高いオブジェクトを**安定なオブジェクト**として，同じブロックに割り付ければよい．

そこで，使用中ブロックを安定なオブジェクト用のブロックとそれ以外のブロックに分類する．オブジェクトの割当ては対応するブロックに行う．そのため，空き領域の先頭ポインタはそれぞれのオブジェクト用に別々に用意する．これにより安定なオブジェクトがヒープ全体に分散するのを防ぐことができ，コピー GC の対象とならないブロックを限定できる．

4.2 節で行った実験で移動できなかったオブジェクトの種類を調べると，シンボルの実体である文字列オブジェクトが圧倒的に多かった．そこで，シンボルを管理するルーチンがシンボルの実体である文字列オブジェクトを生成する際に，安定なオブジェクト用のブロックに割り付けるようにした．なお，文字列以外ではクラステーブルから指されているクラスオブジェクトと，抽象構文木を構成するノードオブジェクトが多かったが，今回の実装では，これらは通常のオブジェクトとして扱った．

© 2009 Information Processing Society of Japan

5.2　オブジェクト外部のデータの解放

提案アルゴリズムでは，オブジェクト外部のデータの解放をするために from 空間をスキャンする．そのため，GC には空間の大きさに比例した時間がかかる．しかし，オブジェクトの型によっては，外部にデータを持たないものがある．そのような型のオブジェクトは，5.1 節の要領で専用のブロックに分けて確保する．オブジェクト外部のデータを持たないオブジェクトのみを含むブロックは，コピー GC の対象となったときスキャンを省略できる．これにより GC 時間の削減が期待できる．

5.3　拡張モジュールのマーク処理

Ruby には C 言語で記述された拡張モジュールのデータをカプセル化するオブジェクトがある．カプセル化されたデータの中にも Ruby オブジェクトへのポインタを含むことができるが，ガベージコレクタはどこにポインタがあるかを知ることができない．そこで Ruby では拡張モジュールがガベージコレクタに，どのポインタを含んでいるかを伝える関数を提供することになっている．

もし，カプセル化されたデータから指された Ruby オブジェクトを移動させる場合，そのオブジェクトを指すポインタを更新しなければならない．しかし，既存の拡張モジュールはポインタを更新する関数は提供していないので，拡張モジュールの大きな変更が必要になる．そこで，拡張モジュールから指されるオブジェクトは移動させないことにした．

ところで，拡張モジュールによっては，Ruby オブジェクトである配列を確保し，C 言語のデータに Ruby オブジェクトへのポインタを格納する際は，同時に配列にも同じポインタを格納する構造になっているものがあった．この構造になっていると Ruby オブジェクトへのポインタは，ガベージコレクタには Ruby オブジェクトである配列からしか指されていないように見える．そのため，拡張モジュールのこの部分だけは，配列の中身をあいまいなルートとしてガベージコレクタに伝えるように変更した．

6.　性 能 評 価

YARV に実装した提案手法の評価を行うために，いくつかの測定を行った．この章では，その結果と考察を述べる．

6.1　ヒープの縮小

まず，プログラムが一時的に大量のデータを使った後，拡張されたヒープをどの程度縮小できるかを調べるために，**図 4** と**図 5** に示す 2 つのベンチマークプログラムを作成した．これらはいずれも生きているオブジェクトの数が一時的に増えた後それらがごみとなり，生きているオブジェクトが少ない状態で仕事をする．プログラムの途中には，チェックポイント（CP()）を埋め込んであり，そこでヒープサイズを計測する．

```
2.times{
  ary = Array.new
  10000.times {|i|        # (1) オブジェクトの生成
    ary[i] = Array.new
    (1..100).each {|j| ary[i][j-1] = i.to_f / j.to_f }
    if (i % 100 == 0) then CP() end
  }
  10000.times {|i|        # (2) オブジェクトへの参照を切る
    ary[i] = nil
    if (i % 100 == 0) then CP() end
  }
  30000.times {|i|        # (3) 生きているオブジェクトが少ない状態で仕事をする
    100.times { "" }
    if (i % 100 == 0) then CP() end
  }
}
```

図 4　ベンチマークプログラム 1
Fig. 4　Benchmark program 1.

```
2.times{
  ary = Array.new
  10000.times {|i|        # (1) オブジェクトの生成
    ary[i] = Array.new
    (1..100).each {|j| ary[i][j-1] = i.to_f / j.to_f }
    if (i % 100 == 0) then CP() end
  }
  10000.times {|i|        # (2) オブジェクトへの参照を切る
    sum = 0
    ary[i].each {|x| sum += x }
    ary[i] = sum
    if (i % 100 == 0) then CP() end
  }
  30000.times {|i|        # (3) 生きているオブジェクトが少ない状態で仕事をする
    100.times { "" }
    if (i % 100 == 0) then CP() end
  }
}
```

図 5　ベンチマークプログラム 2
Fig. 5　Benchmark program 2.

図 4 のベンチマークプログラム 1 では，まず (1) のループで浮動小数点オブジェクトを大量に生成し，配列に格納する．配列は 2 段階になっており，外側の配列が 10,000 要素，内側の配列が 100 要素となっている．内側の配列の各要素には浮動小数点オブジェクトが格納される．各要素の計算では少量のごみが作られる．(2) のループでは，外側の配列の要素

© 2009 Information Processing Society of Japan

を `nil` で上書きし，(1) のループで作ったオブジェクトへの参照を切る．(3) のループは文字列オブジェクトを作ってはすぐにごみにしているだけで，寿命の長いオブジェクトは作らない．各ループは 100 回まわるごとにチェックポイントを呼び出す．以上の処理を 2 回繰り返す．

図 5 のベンチマークプログラム 2 もベンチマークプログラム 1 とほぼ同じだが，(2) のループでオブジェクトの参照を切るときに計算をともない少量の生きているオブジェクトを生成する点が異なる．ここでは，内側の配列の要素の総和を求めて，それを内側の配列と置き換えている．総和を求める際の中間結果はただちにごみとなる．

これを，従来の YARV（YARV r590）とそのガベージコレクタを提案手法に置きかえた VM で実行し，チェックポイント通過時のヒープサイズを測定した．また，Ruby 1.9 と統合された YARV ではヒープを 16 KB のチャンクに分割して，ヒープチャンクごとに解放する修正がなされているため，この VM でも測定した．実験環境を以下に示す．

CPU：Pentium4 3 GHz（32 bit），L2 Cache 2 MB
OS：Linux 2.6.22（Ubuntu 7.10）
コンパイラ：gcc 4.1.3（-O2）
libc：2.6.1

実験の結果，**図 6** のグラフに示すようになった．図 6 では，提案手法を実装した VM を **MC**，従来の YARV を **MS**，Ruby 1.9.0-4 の VM を **MS(chunk)** としている．また，生きているオブジェクトの総量を **live object** で示す．

MC は 2 空間コピー GC を基にしているため，最低でも生きているオブジェクトの 2 倍

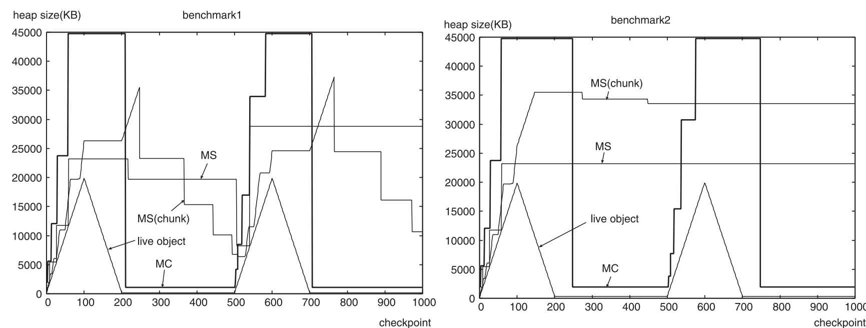

図 6　ヒープサイズ
Fig. 6　Heap size.

のヒープを必要としており，ピーク時のヒープサイズは 3 つの VM の中で一番大きくなっている．しかし，生きているオブジェクトが減ったときには一番多くのヒープを解放できている．今回の実装では，使用中ブロックが全ブロックの 1/8 以下になったときに，使用中ブロックが全ブロックの 1/8 以上になるまでブロックを解放するようにパラメータを設定した．その結果，いずれのチェックポイントでもヒープ使用率が 12.4%以上となっていた．一方，**MS(chunk)** では 1.1%，**MS** では 0.6%まで落ち込むことがあった．

詳しく見ると，ベンチマークプログラム 1 で，2 回目の (3) のループでは，**MS** ではヒープの縮小ができておらず，**MS(chunk)** でも 1 回目のループほど縮小できていない．これは，実行が進むにつれてヒープが汚れ，生きているオブジェクトが多くのチャンクに分散してしまったためと考えられる．長時間動くプログラムであればこの傾向はさらに強く現れると考えられる．

ベンチマークプログラム 2 では，(2) のループでオブジェクトへの参照を切ると同時に，いくつかのオブジェクトを生成している．このうちいくつかは，長時間生き残るオブジェクトである．その結果，長時間生き残るオブジェクトは多くのチャンクに分散する．これが **MS** でも **MS(chunk)** でもほとんどヒープが縮小できていない原因と考えられる．

これらに対し，**MC** ではコンパクションの効果により，2 回目の (3) のループでもベンチマークプログラム 2 でも十分にヒープが縮小でき，ほぼ設定どおりのヒープ使用率を保つことができた．

以上の結果より，一時的に多くのオブジェクトを扱うプログラムにおいて，ヒープ使用率が低下したときに，**MC** はマークスイープ GC より効率的にヒープを縮小できることが分かった．しかし，2 空間コピー GC を基にしているため，生きているオブジェクトが多いときはマークスイープ GC よりヒープサイズが大きくなるという欠点も明瞭に現れた．

生きているオブジェクトが多いときにヒープサイズが大きくなるという問題を改善するには，ヒープ使用率によりマークスイープ方式と切り替えて使うことが考えられる．提案手法では，GC の最初にすべてのオブジェクトをたどってマークを付けるので，そのときにヒープの使用率を計算し，使用率が高いとすべてのブロックに対してスイープを行うようにすればよい．

6.2　実 行 時 間

次に実行時間を従来の YARV，提案手法を実装した YARV，そして Bartlett のアルゴリズムをほぼそのまま実装した YARV で測定した．ベンチマークプログラムには，YARV のソースコードと同時に配布されている単機能のベンチマークの中から，GC が発生したもの

© 2009 Information Processing Society of Japan

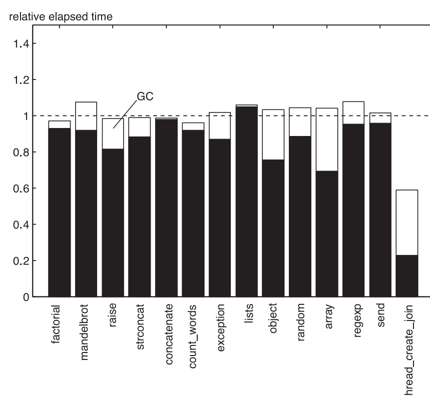

図 7 提案手法を用いた YARV の実行時間
Fig. 7 Elapsed time of our YARV.

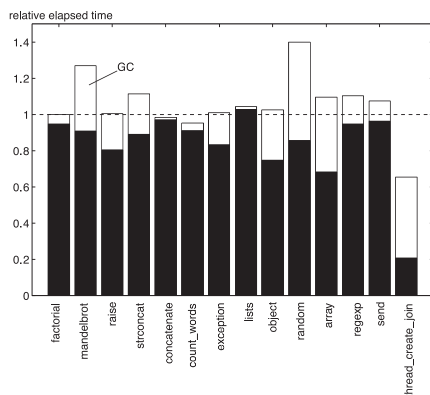

図 8 Bartlett のアルゴリズムを用いた YARV の実行時間
Fig. 8 Elapsed time of YARV with Bartlett's algorithm.

を選んだ．実験には 6.1 節と同じ実験環境を用いた．

実験結果は，**図 7** と**図 8** のようになった．これらのグラフは，従来の YARV での実行時間を基準として，実行時間の比を示している．また，そのうちの，GC にかかった時間を白色で表している．

2 つの実験結果を比べると，Bartlett のアルゴリズムをそのまま実装した場合，GC の時間が占める割合が大きく，実行にかなり時間かかっている．この主な原因は，**表 2** に示すように，GC 回数の増加によるものであることが分かった．表 2 は，**Original**，**Our VM**，**Bartlett** にそれぞれ従来の YARV，提案手法を実装した YARV，そして Bartlett のアルゴリズムをほぼそのまま実装した YARV での GC 回数を示している．

表 2 GC 回数
Table 2 Number of GC invoked.

benchmark	Original	Our VM	Bartlett
factorial	16	16	17
mandelbrot	24	65	235
raise	38	84	222
strconcat	32	70	186
concatenate	9	9	10
count_words	15	15	16
exception	52	76	207
lists	8	8	9
object	36	70	190
random	53	134	405
array	64	141	370
regexp	64	340	374
send	64	141	371
thread_create_join	66	66	67

提案手法を実装した YARV では，**thread_create_join** を除けば実行時間は最小で 96.0%，最大で 107.7%とベンチマークにより結果が分かれていた．単純平均すると 101.9%であった．なお，**thread_create_join** では，malloc で Ruby ヒープの外部に確保したメモリが大幅に増えたことにより Ruby ヒープがほぼ空の状態で GC が起動されているため，結果が他と大きく異る．

さらに，プログラム全体の実行時間と GC の時間のそれぞれについて，従来の YARV と提案手法を実装した YARV で実行に要した時間の差は**図 9** に示すようになった．図 9 では，プログラム全体の実行時間の差（**total**）も GC の時間の差（**GC**）も，従来の YARV でのプログラム全体の実行時間に対する割合で示しており，値が正になっているグラフは提案手法を実装した YARV の方が時間がかかっていることを表している．これを見ると，**object** や **regexp** では GC の時間が増えた分だけ実行時間が長くなっているといえる．一方で，**raise** や **strconcat** では，GC の時間が増えているにもかかわらず，プログラム全体の実行時間は短くなっている．全体として，提案手法を実装した YARV では GC の時間

© 2009 Information Processing Society of Japan

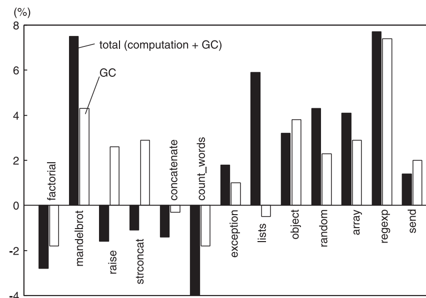

図 9　実行時間および GC 時間の差
Fig. 9　Difference in elapsed time and GC time.

が従来の YARV と異なるが，プログラム全体の実行時間の変化には GC の時間以外の要因もあるといえる．GC の時間以外の要因としては，コピーによりオブジェクトの配置が変更されたことによるキャッシュ効率の変化などが推測され，キャッシュヒット率などの測定が今後の課題となっている．

次に，図 9 で GC の時間が増加しているプログラムについて表 2 を見ると，GC 回数も増えている．このことから，GC の時間が増加したのは，ヒープを to 空間と from 空間に分割するために利用可能な領域が減少し，GC 回数の増加したためと考えられる．

7. 関 連 研 究

Mostly-copying GC の変形には，Attardi らによる Customisable Memory Management Framework [8]（以下，CMM）と，Smith らによる MCC [9] があげられる．

CMM は C++用のメモリ管理ライブラリで，mostly-copying GC を実装している．CMM では，GC 開始時のルートスキャンでルートから直接指されているオブジェクトに印を付ける．これにより，昇格されたブロック中のごみを識別し，ごみから指されたオブジェクトがコピーされるのを防いでいる．この点は我々の提案手法と似ている．我々の提案手法では，1) 昇格されたブロック中のごみ自身も回収する点と，2) コピーの前にすべてのあいまいなルートから指されているオブジェクトに印を付けることで，あいまいなルートから指されているオブジェクトがコピーされないようにする点で異なる．

MCC はオブジェクトを移動させる前にあいまいなルートから指されているオブジェクトに印を付けることで，昇格したブロック内のごみが生き残ることを防いでいる点で我々の提案手法と似ている．MCC では，あいまいなルートがどこにあるかが簡単に見付け出せるように，ヒープ中にあるあいまいなルートをすべてリストにつないで管理している．この方法を Ruby に適用すると，拡張モジュール中の様々な箇所であいまいなルートをリストに登録しなければならなくなる．また，あいまいなルートが多い場合，リストを構成するためのメモリもオーバヘッドになるため，あいまいなルートが多い Ruby には向いていない．これに対して我々の提案手法では，ポインタをたどることであいまいなルートから直接指されるオブジェクトを探している．そのため，拡張モジュールの更新を最小限にとどめ，リストなど余分な構造も必要としない．

木山は Ruby に世代別 GC を実装した [10]．しかし，オブジェクトを移動させる方式ではなく，オブジェクト自身に世代を持たせてマークスイープ GC の処理を軽減させるものであった．一方，Bartlett は mostly-copying GC を用いて，あいまいなルートを許しながら世代間で空間を分けた世代別 GC を実現している [11]．我々の実装も mostly-copying GC を基としておりオブジェクトを移動させられるので，これを発展させれば Ruby でも世代間で空間を分けた世代別 GC が期待できる．

Ruby に lazy sweep [12] の手法を導入して，GC による停止時間を削減しようという試みもなされている．この手法は我々の提案手法と相反するものではない．我々の提案手法では，あいまいなルートを見つけるためのマーク処理，オブジェクトの移動およびポインタの更新は一括して行わなければならない．しかし，これらの処理は生きているオブジェクトの数にしか比例しない．一方で，オブジェクト移動後の from 空間をスキャンして，オブジェクトの外に malloc で確保したデータを解放する処理はヒープ全体の大きさに比例するが，この処理は lazy sweep の手法を用いてインクリメンタルに行える．

8. 今後の課題

提案手法では，生きているオブジェクトが多いとき，マークスイープ GC より性能が劣るという 2 空間コピーの欠点が問題となることが分かった．この問題を解決するために，ヒープの使用率が高いときはマークスイープ GC に切り替える方式が考えられる．この方式の実装と評価は今後の課題である．

また，提案手法のメモリ管理ではフリーリストなど，ほとんどの機能がブロックごとに独立している．そのため，ブロックごとにオブジェクトのサイズを変えることも容易にできる．これを利用して，本来 5 ワードのデータを持たないにもかかわらず，他のオブジェクトとサイズを揃えるために 5 ワードに切り上げられた型のオブジェクトを専用のブロックに

© 2009 Information Processing Society of Japan

割り当てることで，内部フラグメンテーションを避けられると考えられる．この手法も今後検討する課題となっている．

本研究により，Ruby のオブジェクトを移動させることが可能となったので，キャッシュを考慮したオブジェクトの再配置[13),14)] や，世代別 GC などへの応用も期待できる．

9. ま と め

本研究では，Ruby VM に mostly-copying GC を実装した．Ruby VM に実装するにあたって，Bartlett のアルゴリズムではあいまいなルートに含まれるポインタが多いとメモリ効率が悪くなるため，そのような場合でも性能低下が少なくなるように mostly-copying GC のアルゴリズムを変更した．さらに，移動できないオブジェクトをヒープ全体に分散させない工夫なども行った．これにより，多くのハードウェアページにオブジェクトがない状態を作り出し，そのようなページのメモリを OS に返すようにした．

これらを実装して，一時的に多くのオブジェクトを扱うプログラムで，ヒープの使用率が下がったときには，ヒープの一部を解放してプロセスのワーキングセットを小さくすることができた．また，提案手法を実装した VM は従来の VM と比べ平均して 2%ほどの速度低下が見られ，この主な原因は GC 回数の増加によるものと分かった．

本研究ではワーキングセットを削減することを目的にオブジェクトを移動させたが，オブジェクトを移動させる GC には，キャッシュを考慮したオブジェクトの再配置や世代別 GC などへの応用も考えられる．

謝辞　本研究は科研費（20700023）の助成を受けたものである．

参 考 文 献

1) 和田　勝：プログラミング Ruby 第 2 版言語編，株式会社オーム社 (2006).
2) Boehm, H.J. and Weiser, M.: Garbage collection in an uncooperative environment, *Software — Practice and Experience*, Vol.18, No.9, pp.807–820 (1988).
3) Bartlett, J.F.: Compacting garbage collection with ambiguous roots, *ACM SIGPLAN Lisp Pointers*, Vol.1, No.6, pp.3–12 (1988).
4) 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.42, No.SIG2(PRO28), pp.57–73 (2006).
5) Kelsey, R., Clinger, W. and Rees, J. (Eds.): Revised^5 Report on the Algorithmic Language Scheme, *ACM SIGPLAN Notice*, Vol.33, No.9, pp.26–76 (1998).
6) Bartlett, J.F.: Scheme->C: A Portable Scheme-to-C Compiler, Technical Report, DEC Western Research Laboratory (1989).
7) Cheney, C.J.: A nonrecursive list compacting algorithm, *Comm. ACM*, Vol.13, No.11, pp.677–678 (1970).
8) Attardi, G. and Flagella, T.: A customisable memory management framework, *Proc. 1994 USENIX C++ Conference*, pp.123–142 (1994).
9) Smith, F. and Morrisett, G.: Comparing mostly-copying and mark-sweep conservative collection, *Proc. 1st International symposium on Memory management* (*ISMM'98*), pp.68–78 (1998).
10) 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.SIG3(PRO10), pp.40–48 (2001).
11) Bartlett, J.F.: Mostly-Copying Garbage Collection Picks Up Generations and C++, Technical Report, DEC Western Research Laboratory (1989).
12) Hughes, R.J.M.: A semi-incremental garbage collection algorithm, *Software — Practice and Experience*, Vol.12, No.11, pp.1081–1084 (1982).
13) 八杉昌宏，伊藤智一，小宮常康，湯淺太一：階層的グループ化に基づくコピー型ごみ集めによる局所性改善，情報処理学会論文誌：プログラミング，Vol.45, No.SIG5(PRO21), pp.36–52 (2004).
14) Yasugi, M. and Yuasa, T.: Improving Search Speed on Pointer-Based Large Data Structures Using a Hierarchical Clustering Copying Algorithm, *Post-proceedings of the International Workshop on Innovative Architecture for Future Generation Processors and Systems 2007* (*IWIA 2007*), pp.43–52 (2007).

(平成 20 年 9 月 28 日受付)
(平成 20 年 12 月 13 日採録)

鵜川　始陽（正会員）

昭和 53 年生．平成 12 年京都大学工学部情報学科卒業．平成 14 年同大学大学院情報学研究科通信情報システム専攻修士課程修了．平成 17 年同専攻博士後期課程修了．平成 17 年京都大学大学院情報学研究科特任助手．平成 20 年より電気通信大学助教．博士（情報学）．プログラミング言語とその処理系に関する研究に従事．

© 2009 Information Processing Society of Japan