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

Ruby Garbage Collection Using Efficient Bitmap Marking

中村 成洋, 松本 行弘

ネットワーク応用通信研究所

発表内容

1. はじめに
2. 現在のRubyGC概要
3. 問題点
4. 問題の改善
5. ビットマップ位置探索
6. Rubyのヒープ構造を利用した新規探索手法

発表内容

7. RubyGCへのビットマップマーキング実装
8. 性能評価

(1)はじめに

Rubyについて

Ruby

- オブジェクト指向スクリプト言語
- 動的型言語
- C言語による実装
- Webアプリケーションへの使用が顕著

GarbageCollection について

GarbageCollection

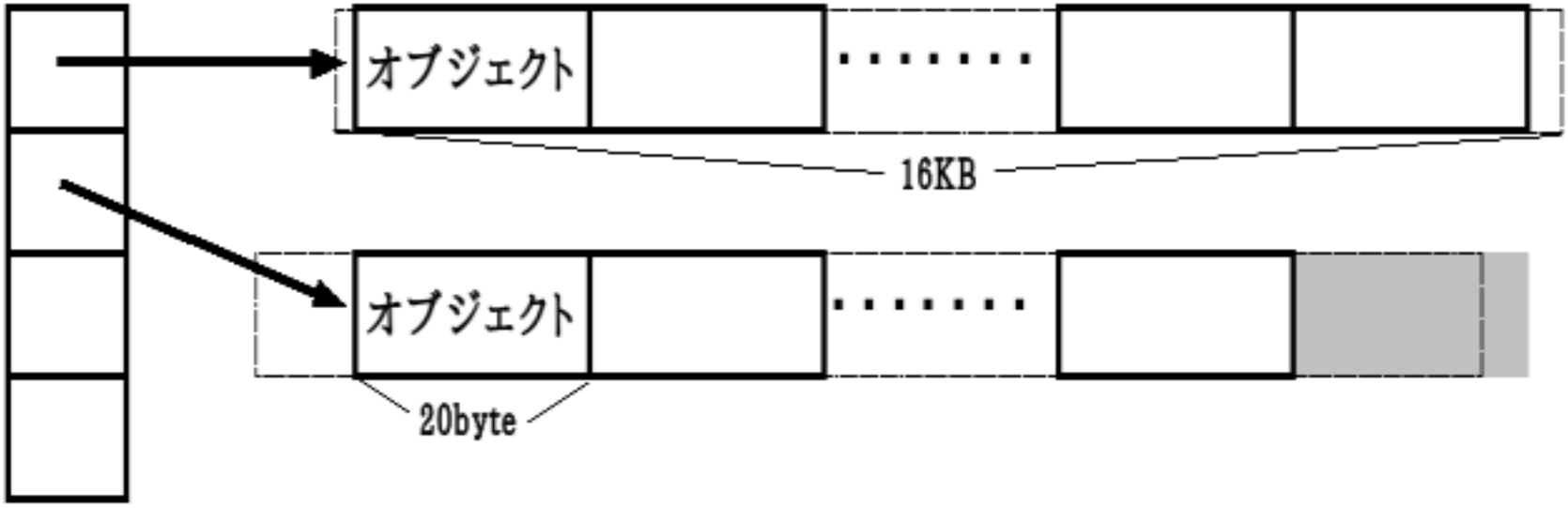
- 不要になったメモリ領域の自動的解放
- プログラマがメモリをそれ程意識しなくてもよくなった

(2)現在のRubyGC概要

現在のRubyのGC

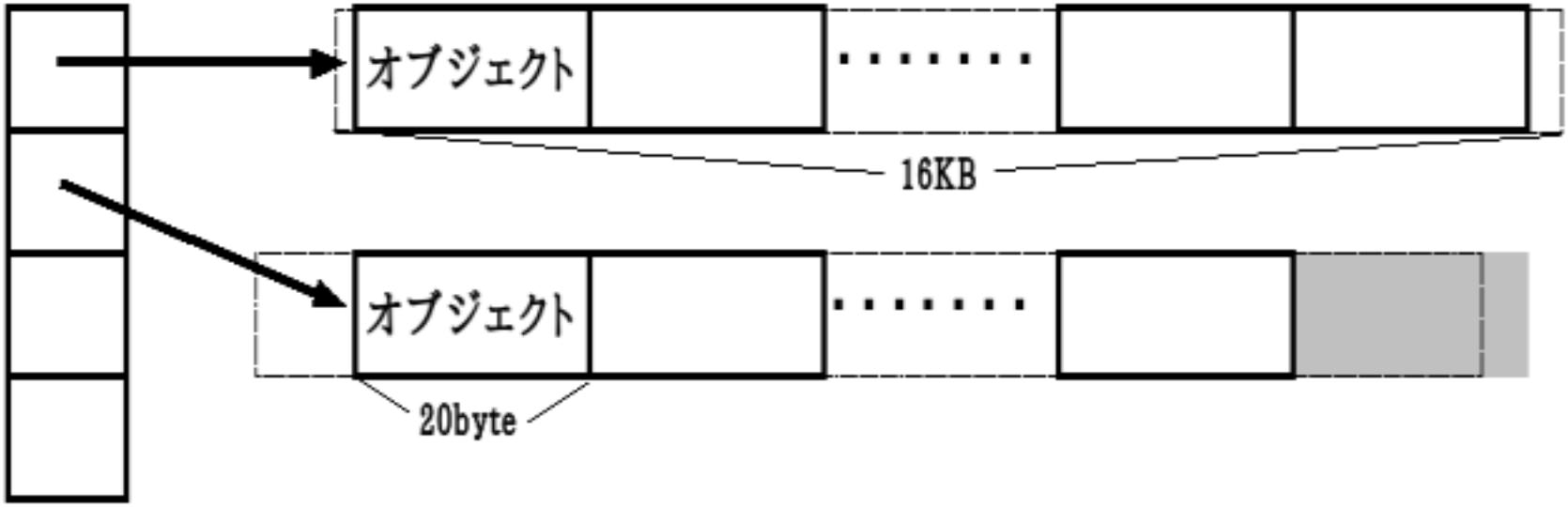
マークスイープ方式

Rubyのヒープ構造



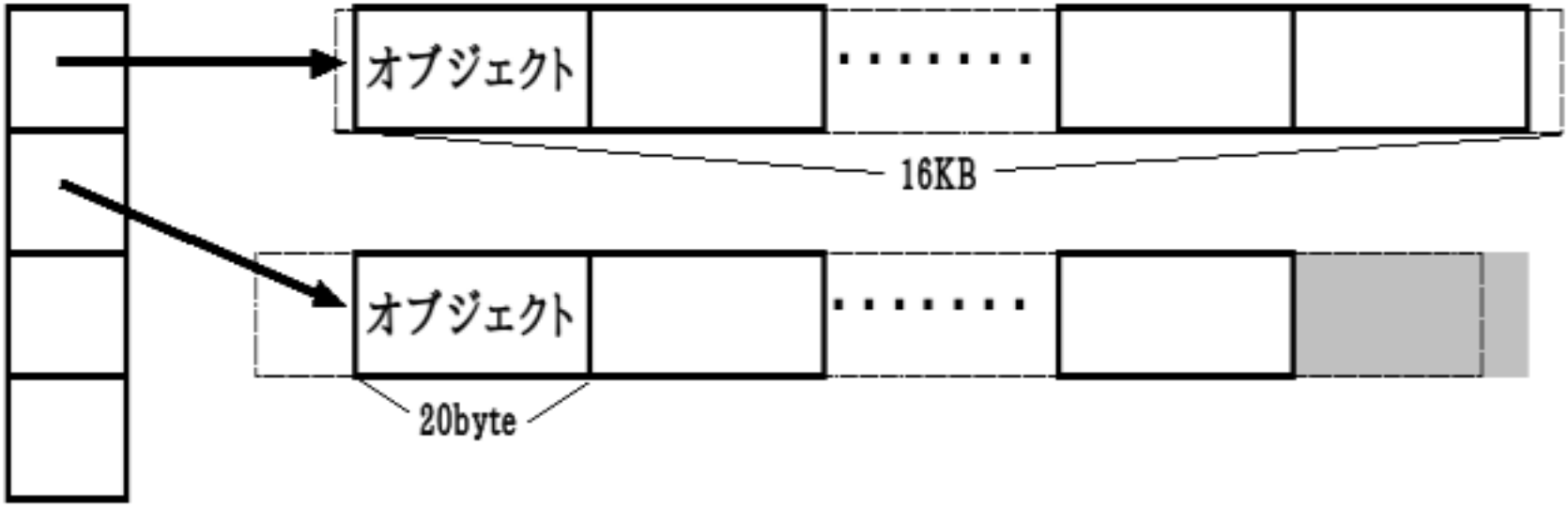
- mallocにて16KBのメモリ領域を確保
  - ヒープスロット

Rubyのヒープ構造



- オブジェクトの配列を作成
- オブジェクトサイズの20byteでアラインメント

Rubyのヒープ構造



- ヒープスロットはheaps_slotの配列で管理
- 各オブジェクトはフリーリストと呼ばれるリストにつないでおく

オブジェクトの割り当て

1. 処理系から割り当て要求
2. フリーリストからオブジェクトを一つ取り出し
3. 取り出したオブジェクトを初期化

マーク処理

1. フリーリストが尽きるとマーク処理開始
2. ルートから各オブジェクトを再帰的にマーク
3. マーク処理は各オブジェクト内フラグのマークビットを1にする
4. 全てのルートに対してマークを終えるとマーク処理終了

スイープ処理

1. ヒープ内のすべてのオブジェクトを走査
2. マークされているオブジェクトはそのマークを消す
3. マークされていないかつ使用済みのオブジェクトはどこからも参照されていないと見なし，スイープ処理を行う
4. ヒープ内を全て走査し終わるとスイープ処理終了

シンプルな
マークスイープの実装

(3)問題点

マークスイープによるメモリ共有の阻害

プロセス間のメモリ共有

- CopyOnWriteという仕組み
  - 多くのLinux環境ではサブプロセス作成時(fork)のメモリは共有領域に置かれる
  - 書き込みがあった際に，それぞれのプロセスの私有領域にコピーされる
    - この仕組みをCopyOnWriteと呼ぶ

しかし，CopyOnWirteと
RubyGCは相性が悪い

無駄なコピーの発生

- RubyGCではマークの際に生存している全てのオブジェクトに対して1bitだけ書き込みする
- コピーする必要のないオブジェクトまでサブプロセスの私有領域にコピーされてしまう

Rubyの特徴

- RubyはWebアプリケーションへの使用される事が多くなっている
- RubyOnRailsの誕生により多くのWebアプリケーションがRubyで書かれるようになった

Webアプリケーションでの問題

- Apache HTTPサーバでの使用
  - Apacheは仕組み上多くのサブプロセスを生成する
- Rubyを使うと「無駄なコピー」によって多くのメモリを消費してしまう

ここまでのまとめ

- RubyGCはマークスイープ方式を採用
- 現在のRubyGCとCopyOnWriteは相性が悪い
- Apache HTTPサーバの元で動作するWebアプリケーションでは「無駄なコピー」の分メモリが肥大化する
  - 問題点

(4)問題の改善

ビットマップマーキング という手法

ビットマップマーキング

- オブジェクトのマークビットのみを別の領域に移す
  - これをビットマップと呼ぶ
- オブジェクトの生存，死亡はビットマップにて管理
- マーク時に直接オブジェクトを操作する事がなくなる
- 無駄なコピーが発生しにくい

ビットマップマーキングの効果:前例

- RubyEnterpriseEditionというRuby実装
  - Hongli Laiらによって作成
- RubyOnRails+REE+Apacheで動作させた
  - リクエスト毎秒 22% が高速化
  - 総メモリ使用量が 31% 削減した

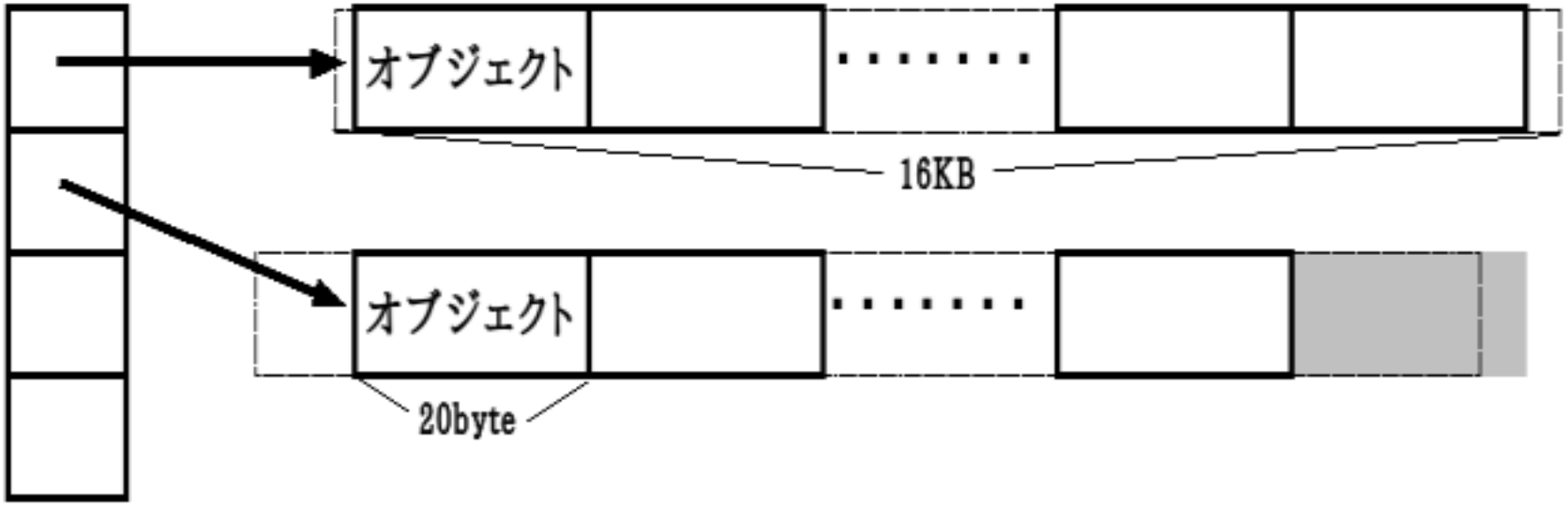
しかし

REEのビットマップマーキングの問題

- ビットマップ位置探索に線型探索を使用している
- 通常のforkしないプログラムの場合，GCがビットマップマーキング処理の分遅くなる

ビットマップマーキング実装においての課題

ビットマップ位置の探索



ビットマップ位置の探索

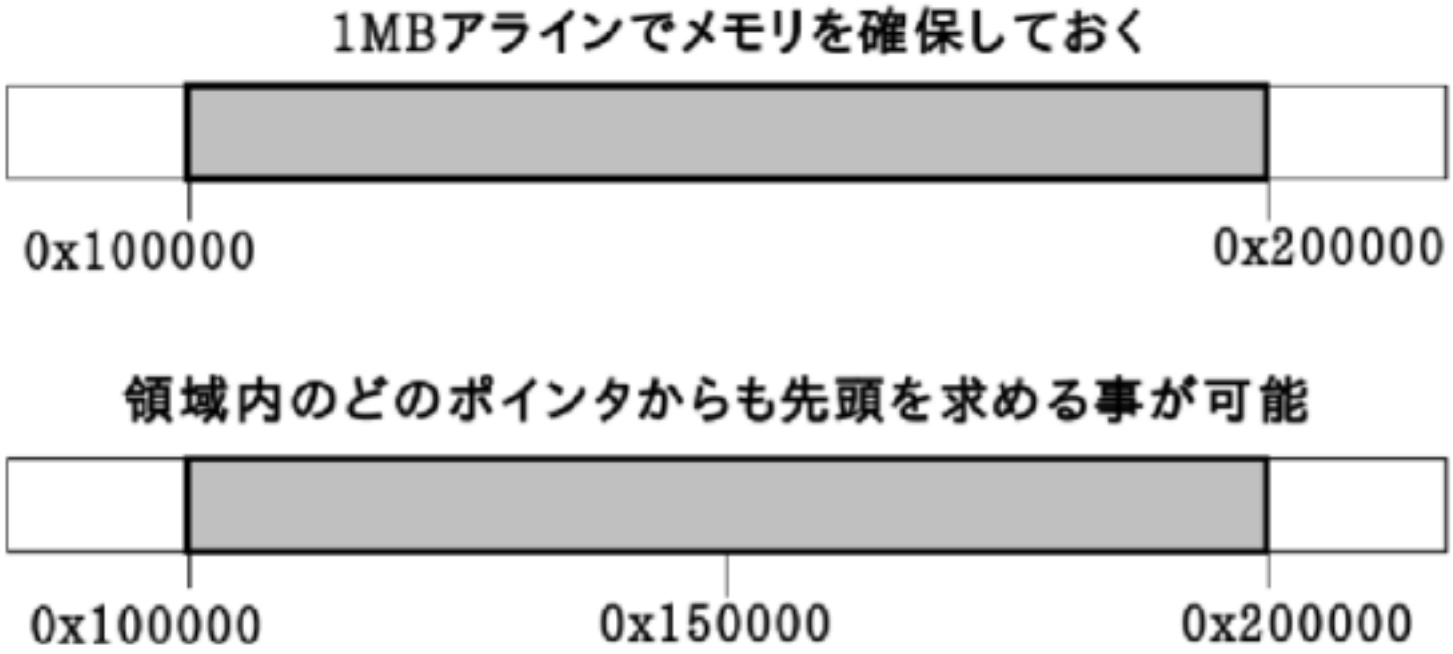
- それぞれのオブジェクトからビットマップ位置を見つける方法
- 各オブジェクトにポインタを持たせる？
  - ヒープ領域が大きくなりメモリ消費量が大きくなってしまう

(5)ビットマップ位置探索

一般的な手法

アラインされたメモリ領域による探索

アラインメモリ領域による探索



0x100000 = 0x150000 & ~0xFFFFF

メモリアラインによる位置探索

- メモリ領域をアライメントして確保
- 先頭のポインタを計算で求められる
  - このケースの場合下位19bitを0にすると求められる
- その場所にビットマップ位置へのポインタを置いておく
  - ビットマップ位置探索可能

この手法の特徴

- 長所
  - ヒープスロットの数が処理速度に関係しない O (1) のアルゴリズム
- 短所
  - アラインしたメモリ領域の確保がポータブルな手法ではない
    - Linuxの場合，posix_memalign を使用
    - しかしPOSIX準拠

Rubyに適用するのは難しい

- Rubyは多くのプラットフォームで動作するアプリケーション
- 環境に依存しないポータブルな手法でなければ適用するのは難しい

ポータブルな
ビットマップ位置の探索

線型探索

線型探索

- REEで使用されている
- 対象のオブジェクトを探索する際に，並んでいるヒープスロットを線型探索
  - ビットマップ位置探索

二分木探索

二分木探索

- 現在のRubyではヒープスロットをアドレス順に配置している
- 対象のオブジェクトを探索する際に今度は二分木探索

これら手法の特徴

- 長所
  - 様々なプラットフォームで動作可能
- 短所
  - ヒープスロットの個数により処理速度が変動する
    - 線型探索 O(n)
    - 二分木探索 O(log2 n)

Rubyに適用するのは難しい

- マーク処理は生存しているオブジェクトに対して行われる処理
- 処理系にとってボトルネックとなりやすい部分
- 毎回，線型探索などを使用した場合，GCの最大停止時間が伸びてしまう

ここまでのまとめ

- メモリ共有阻害の改善としてREEでビットマップマーキングを使用
  - 効果があった

ここまでのまとめ

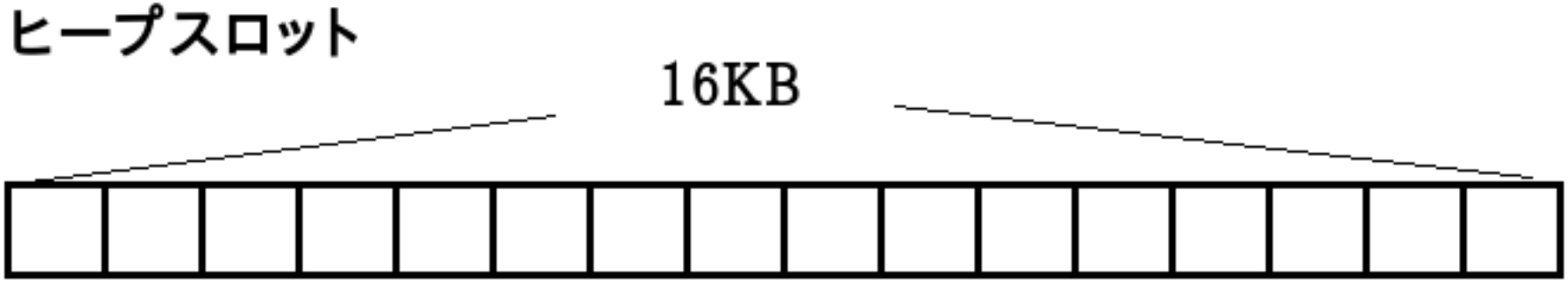
- ビットマップ位置探索の手法には
  - アラインによる位置探索
    - ポータブル性がない
  - 線型，二分木探索
    - 遅い
- Rubyに適用するのは難しい

つまり

RubyGCの
ビットマップ位置の探索は
ポータブルかつ，高速
でなければならない

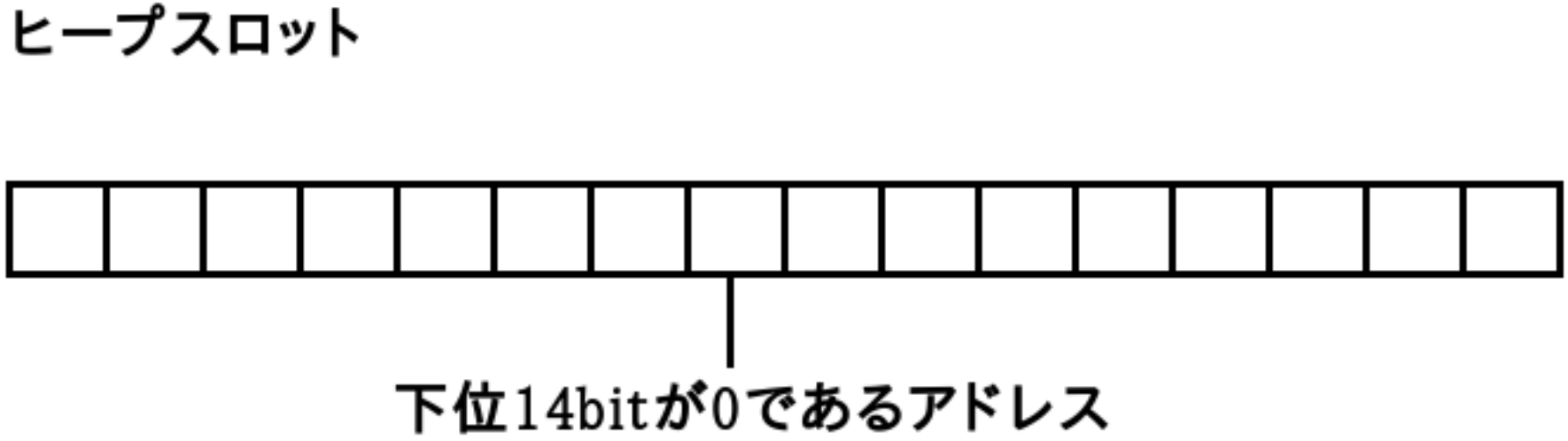
(6)
Rubyのヒープ構造を利用した新規探索手法

基本的なアイデア



- ヒープスロットサイズは16KB
  - 16KB = 2の14乗

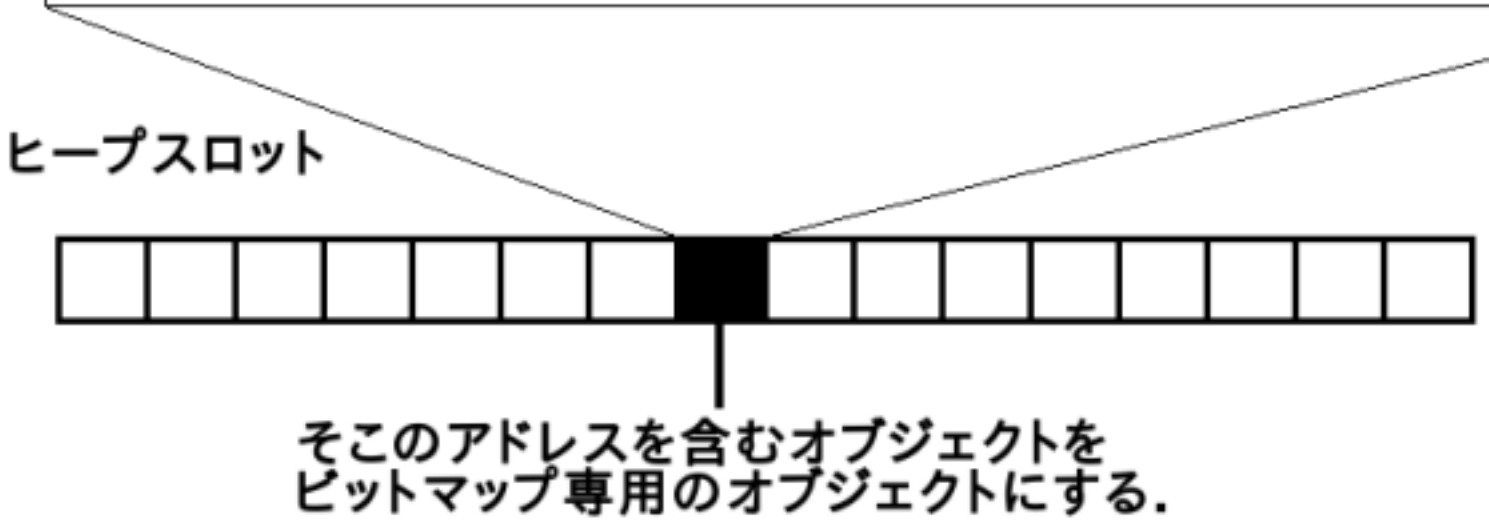
基本的なアイデア



下位14bitが全て0のアドレスがヒープスロット内のどこかに存在するはず

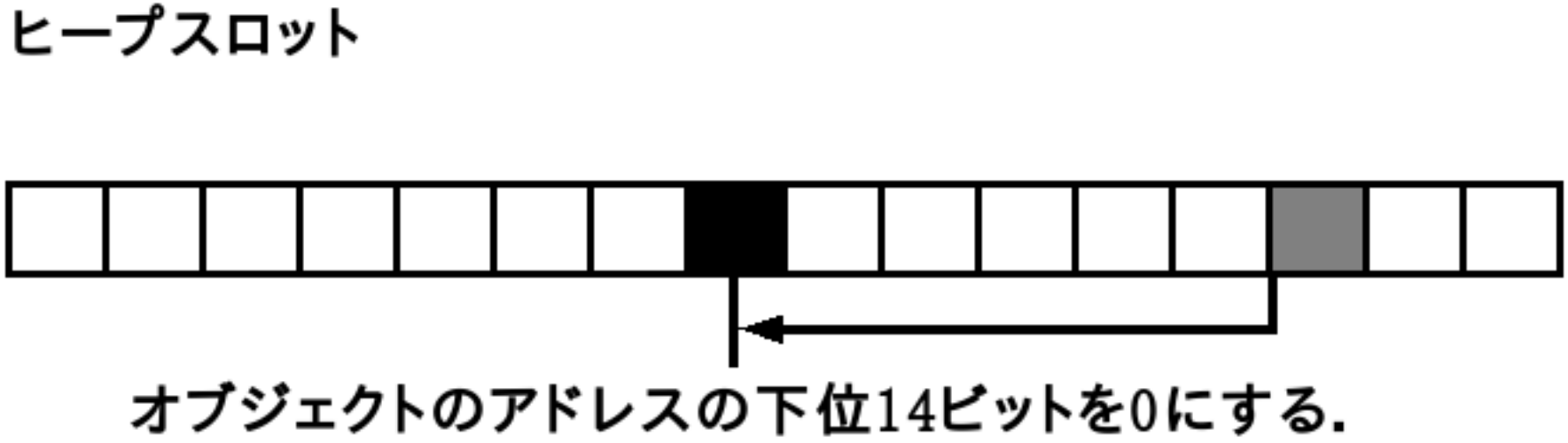
基本的なアイデア

```
typedef struct RVALUE {
    union {
        ...
        struct {                        /* Bitmap用構造体 */
            VALUE flags;                /* Bitmapと識別できるフラグを設定 */
            int *map;                   /* Bitmap領域へのポインタ */
            VALUE slot;                 /* 所属するヒープの先頭 */
            int limit;                  /* ヒープに存在するオブジェクト数 */
        } bitmap;
        ...
}
```

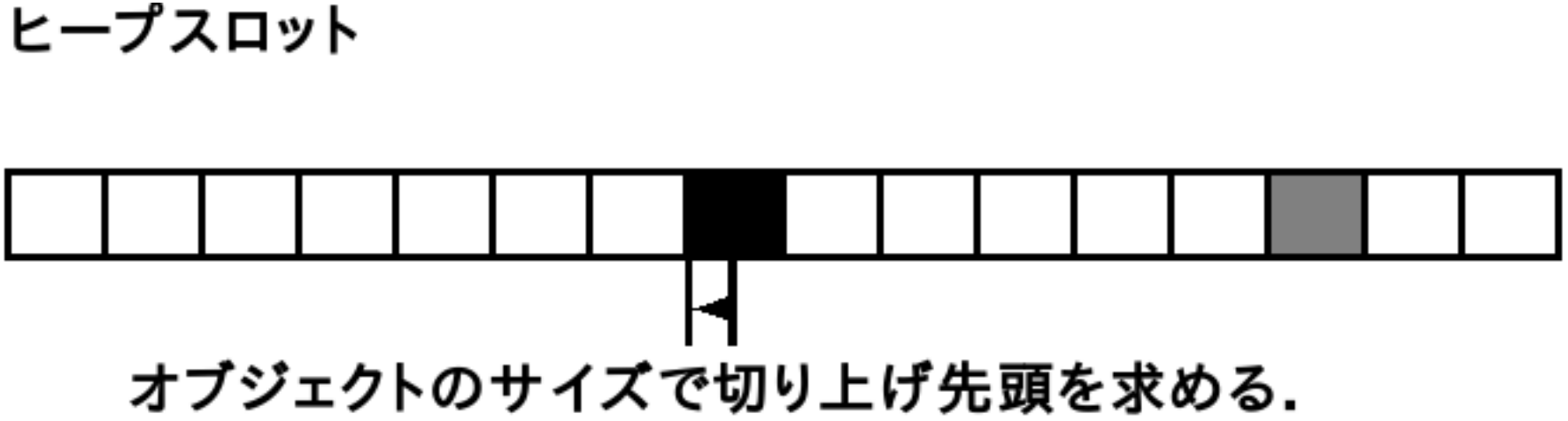


ビットマップ位置探索

ビットマップ位置探索



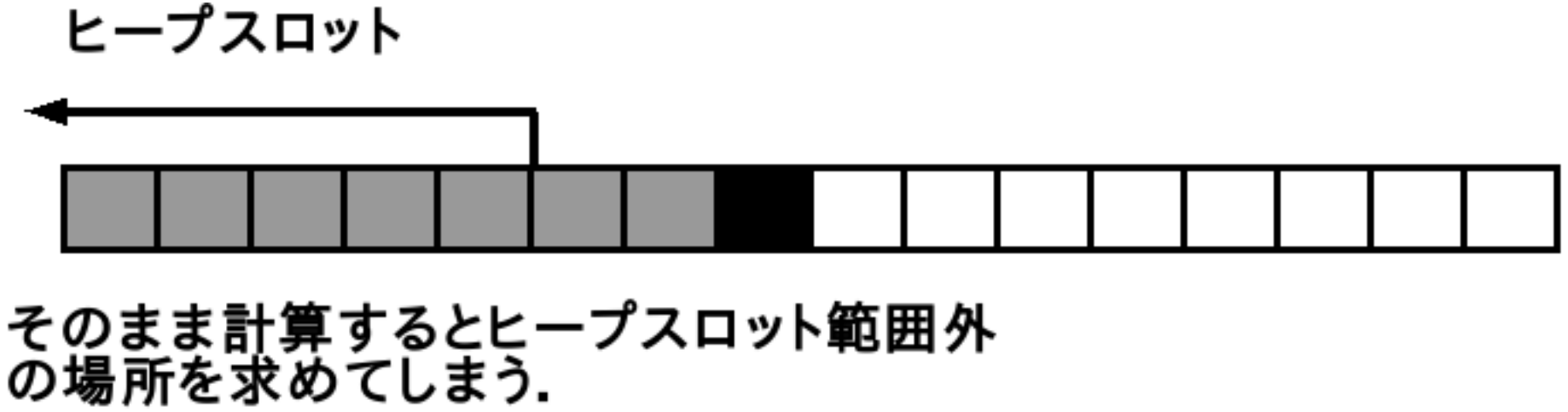
ビットマップ位置探索



ビットマップ位置が探索可能

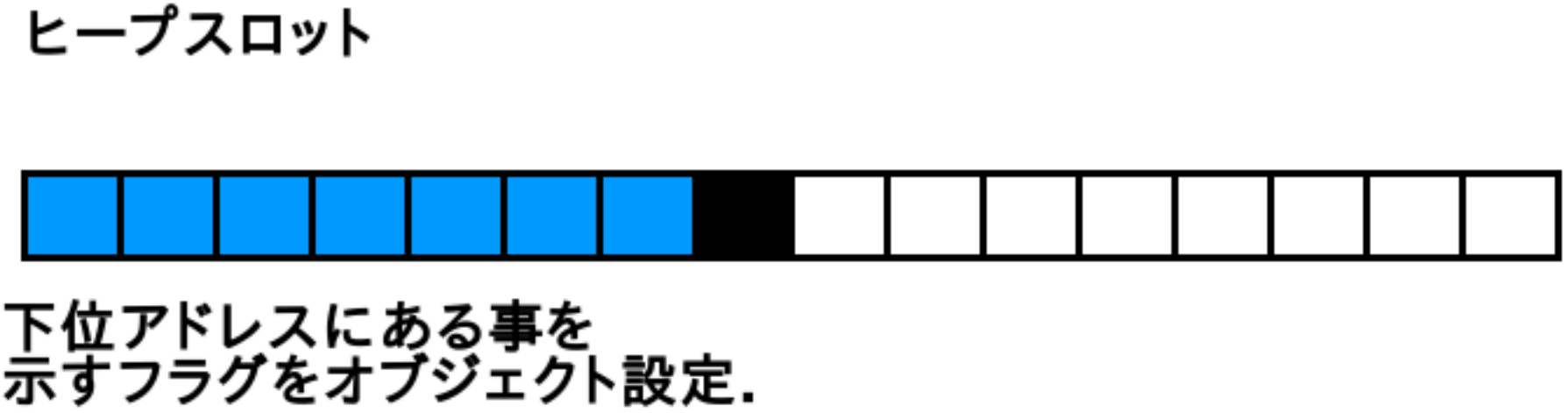
問題

問題



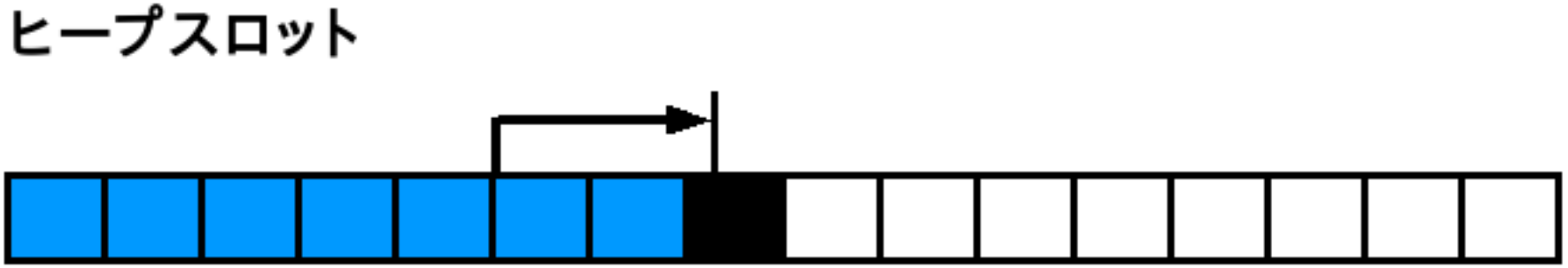
ビットマップ専用オブジェクト以下のアドレスにある
オブジェクトから探索できない

専用のフラグ設定



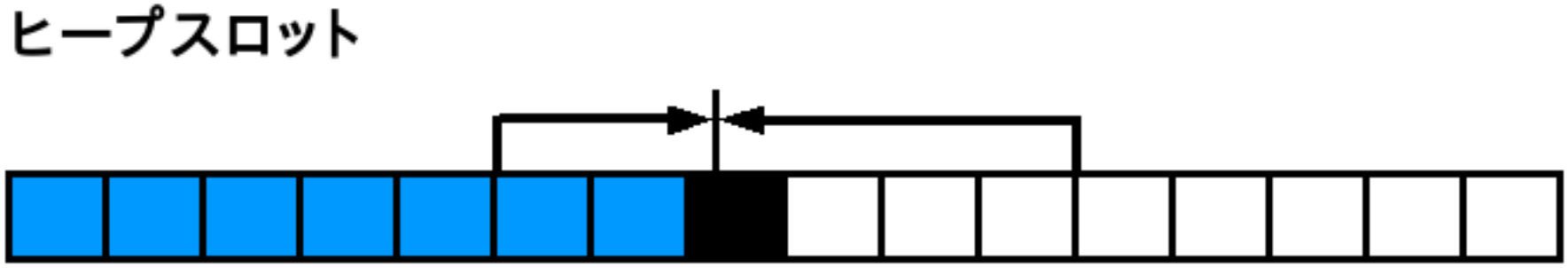
使用されなくなったマークビットを再利用

位置探索が可能に



自分が所属するヒープスロット内の
ビットマップ専用オブジェクトを指すように
分岐して処理を変えられる.

すべてオブジェクトから探索可能に



ヒープスロット内のすべてのオブジェクトからビットマップ位置を探索できる事が可能

実コード

```
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));
}
```

新探索手法の特徴

- ヒープスロットの数が関係ないO(1)のアルゴリズム
  - 高速に動作
- ポータブル性が高く様々なプラットフォームに適用可能

(7)
RubyGCへのビットマップマーキング実装

ビットマップマーキング導入によって変更になった点

- ヒープのサイズを調整(少々拡張)
- ビットマップをint型の配列で作成
- 今回提案のビットマップ位置探索を実装

ビットマップマーキング導入によって変更になった点

- スイープ時に使用されていないオブジェクトに関しては極力操作しない
  - フリーリスト用のリンクポインタ
- マークの消去処理をヒープスロット単位で一括で行う
  - 対応したビットマップ領域を 0 でクリア

(8)性能評価

マシンスペック

- CPU
  - IntelCoreDuo(1.83GHz, 2MB L2 キャッシュ, 667MHz FSB)
- メモリ
  - 2GB
- OS
  - Linux2.6.24

マシンスペック

- コンパイラ
  - gcc 4.2.3

比較対象

- orgin
  - ruby 1.9.0-4
- btree
  - ビットマップマーキング使用
  - ビットマップ位置探索に二分木探索を使用
- align
  - ビットマップ位置探索に提案手法を使用

ベンチマーク用プログラム

- skk.rb
  - SKKサーバの様な動きをする
  - 郵便番号を送ると住所を返してくれるデーモン
  - forkによって5個のサブプロセスを生成し，親プロセスでTCP通信を待つ

ベンチマーク用プログラム

- クライアント側(Rubyで実装)
  - クライント数 10
  - それぞれのリクエスト数 200

評価:GC最大処理時間

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

考察

- GC最大処理時間
  - orign < btree　91%程遅い
  - orign < align 7%程遅い
    - ビットマップ位置の計算処理により若干遅くなった
    - alignではbtreeより随分高速化された

評価:各プロセスの合計メモリ使用量

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

考察

- forkで生成した各5個のプロセスの私有領域の合計
  - ビットマップマーキングを使用する事で半分以上削減された
  - 省メモリで動作している事が分かる

評価:SKKへのサーバアクセス速度

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 (msec)
orign	0.90
btree	0.87
align	0.85

考察

- 平均リクエスト終了時間
  - ビットマップマーキングを使用した方が速い
  - 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
  - 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

まとめ

- RubyGCにビットマップマーキングを実装
  - forkにてサブプロセスを生成するプログラム
    - メモリ使用量が62%削減
  - メモリ消費量を抑える事が出来る
  - ApacheHTTPサーバの環境の元で動作するWebアプリケーションに効果的
    - REEでも実証済み

まとめ

- ビットマップ位置を探索する手法
  - ポータブルかつ高速に探索可能な手法を提案
  - 通常のGCよりもGC最大処理時間が7%遅い
  - 二分木探索使用時よりもGC最大処理時間が43%速い

ご静聴
ありがとうございました