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

中 村 成 洋[†1] 松 本 行 弘[†1]

Apache HTTP サーバの元で提供される Web サービスのようなひんぱんにサブプロセスが生成される環境下では，現在 Ruby が採用しているマークビットをオブジェクトヘッダ内に持つマークスイープ方式のゴミ集め実装は，copy-on-write によるメモリページ共有を疎外し，必要以上にメモリを消費する．本研究では，マークをオブジェクトヘッダ内から独立したビットマップに移すことによるメモリ消費量実行性能の変化を示す．効率的なビットマップマーキングのためにはオブジェクトポインタからビットマップの特定の位置の計算が必要である．一般的にはビットマップ位置をたとえば 1MB バイト単位でアラインしておきアドレスをマスクすることでビットマップ位置を求めることが行われている．しかし，Ruby のように種々のプラットフォームで動作する言語処理系では，利用できるメモリ割り当て API は malloc() しかないため，効率良くアラインされたメモリを確保する方法は知られていない．本研究では Ruby のオブジェクト配置方式を利用し，移植性を維持したまま，オブジェクトポインタから定数時間でビットマップ位置を計算する手法を提案する．また，二分検索を用いたビットマップマーキングと比較したマーキング性能の改善も示す．

Ruby Garbage Collection Using Efficient Bitmap Marking

Narihiro Nakamura[†1] and Yukihiro Matsumoto[†1]

Since mark-and-sweep garbage collection scheme, which Ruby interpreter uses modifies every living object, it suffers performance problems due not to utilize copy-on-write memory page sharing among processes, under the circumstances like web-services running under Apache HTTP servers. In this paper, we proposes adding bitmap marking for Ruby's garbage collection. We show much memory usage and performance change by bitmap marking. For efficient bitmap marking, it is needed to calculate bitmap position from an object pointer. Prior art uses aligns heap memories and pointer masking to retrieve bitmap position from object pointer. But Ruby interpreter runs on various platforms, and we do not have portable memory allocation API to obtain aligned memory region without wasting region. In this paper, we propose portable scheme to map from object pointers to corresponding bitmap table in constant time. We also show how much proposed bitmap marking improves performance, comparing bitmap marking method using binary search to obtain bitmap position from object pointers.

1. はじめに

近年，スクリプト言語は様々なシーンで見かけるようになった．これは，スクリプト言語が技術者にとって扱いやすく，生産性が高いためである．[1)2)]

現在，多くの Web アプリケーションはスクリプト言語で記述されている．多くのユーザを抱える大規模な Web アプリケーションが Perl，Ruby[3)] の様なスクリプト言語で書かれているというのは珍しくない．これからも，多くの Web アプリケーションはスクリプト言語で書かれるであろう．

しかし，その場合，問題点は多くある．その一つがメモリの大量消費である．常駐する Web サーバの様なプロセスでは，省メモリ化は重要な要素である．そこで筆者らはスクリプト言語である Ruby に注目し，省メモリ化に取り組んでいる．

Web アプリケーションにおいて，メモリを大量に消費する原因は，言語内のガーベジコレクション (以下 GC) が一つの原因であると考える．

現在，Ruby では GC にシンプルなマークスイープ方式を採用している．これは，オブジェクトの参照ルートから，生きている全てのオブジェクトに 1bit の印付けをし，最後に印付けがされていないオブジェクトを解放する GC のアルゴリズムである．

しかし，この方式は copy-on-write[4)] によるメモリ共有を阻害する問題がある．通常 Linux ではサブプロセス生成時に，メモリ空間をすべてコピーしない．多くのメモリ空間を親プロセスと共有する．共有されて

†1 (株) ネットワーク応用通信研究所
Network Applied Communication Laboratory Inc.

いるメモリ領域が，サブプロセスにコピーされるタイミングは，そのメモリ領域に書き込みがあったタイミング (変化する際) である．これを， copy-on-write(以下 CoW) と呼ぶ.

つまり，オブジェクトに対して，印付けを行うマークスイープ方式は， CoW と非常に相性が悪い．なぜなら，マークビットの操作によりそのページ全体がコピーされるからである．この事により，ApacheHTTP サーバ[5] の様な頻繁にサブプロセスを生成する環境の元では，うまくメモリ共有できず、多くのメモリ空間が無駄に確保される事になる．

これを軽減する手法がビットマップマーキング[6] である．この手法では，オブジェクトに直接マークせず，ビットマップと呼ばれる専用の領域にオブジェクトをマッピングし，そこにマークを行うことで，メモリをコンパクトに使用し，メモリ共有を阻害しにくいという利点がある．

ビットマップマーキングでは，マークする際，オブジェクトのポインタからビットマップ位置を探索しなければならない．この処理は，一定時間であり，かつ高速であることが求めらる．なぜなら，マーキング処理は生きているオブジェクトすべてに対して行われる処理であり，処理系にとってボトルネックの部分であるからだ．

また，Ruby はさまざまなプラットフォームで動作している．この様に多くのプラットフォームで動作するアプリケーションでは，プラットフォームに依存しないポータブルなビットマップ位置探索手法が求められる．

本研究では，その様な要求を満たすビットマップ位置探索手法を提案する．また，それを利用したビットマップマーキングを Ruby に実装する．

2. Ruby のメモリ管理

この章では現在の Ruby のメモリ管理方法と問題点について簡単に述べる．

2.1 Ruby のマークスイープ方式 GC

Ruby では，生成されるオブジェクトや文字列などのデータは Ruby で独自に管理しているヒープ領域に確保される．そのため，GC はその領域を対象として行われる．

以下に Ruby のメモリ管理の大まかな動作を記述する．また，Ruby(1.9.0-4) のヒープ構造を**図 1** に示す.

(1) malloc にて 16KB のメモリ領域を確保し，オブジェクトの配列 (以下ヒープスロット) を作成する．現在，オブジェクトのサイズはすべて 20byte である．

(2) 確保した 16KB のメモリ領域で，オブジェクトサイズの倍数のアドレスからヒープスロットとして使用する．つまり，20byte でアラインを行う．これにより注意すべき点は，確保したメモリのアドレスによっては，配列内に定義されるオブジェクトの個数が 1 つ減る事があるという点である (図 1).

(3) ヒープスロットを構築したのちに，フリーリストと呼ばれるリストに配列内の要素をリンクしておく．処理系から使用割り当ての要請があった際にフリーリストから，一つだけオブジェクトを取り出し，初期化を行う．

(4) フリーリストが尽きると，マーク処理を開始する．マシンスタックや CPU レジスタなどの領域からオブジェクトへの参照であるルートを発見し，そのルートからオブジェクトの参照関係を再帰的にマークする．マークキングではオブジェクト内の flags のマークビットを 1 にしている．これで現在利用しているオブジェクトにはすべて印付けが行われた事になる．

(5) スイープ時にはヒープ内のをすべてのオブジェクトを探索し，マークされていたオブジェクトは，マークビットを 0 に戻し，マークされていないオブジェクトは回収し，フリーリストにつなぐ．

これでマークスイープは終了となる．

このように Ruby ではシンプルな保守的なマークスイープ[7] が実装されている．

2.2 マークスイープによるメモリ共有の阻害

近年，簡単に素早く Web アプリケーションを作成できるという特性をもった，RubyOnRails[8] という Ruby の Web フレームワークが誕生し，Web サービスで Ruby を使用する例が非常に多くなっている．

その場合，多くの Web サービスでは，HTTP サーバに ApahceHTTP サーバを使用している．Apache は安定しており，性能も優れた HTTP サーバであるからだ．しかし，Apache と Ruby を組み合わせると，ある問題が発生する．

Apache では，クライアント通信を行う際に，頻繁にサブプロセスの生成を行う．Linux カーネルではサブプロセス生成時に，親プロセスで使用していたメモリ空間を複製しない．親プロセスにて使用していたメモリ空間をすべて共有し，書き込みが発生した場合

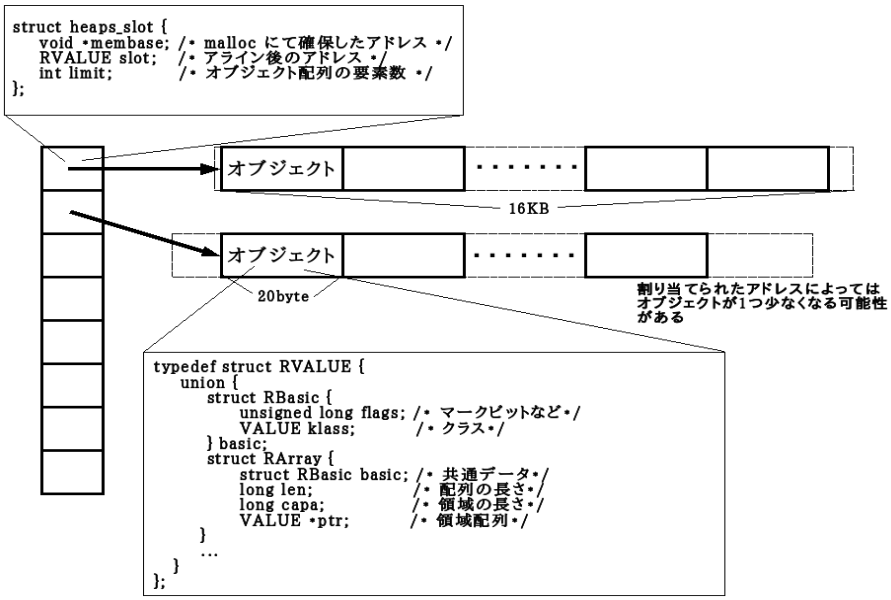

図 1 Ruby のヒープ構造
Fig. 1 Heap struct in Ruby

に初めてサブプロセスに複製される．これが CoW である．

しかし，この CoW と現在 Ruby のマークスイープ GC とは非常に相性が悪い．なぜなら，すべての生きているオブジェクトのマークビットを立てる際に，共有されているページに書き込みを行い，サブプロセスへの複製が発生するからである．

これを改善する手法としてビットマップマーキングという手法がある．オブジェクトのマークビットのみを別の領域 (ビットマップ) に移し，オブジェクトの生存，死亡はそのビットマップにて管理する手法である．この手法を使用すると，マーク時に直接オブジェクトを扱うことがなくなり，無駄な複製を抑える事ができる．

この手法で Ruby の GC を改善する事によって，CoW にうまく対応し，サブプロセス動作時のメモリ消費量を抑える事が可能になる．つまり，Apache の環境で動く，Ruby を使用した多くの Web アプリケーションが省メモリで動作するようになる．

3. Ruby へのビットマップマーキングの実装

この章では，従来のビットマップ位置の探索の手法と，本研究で提案する新しい手法を示す．

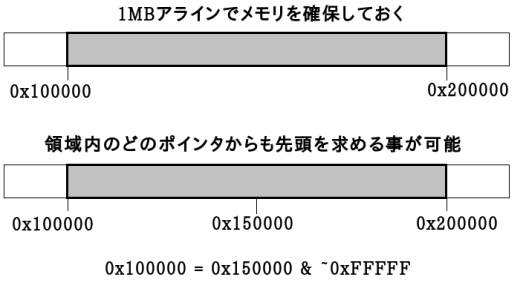

図 2 アラインされたメモリ領域を利用した定位置探索
Fig. 2 The fixed position search by aligned memory area

3.1 アラインされたメモリ領域の確保によるビットマップ位置の探索

ビットマップマーキングを行う際，オブジェクトのポインタからビットマップの位置を探索する必要がある．

Ruby において，もっとも簡単な方法は各オブジェクトにビットマップへのポインタを持たせることである．しかし，これではオブジェクトのポインタのフィールド分メモリ領域を圧迫することになり，今回の目的である省メモリ化の意味がなくなる．

オブジェクトにフィールドを追加せずに，ビットマップ位置を探索する手法の一つとして，アラインされたメモリ領域を利用する方法がある．

```
static inline struct heaps_slot *
find_heap_slot_for_object(RVALUE *object)
{
  int i;
  for (i = 0; i < heaps_used; i++) {
    struct heaps_slot *heap = &heaps[i];
    if (object >= heap->slot
      && object < heap->slot + heap->limit)
      return heap;
  }
  return NULL;
}
```

図 3 線型探索によるビットマップ位置探索
Fig. 3 BitMap position search by liner

例えば，1MB アラインで確保したメモリ領域内にオブジェクトを配置し，確保したメモリ領域の先頭にはビットマップへのポインタを設定しておく．そして，ビットマップ位置を探索する際には，オブジェクトポインタの下位 20bit を 0 でマスクし，アラインされたメモリ領域の先頭の位置を求める．(**図 2**)

この手法は，ヒープスロットの個数は関係がない．O(1) のアルゴリズムである．

ビットマップ位置探索は前述した通り，処理系にとってボトルネックになる部分である．この手法は，高速に動作するという面では，今回の要求をクリアしている．

さて，問題はアラインしたメモリ領域の確保方法である．Linux の場合，アラインされたメモリ領域の確保には，posix_memalign を使用する．posix_memalign は POSIX[9] 準拠の API である．指定したサイズでアラインされたポインタを返却する働きをする．

しかし，Ruby は Linux 以外でも様々な環境で動作する．アラインされたメモリ領域の確保は，環境に依存した手法しかなく，多くのプラットフォームで動作するアプリケーションには適用できない．その様なアプリケーションでは，この手法ではなく，プラットフォームに依存しないビットマップ位置探索手法が必要である．

3.2 ポータブルなビットマップ位置の探索

Ruby へビットマップマーキングを実装した前例として，Hongli Lai らによる RubyEnterpriseEdition(以下 REE)[10] という処理系がある．

彼らは，その REE を ApacheHTTP サーバ環境の元で動作する RubyOnRails アプリケーションに適用し，そのアプリケーションについて計測を行っている．

その結果として，リクエスト毎秒 22 %が高速化し，総メモリ使用量が 31 %削減したと報告した．[11]

```
static inline struct heaps_slot *
find_heap_slot_for_object(
  rb_objspace_t *objspace,
  RVALUE *object)
{
  int i;
  register size_t hi, lo, mid;

  lo = 0;
  hi = heaps_used;
  while (lo < hi) {
    mid = (lo + hi) / 2;
    struct heaps_slot *heap = &heaps[mid];
    if (heap->slot <= object) {
      if (object < heap->slot + heap->limit) {
        return heap;
      }
      lo = mid + 1;
    }
    else {
      hi = mid;
    }
  }
  return NULL;
}
```

図 4 二分によるビットマップ位置探索
Fig. 4 BitMap position search by binary

この改善は，ビットマップマーキングにより，多くのメモリが共有できた事によるものだ[12] と，彼らは考えている．ApacheHTTP サーバによって，頻繁に生成されるサブプロセスと，親プロセス間で，うまくメモリが共有され，省メモリで各プロセスが動作する事で Web アプリケーションの性能が向上したという事である．

REE では，ビットマップ位置探索に，ポータブルな手法として線型探索を使用している． 各ヒープスロットを，先頭から一つずつ，オブジェクトのポインタが，どのヒープスロットに属するのか比較していくという非常にシンプルな実装である．実際のコードを**図 3** に示す．

また，類似したもう一つの案として，二分探索[13] の方法がある．図 1 に示している各ヒープスロットは，現在二分探索の為，アドレスの若い順に格納されている．これを利用し，オブジェクトポインタを元に，探索を行う．実際のコードを**図 4** に示す．

二分探索は，Ruby のヒープスロットの数に比例して処理時間は O(log2 n) で増加する．

また REE に至ってはビットマップ位置検索に線型探索を使用している．処理時間はヒープスロットによって O(n) で増加する．これは二分探索よりも処理速度が遅い．

これら，二つの手法は，プラットフォームに依存し

ないポータブルな手法であるが，処理速度が遅い．ビット マップ位置探索はオブジェクト をマークする際に必ず行う 処理である．このボト ルネックの箇所に O(log2 n) や，O(n) のアルゴリ ズムを毎回使用すること は，処理系にとって大きな負荷になる．

効率的なビット マップマーキングを実現するためには，高速でビット マップ位置を探索する事は，非常に重要な要素といえる．また，このビット マップ位置探索を O(1) のアルゴリ ズムに変更することで，前述したよう な実用的な Web アプリ ケーション は，更に性能が向上するはずである．

3.3 Ruby のヒープ構造を利用した新規探索手法

本研究では Ruby における，ポータブルな探索手法であり，かつ O(1) のアルゴリ ズムとして以下を提案する．

Ruby のヒ ープスロット はそれぞれ 16KB で確保されている．16KB とは，2 の 14 乗である．つまり，ヒ ープスロット 内には，下位 14bit が 0 である箇所が，先頭か，末尾か，どこかに存在するはずである．その場所にビット マップへのポインタを定義すれば，オブジェクト のポインタの下位 14bit をマスクすることで，目的のビット マップ領域が探索可能と なるはずである．

だが，これではオブジェクト のアドレスが，下位 14bit でマスクしたアドレスより 小さい場合，ヒ ープスロット 外の違う 場所をさすことになる．そこで，ヒ ープスロット の開始位置から，下位 14bit が 0 であるアドレスまでのオブジェクト に，フラグを立てる．(**図 6**) このフラグには，ビット マップマーキングの適用により，使用されなくなったマークビット を使用した．マーキングの際には，このフラグを利用し，オブジェクト の位置を判断し，ビット マップ位置の計算を行う．

さらに，ヒ ープ内はオブジェクト のサイズによってアラインされている．その為，位置計算の際にはオブジェクト ポインタの下位 14bit をマスクしたアドレスを，オブジェクト のサイズでアラインする．**図 5** に実際に行う C のコード を示す．

また，Ruby の場合，ヒ ープスロット を拡張する必要がある．

この手法では，必ず一つだけ，下位 14bit が 0 であるアドレスを含むオブジェクト が，そのヒ ープスロット 内に存在しなければならない．しかし，現在，Ruby ではヒ ープスロット をオブジェクト サイズでアラインする際に，ヒ ープスロット 内に存在するオブジェクト の個数が変動する事は，1 章にて前述した通り である．つまり，位置計算して求めたオブジェクト が削られる

```
if (((RVALUE *)p)->as.free.flags & FL_ALIGNOFF)
{
  res = (RVALUE *)((((VALUE)p & ~0x3FFF) + 0x4000) /
    sizeof(RVALUE) * sizeof(RVALUE));
}
else {
  res = (RVALUE *)(((VALUE)p & ~0x3FFF) /
    sizeof(RVALUE)*sizeof(RVALUE));
}
```

図 5 Ruby におけるアライン手法でのビット マップ位置探索
Fig. 5 Bitmap place search by Align method in Ruby

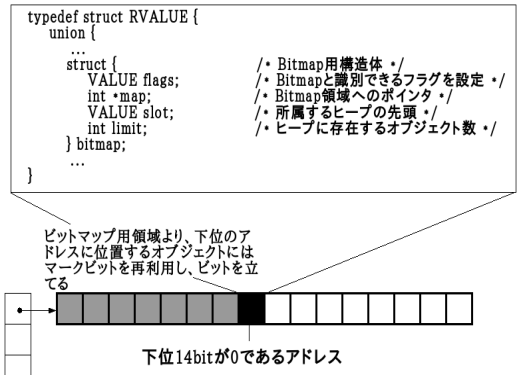

図 6 ヒ ープスロット の初期化
Fig. 6 Initialize heap slot

```
(0x4000 /sizeof(RVALUE) + 2) * sizeof(RVALUE);
```

図 7 ヒ ープスロット サイズの計算
Fig. 7 Calculation of the heap slot size

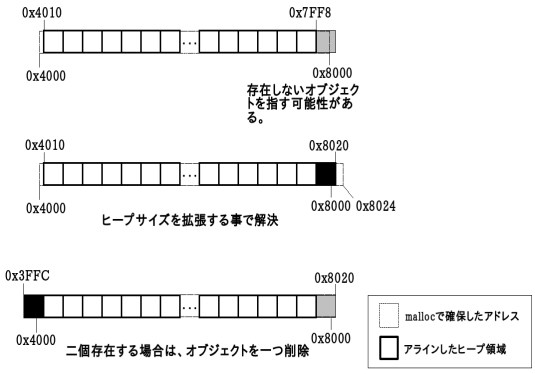

図 8 ヒ ープスロット サイズの調整
Fig. 8 Adjustment of the heap slot size

可能性があるのだ．(**図 8**) そこで，今回は **図 7** 様なコード でヒ ープスロット のサイズを少し 拡張した．

また，綺麗に 20byte でアラインされたアドレスのヒ ープスロット が確保された場合，ビット マップへのポインタが二つになってしまう 為，このケースにおい

```
obj_num = HEAP_SIZE / sizeof(struct RVALUE) - 1
size = sizeof(int) * (obj_num / (sizeof(int) * 8)+1);
map = (int *)malloc(size);
```

図 9 ビットマップ領域の確保
Fig. 9 Allocate bitmap area

```
obj_index = (obj_p - slot) / sizeof(RVALUE);
index = obj_index / (sizeof(int) * 8);
offset = obj_index % (sizeof(int) * 8);
map[index] |= 1 << offset;
```

図 10 ビットマップへのマーキング
Fig. 10 Marking to bitmap

ては意図的に一つオブジェクトを削る事とする．これにより，ヒープスロット内にず一つだけ，ビットマップへのポインタが確保される事が保証された．(図 8)

この探索手法ではヒープスロットの個数は関係ない．O(1) のアルゴリズムである．よって，ボトルネックであるマーク処理内において，高速かつ，安定して動作する．

また，posix_memalign の手法とは違い，プラットフォーム依存が少ない．よりポータブルな手法であり，Ruby のように多くのプラットフォームで動作するアプリケーションに，適用可能である．

3.4 Ruby へのビットマップマークスイープ実装

本手法を適用した Ruby 処理系の，ビットマップマーキングからスイープまでの手順を以下に記す．処理内容に変更の無い箇所については，説明を割愛する．

(1) ビットマップは int 型の配列で作成する．作成した配列のサイズ (bit) は，ヒープスロット内に存在するオブジェクトの個数分確保する．
C のソースコードを**図 9** に示す．

(2) マーク時には，オブジェクトのポインタから図 5 を使用し，ビットマップ位置を探索を行う．

(3) ビットマップにマーキングする際，ヒープスロットからマーク対象のオブジェクトまでのインデックスを求める．そのインデックスを int のサイズで除算し，マーキング位置のインデックスを求める．また，int のサイズで剰余演算を行い，マーキング位置へのオフセットを求める．この二つの値を元に，ビットマップ位置へマーキングを行う．
C のソースコードを**図 10** に示す．

(4) スイープ時に，フリーリストにオブジェクトをつなぎ直す際，元々フリーリストに繋がっており，使用されていないオブジェクトに関しては，何も操作せず，そのままにしておく．
これは，極力，CoW されないようにする為である．

(5) 一つのヒープスロットをスイープした後，マークを消去する為，ヒープスロットに対応したビットマップ領域を 0 でクリアする．

(6) すべてのヒープスロットに対してスイープを行い，GC を終了する．

この様に Ruby 処理系に，本研究で提案したビットマップ探索手法を利用した，ビットマップマーキングを実装した．

4. 性能評価

この章では新手法の性能を評価する．評価では以下に示す 3 種類の Ruby 処理系で同一のプログラムを実行することで行う．

- `orign`
  従来の Ruby 処理系．バージョンは 1.9.0-4 である．
- `btree`
  Ruby1.9.0-4 にビットマップマーキングを適用した処理系．ビットマップの探索には二分探索を使用している．
- `align`
  Ruby1.9.0-4 にビットマップマーキングを適用した処理系．ビットマップの探索には本研究で紹介した手法を使用している．

計測は IntelCoreDuo(1.83GHz, 2MB L2 キャッシュ, 667MHz FSB)，メモリ 2 GB の計算機で行った． OS は Linux2.6.24，コンパイラは gcc 4.2.3 を用いた．

4.1 skk.rb

skk.rb[14] は SKK サーバ[15] の様な動きをする．起動時に，辞書ファイルをハッシュに詰め込み，自身をデーモン化する．今回，郵便番号と住所の辞書ファイル[16] を使用する事とする．

その後，5 個のサブプロセスを生成し，親プロセスで TCP での通信を待つ．クライアント側では，SKK サーバが待っているポートに郵便番号を TCP プロトコルにて送信する．その通信を親プロセスがキャッチし，生成しておいたアイドリング中のサブプロセスに処理を渡す．サブプロセスは自身のもつハッシュにて郵便番号を住所に変換し，結果をクライアント側に返す仕組みである．

今回の計測は，クライント数 10，それぞれのリク

表 1 GC 最大処理時間
Table 1 GC max time

処理系	処理時間	マーク時間	スイープ時間
orign	48	28	20
btree	92	72	20
align	52	36	16

単位はすべて msec

表 2 各プロセスの合計メモリ使用量
Table 2 Memory usage

処理系	合計メモリ使用量	共有メモリ使用量
orign	123956	46304
btree	50348	90836
align	51740	91152

単位はすべて KB，また，ヒープ領域内のみの使用量

表 3 サーバアクセス速度
Table 3 Server access speed

処理系	平均リクエスト終了時間 (秒)
orign	0.0090224
btree	0.0086908
align	0.0084847

エスト数 200 回で測定を行った．

表 1 では発生した GC の最大処理時間の比較を示している．

GC の最大処理時間では，オリジナルの Ruby 処理系 (orign) に比べて本研究により提案したビットマップマーキングを用いた処理系 (align) の方が 7 %程遅くなっている．また，二分探索によるビットマップマーキングを用いた処理系 (bree) では 91 %程遅い．

align の速度低下についてはビットマップを計算するインストラクションが増えた為，仕方のない範囲の劣化である筆者らは考える．また，align はビットマップ位置検索に二分探索を使用した btree よりもかなり高速に動作している事が分かる．

スイープ時間では align が，25 %程速い．ビットマップマーキングではスイープ時にマークフラグを一つ一つ消す必要がなく，ビットマップ領域を一度初期化するだけでよい．これによりスイープ処理が速くなったと思われる．

表 2 では計測後の，各 5 個のプロセスのメモリ使用の合計値を表している．

ビットマップマーキングを GC に適用する事により，総メモリ使用量が通常の半分以下に削減されていることが分かる．これは，2 章で述べた通り，CoW 向けの改善によるものである．

表 3 では SKK サーバへのリクエストから，クライアントまでのレスポンスの速度を表している．

ビットマップマーキングを適用し，メモリ消費量を抑える事によって通常よりも高速に動作している事が分かる．また，ビットマップ位置探索に，本研究で提案した手法を用いると，更に性能が向上している事がわかる．

5. まとめ

本研究では，RubyGC にビットマップマーキングを実装する事によって，サブプロセスを生成する実用的なプログラムにてメモリ使用量が 62 %削減され，メモリ消費量を抑える事に効果的である事を示した．ApacheHTTP サーバの環境の元で動作する Web アプリケーションのように，頻繁にサブプロセスを生成するアプリケーションでは，この手法を適用する事で，より少ないメモリ消費量で動作する事が可能である．

また，ビットマップ位置を探索する手法として，ポータブルで，かつ，高速に探索可能な手法を提案し，これを RubyGC に実装した．これにより，通常のマークスイープ GC よりも処理時間が 7 %遅くなった．しかし，ビットマップ位置探索に二分探索を使用した場合よりも GC 処理時間を 43 %速くする事に成功した．

参考文献

1) Ousterhout, J.: Scripting: Higher level programming for the 21st century, *IEEE Computer*, Vol.31, No.3, pp.23–30 (1988).
2) Prechelt, L.: An Empirical Comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a Search/String-Processing Program, Technical Report 2000-5, Fakultät für Informatik, Universität Karlsruhe, Germany (2000). ftp.ira.uka.de.
3) 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，アスキー出版局 (1999).
4) 高橋浩和，小田逸郎，山幡為佐久：Linux カーネル 2.6 解読室，ソフトバンククリエイティブ (2006).
5) The Apache Software Foundation: Apatch HTTP Server Project, `http://httpd.apache.org/`.
6) Jones, R. and Lins, R.: *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, Wiley (1996).
7) Hans-Juergern Boehm and Mark Weiser: Garbage collection in an uncooperative enviroment, *Software Practice and Experience*, Vol.18, No.9, pp.807–820 (1988).
8) David Heinemeier Hansson: Ruby on Rails, `http://www.rubyonrails.com/`.
9) Lewine, D.A.: *Posix Programmer's Guide*, Or-

eilly and Associates Inc (1991).
10) HongliLai and NinhBui: RubyEnterpriseEdition, http://www.rubyenterpriseedition.com/.
11) HongliLai and NinhBui: Performance and memory usage comparisons, http://www.rubyenterpriseedition.com/comparisons.html.
12) HongliLai and NinhBui: 33 % memory reduction and faster? Is this for real?, http://www.rubyenterpriseedition.com/faq.html#thirty_three_percent_mem_reduction.
13) Knuth, D. E.: *The Art of Computer Programming: Fundamental Algorithms*, Addison-Wesley, third edition (1997).
14) : 郵便番号変換簡易 skk サーバ, http://github.com/authorNari/skkzipcode/tree/master.
15) SKK Open lab: http://openlab.jp/skk/index.html.
16) SKK Open lab: http://openlab.jp/skk/skk/dic/zipcode/SKK-JISYO.zipcode.