メモリ効率のよい実時間GC

電気通信大学 情報工学科
鵜川 始陽

自己紹介

鵜川 始陽 （うがわ　ともはる）

- 1996年 京大マイコンクラブ（KMC）入会
  - http://www.kmc.gr.jp/
  - Linux/98: LinuxをNEC PC-9800シリーズに移植
  - 日本語変換エンジンAnthy
    - でもSKKが好き
- 1999年 京大湯淺研究室配属 (4年生)
  - 博士課程修了までは一級継続の実装の研究
  - その後，メモリ管理（GCなど）の研究
- 2008年 湯淺研究室卒業
- 現在　　電通大情報学科

概要

- 組込み用のJava処理系に実時間GCを実装
  - KVM: J2ME CLDCのVM
  - スナップショットGC
  - リターンバリア
  - 複製に基づくコンパクション

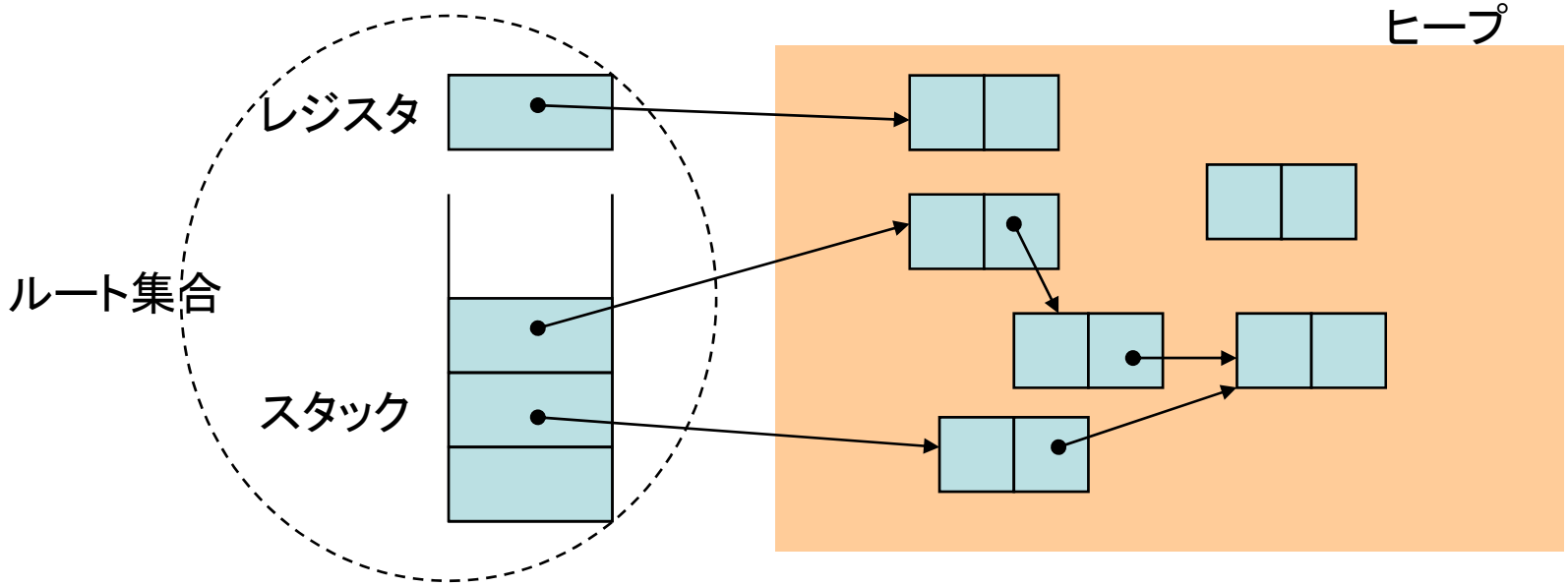
目的

組込みソフトウェアの開発にGCを使いたい

- GC（ごみ集め，Garbage Collection）
  - 自動メモリ管理
  - 最近は空気みたいなもの
    - Java, LISP, Haskell, ML, …
    - Perl, PHP, Python, Ruby, JavaScript, …
    - ないのはC, C++, Fortran, アセンブラぐらい
- 組込みソフトウェア
  - 特定のハードウェアのためのソフトウェア
    - 携帯電話
    - 自動車やロボットの制御
    - 工場のラインで製品の検査装置

ごみ集め(GC)とは

- プログラムが使わなくなったデータ(ごみ)を検出
- 領域を再利用
- ポインタを基準にごみを判定
  - プログラムはポインタで指されなくなったデータを扱えない



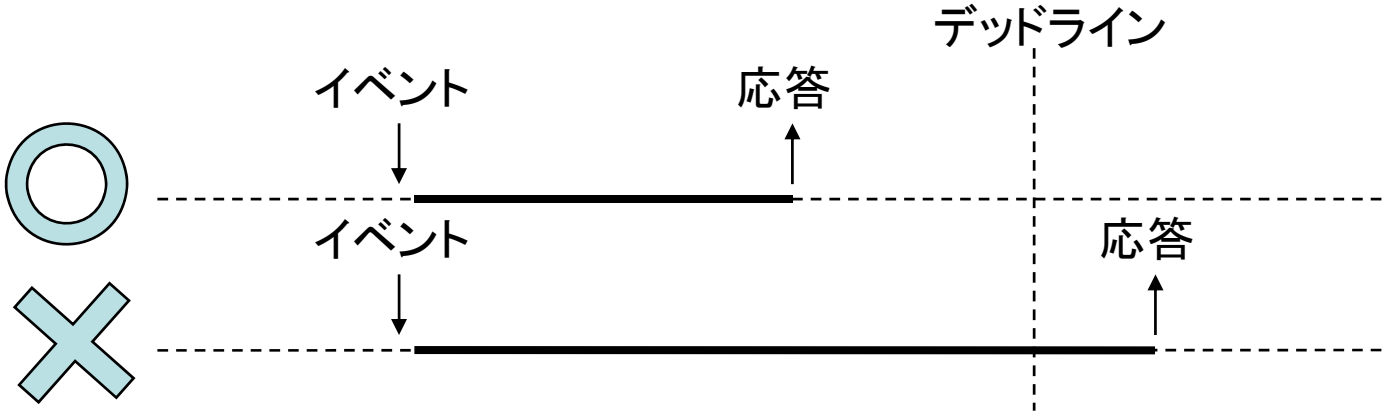
組込みソフトウェア

- 性能があまりよくないハードウェア
  - 遅いプロセッサ
  - 余裕のないメモリ
- バグが出たら致命的
  - ソフトウェアのバグが原因の事故
  - 動作が予測可能
- 実時間アプリケーションが多い

実時間アプリケーション

イベントから一定時間以内に反応する

- 人型ロボットが倒れそうになったら倒れる前に間接を制御してバランスをとる
- ビデオの再生で，バッファが空になる前に次のフレームをデコードする
- ゲームで，1/60秒以内に1フレーム分の処理をする



組込みソフトウェア開発

- 従来はC++で開発される場合がほとんど
  - プログラマがアプリケーションの動きを把握できる
  - アセンブラよりは書きやすい
  - 熟練したプログラマが開発
- そろそろ破綻してきている
  - 短い開発期間
  - プログラムの大規模化
  - 人的リソース不足

⇒プログラムの品質低下

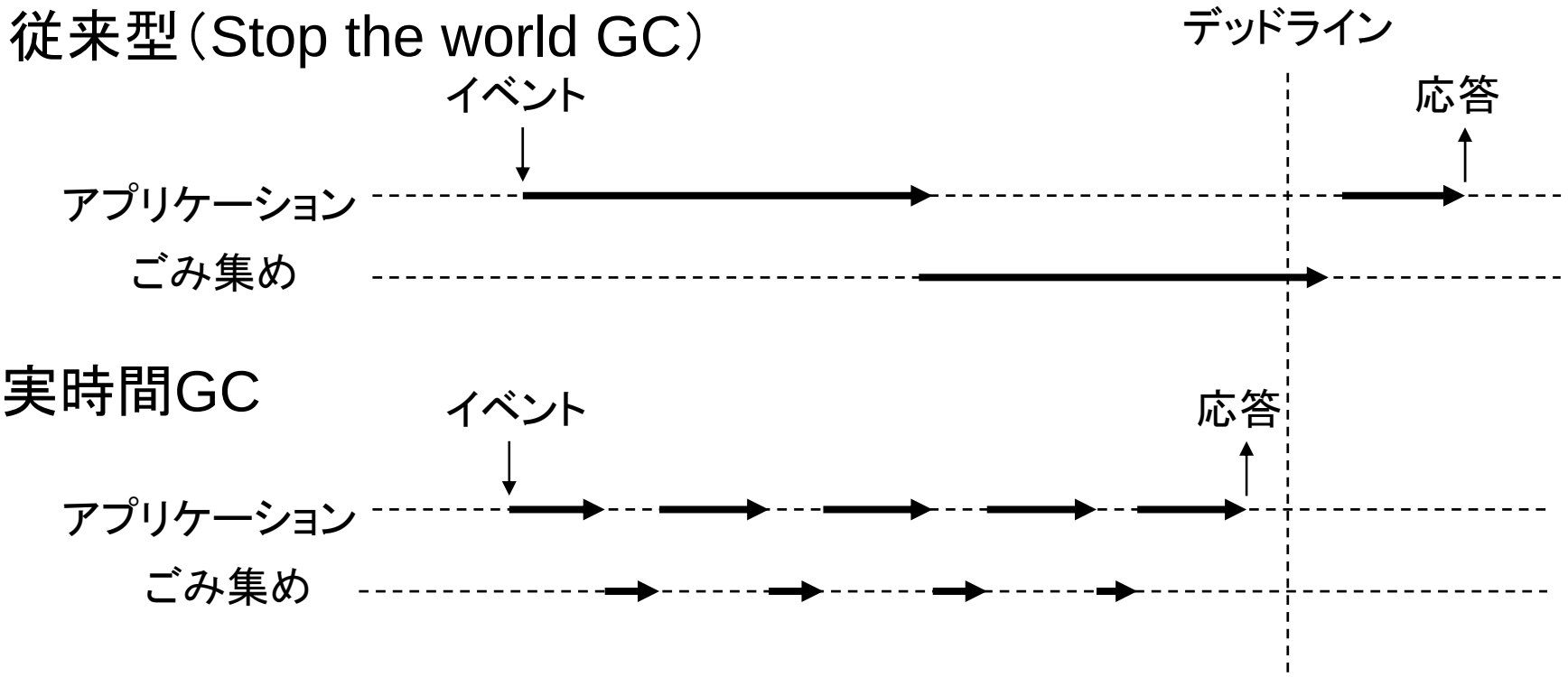
組込みソフトウェアにGCを

GCのある言語で組み込みプログラム開発

- 利点
  - メモリ管理の労力削減
  - メモリ関連のバグがなくなる⇒品質向上
    - メモリリーク
    - 誤った解放
- 欠点
  - 洗練されたC++プログラムより実行速度は劣る
  - GCが始まると，ユーザプログラムが停止する
    - 実時間アプリケーションでは問題

実時間GC

アプリケーションプログラムの実行の間に
少しずつGCを進める



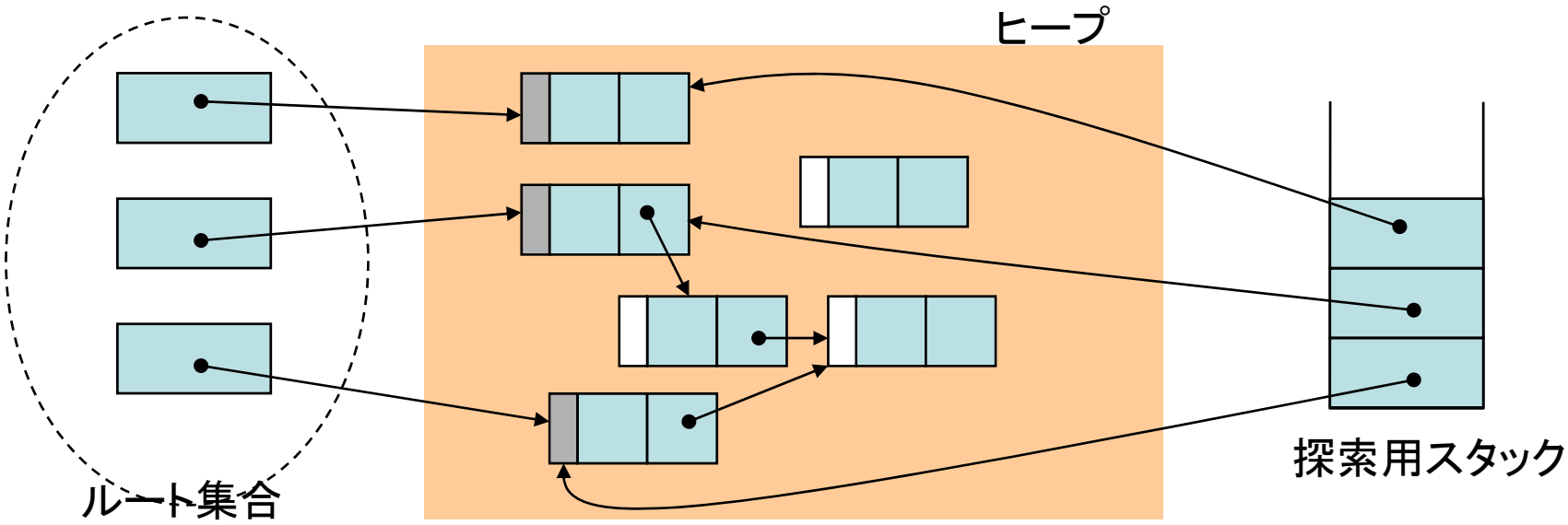
実時間GCのアルゴリズム

- スナップショットGC
  - 実時間化したマークスイープGC
- リターンバリアによるルートスキャン
- 複製に基づく実時間コンパクション

マークスイープ方式（1）

マーク・フェイズ

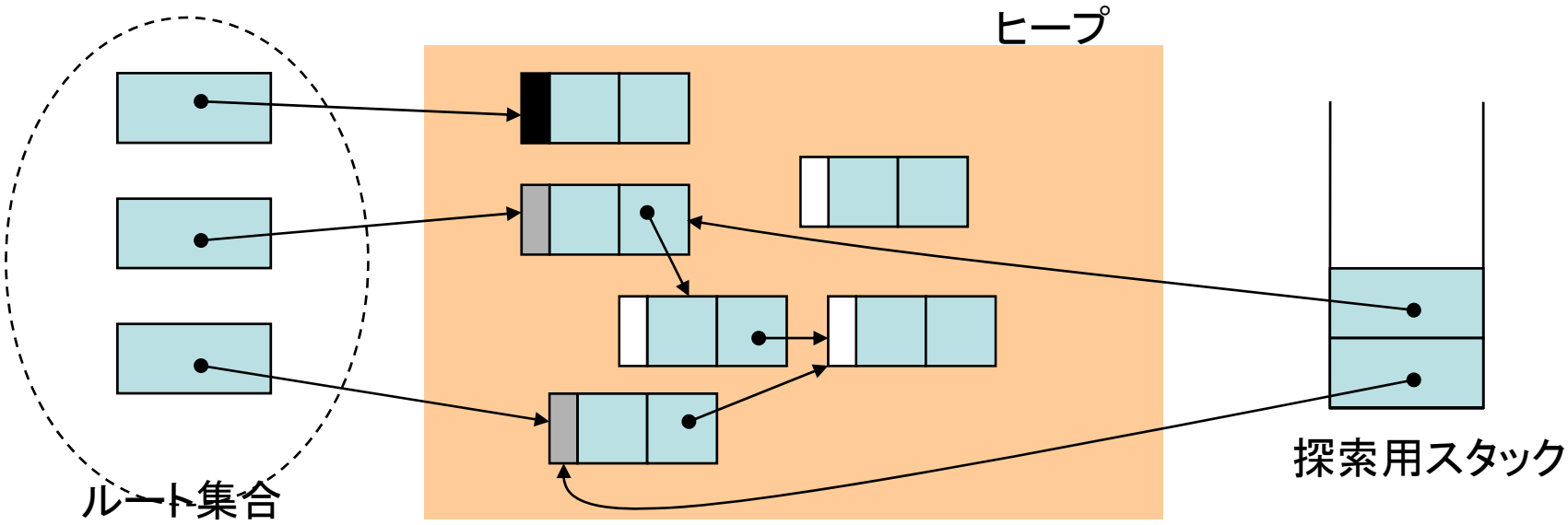
- ルートからポインタをたどる
- 到達できたオブジェクトにマーク付け
- グラフの探索



マークスイープ方式（1）

マーク・フェイズ

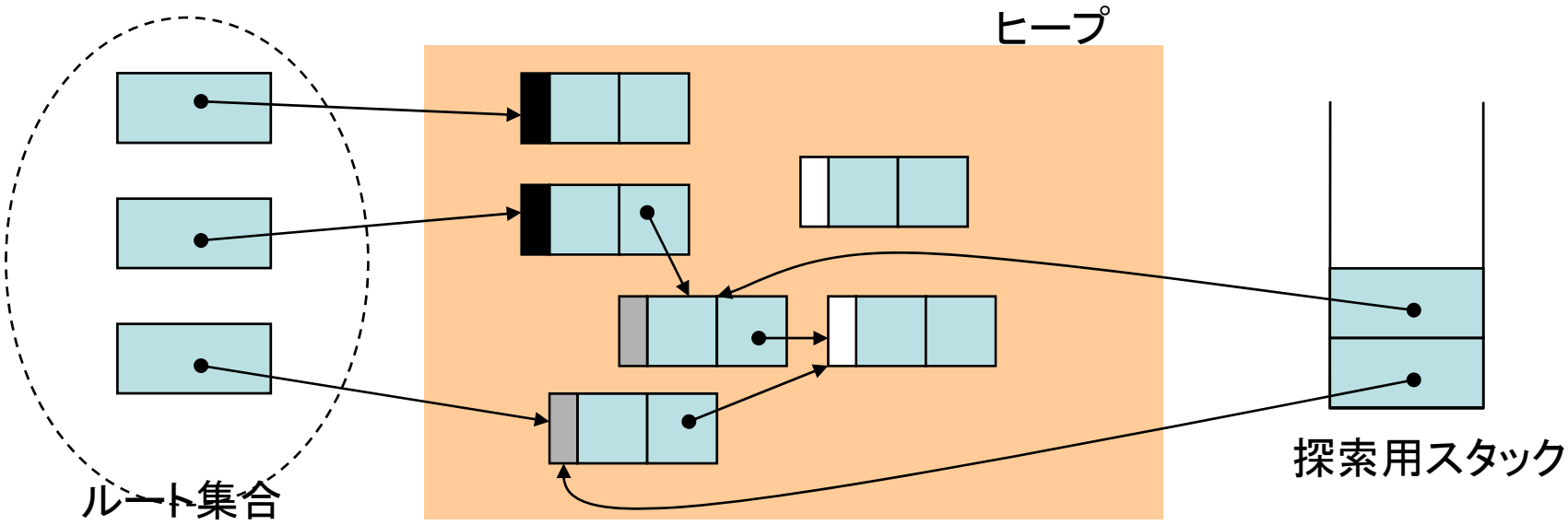
- ルートからポインタをたどる
- 到達できたオブジェクトにマーク付け
- グラフの探索



マークスイープ方式(1)

マーク・フェイズ
- ルートからポインタをた
- 到達できたオブジェク
- グラフの探索

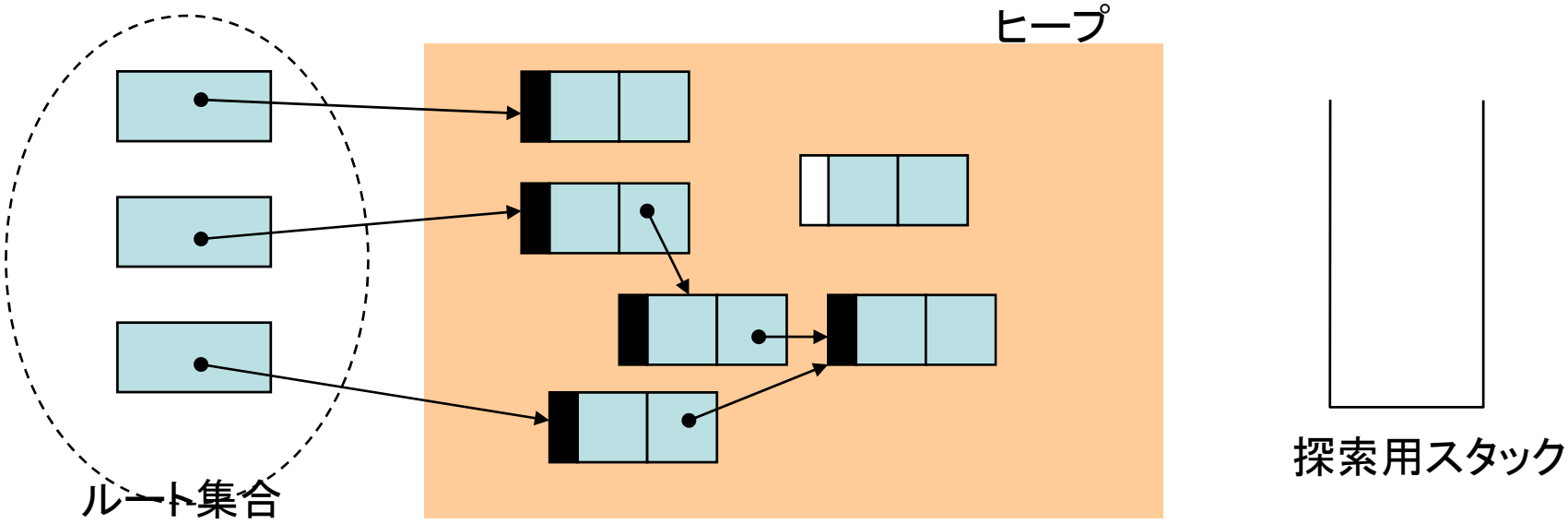
- 白:まだ到達していない
- 灰:到達したが,
  その先はまだ調べていない
- 黒:調べ終わった
  (二度と調べられない)



マークスイープ方式(1)

マーク・フェイズ
- ルートからポインタをた
- 到達できたオブジェク
- グラフの探索

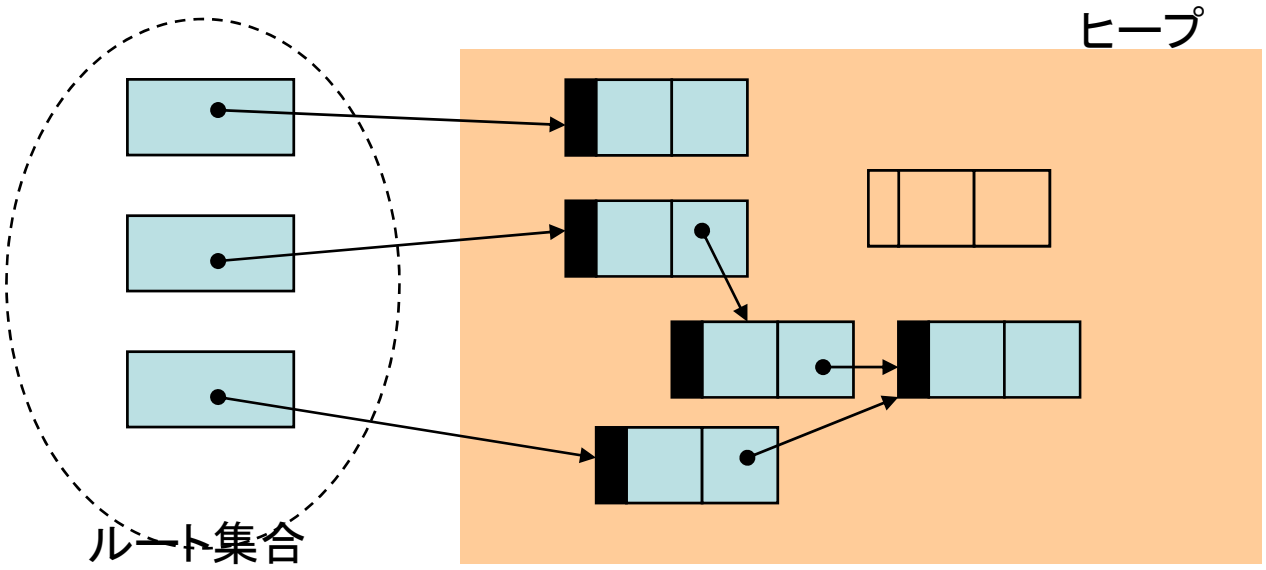
- 白:まだ到達していない
- 灰:到達したが,
  その先はまだ調べていない
- 黒:調べ終わった
  (二度と調べられない)



マークスイープ方式(2)

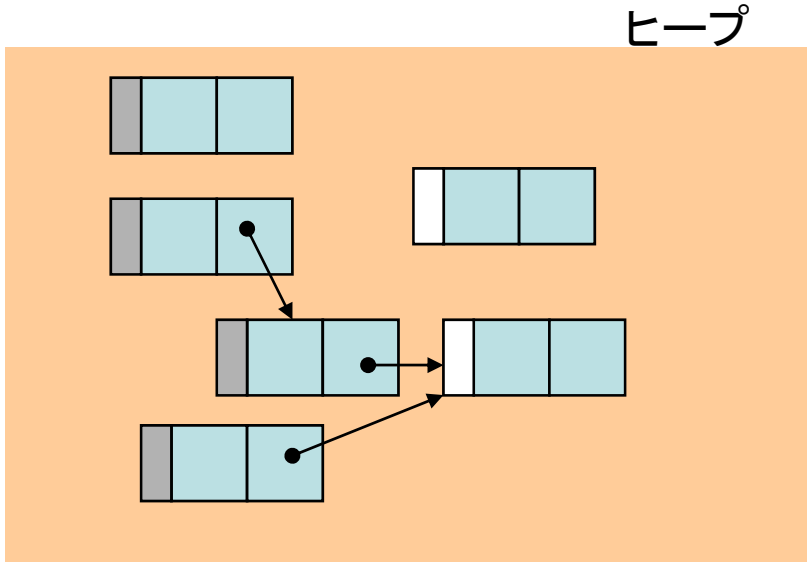
スイープ・フェイズ

- ヒープを端からスキャン
- マークがついていないオブジェクトを回収
  ⇒空き領域のリスト(フリーリスト)に登録



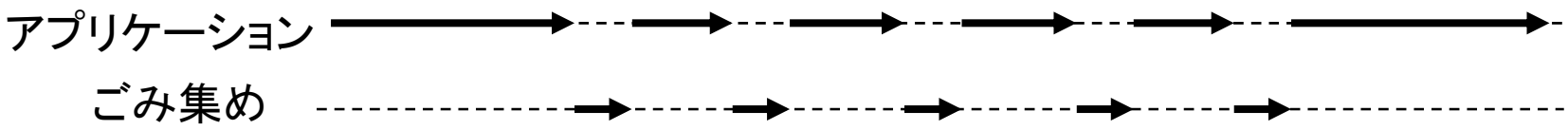
制限されたメモリでマークフェイズ

- 探索用スタック溢れの危険
- 溢れたらヒープをスキャンしてマークのついたオブジェクトを探す
  - 全てのマークがついたオブジェクトから探索しなおす



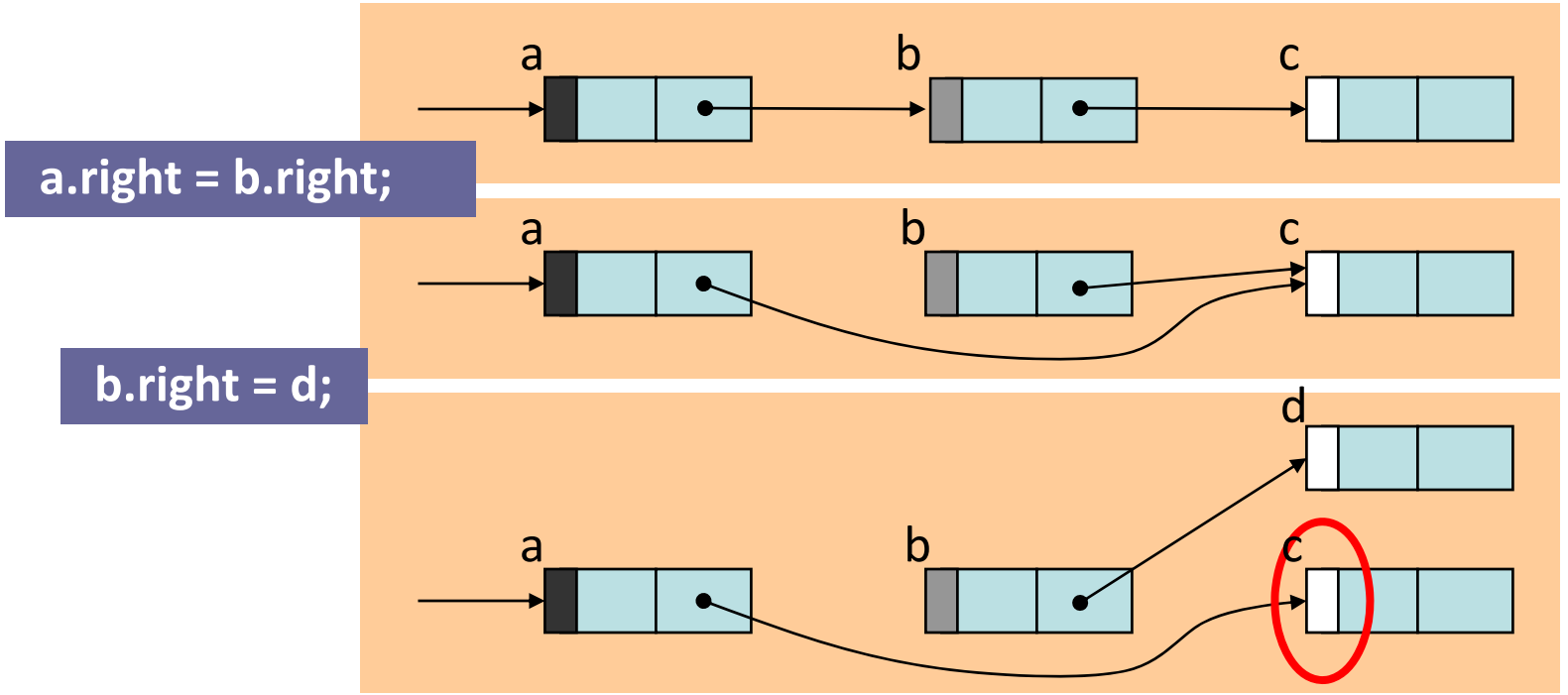
マークスイープの実時間化

- マーク・フェイズとスイープ・フェイズをそれぞれ分割して，アプリケーションの実行の間に少しずつ行う



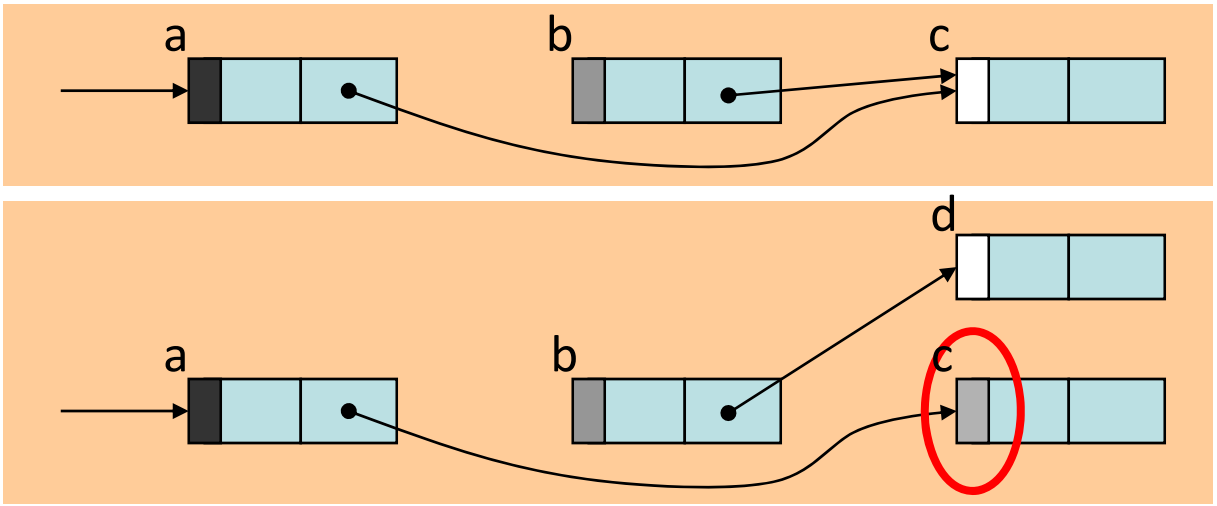
マーク処理の分割

- GC の進行中でも，アプリケーションの実行が進む
- オブジェクト間の参照関係が変化する可能性
  ⇒生きているオブジェクトのマーク漏れ



スナップショットGC [湯浅 ’90]

- 書込みバリア
  - ポインタを書き換えるとき，
    それまで指されていたオブジェクトが白なら灰色にする
- GC開始時にルートは一括してスキャン
- ルートにバリア不要
- GC開始のタイミングが見積もりやすい



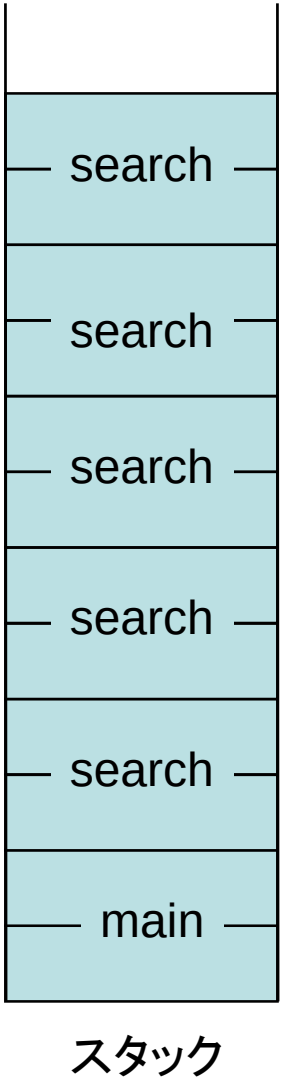
スナップショットGCの問題

- ルートスキャンは分割しない
  - 深い再帰呼出し　　⇒リターンバリア
  - 多数のスレッド
- オブジェクトを移動しない
  - メモリフラグメンテーション
    空き領域の合計は十分あっても，連続領域が確保できず，大きなオブジェクトが作れない

⇒実時間コンパクション

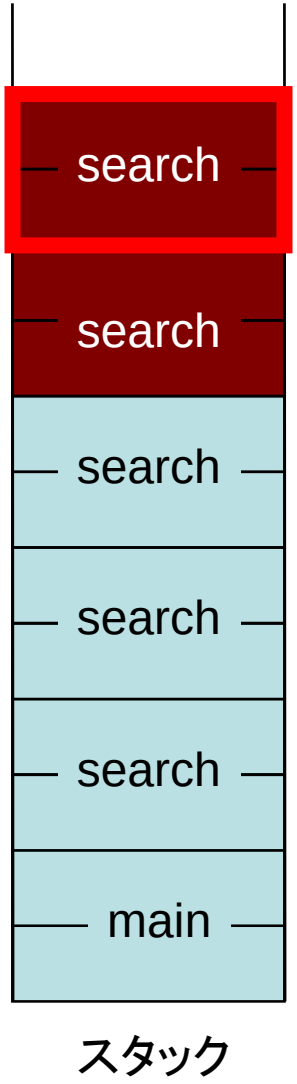
ルートスキャンの問題

- ルート集合
  - レジスタ　　⇒固定
  - 大域変数　　⇒固定
  - スタック　　⇒動的に伸縮
- 全てのスレッドのスタックをスキャンしなければならない
  - イベント処理用スレッドのスタックは小さくても・・・
- スタックのスキャンによる停止時間が問題となる場合も



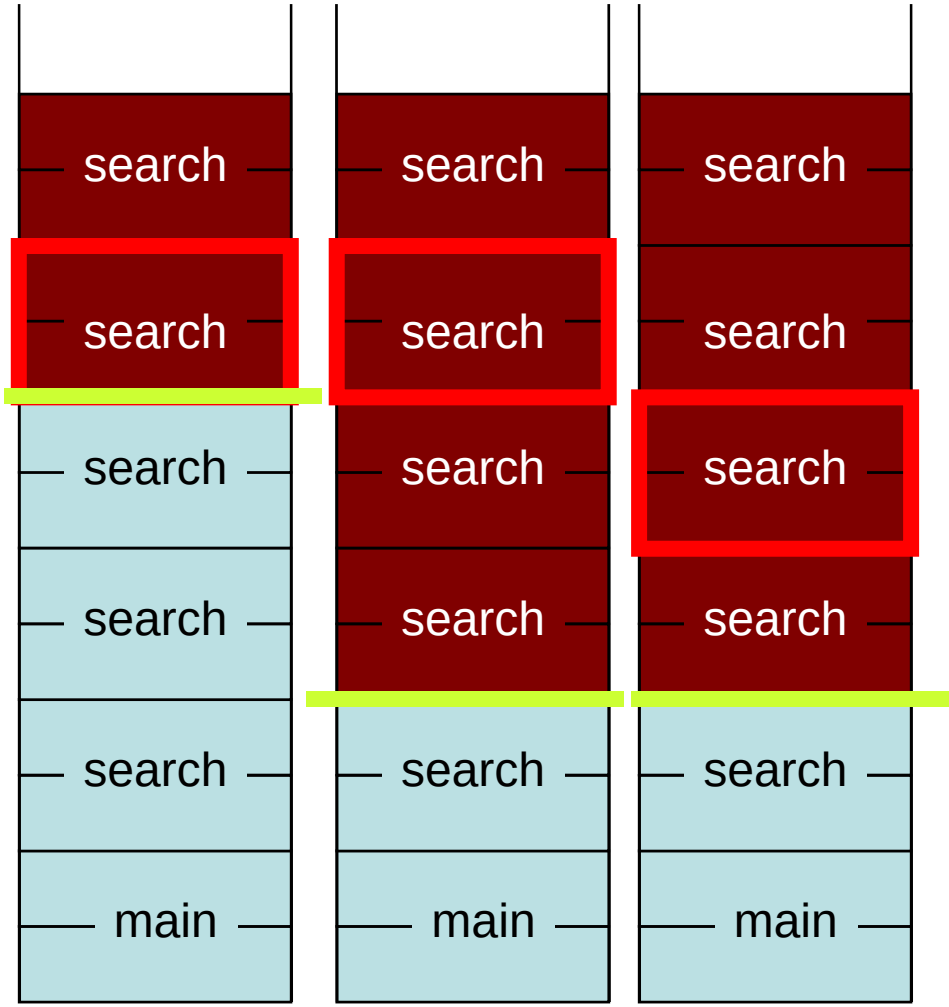
リターンバリア[湯淺ら ’01]

- スタックを関数フレーム単位で分割してスキャン
  - アプリケーションはカレントフレームしかアクセスしない
  - GC開始時：少なくとも全てのスレッドのカレントフレームをスキャン
  - それ以外のフレームはアプリケーションの実行の間に少しずつスキャン



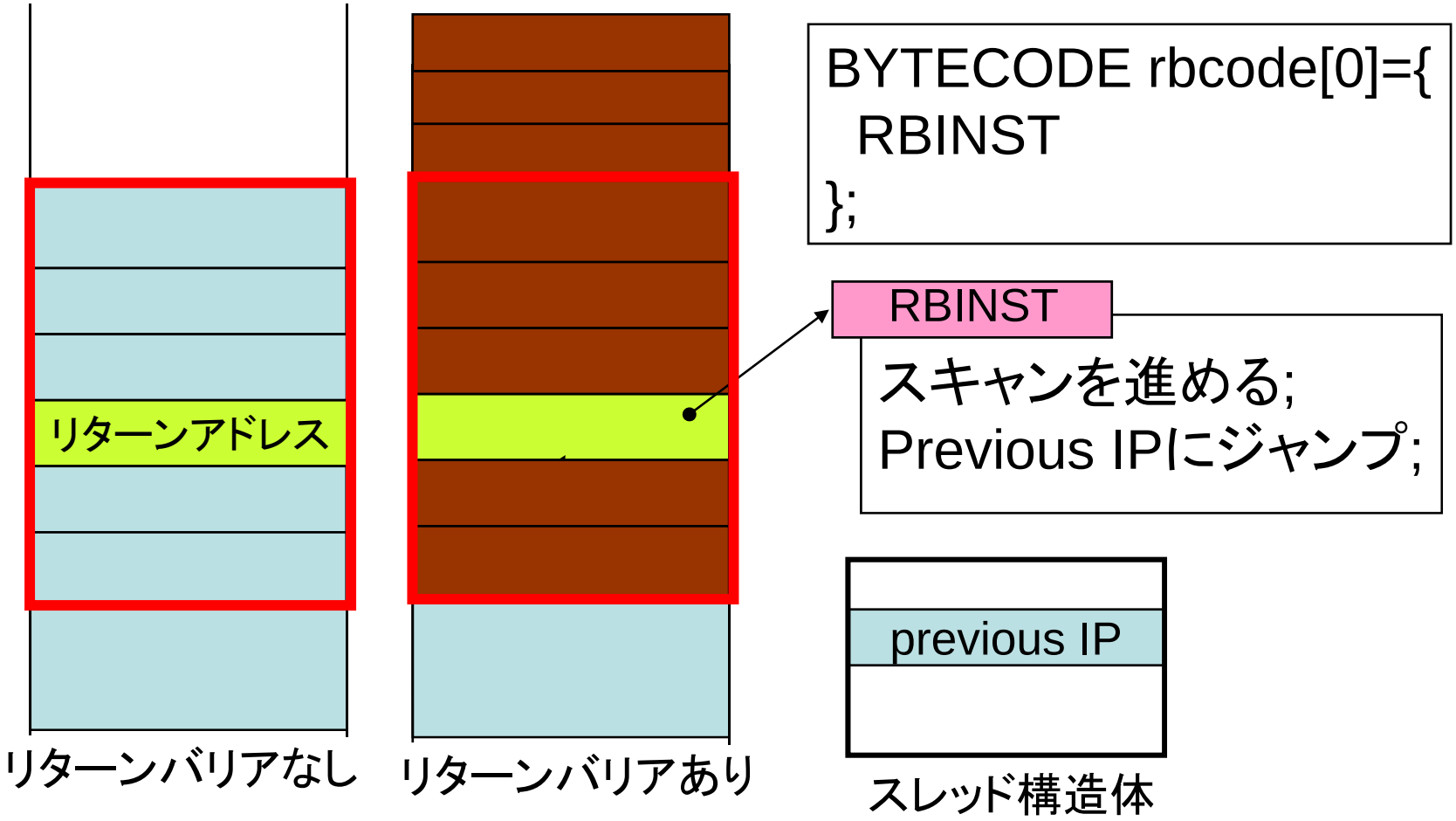
リターンが早いと

- リターンがスキャンを追い越しそうになったら
  1. リターンバリア起動
  2. スキャンを少し進める
  3. リターンバリアを再設定
  4. リターンする

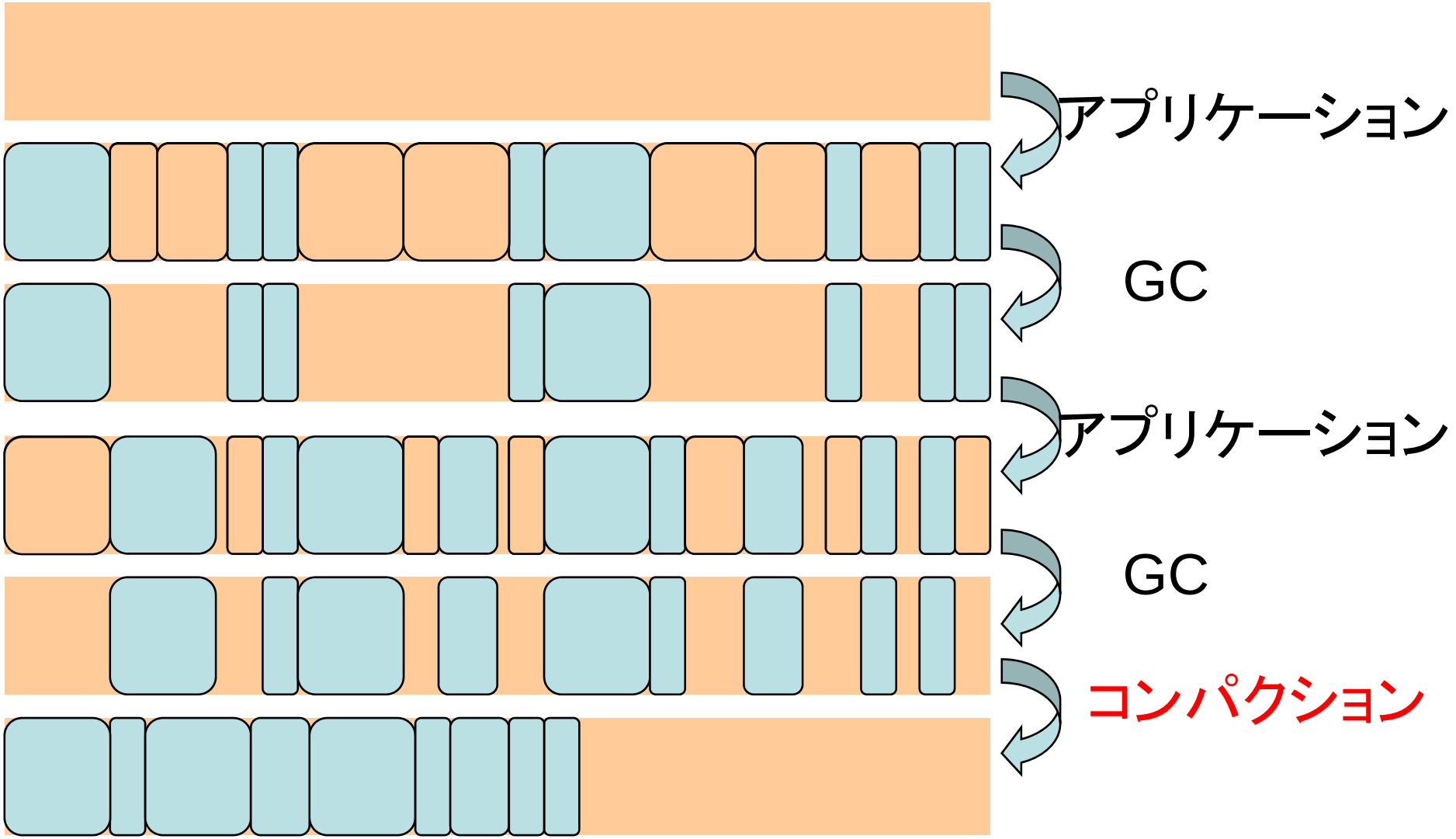


リターンバリアの実装

- リターンアドレスをフック

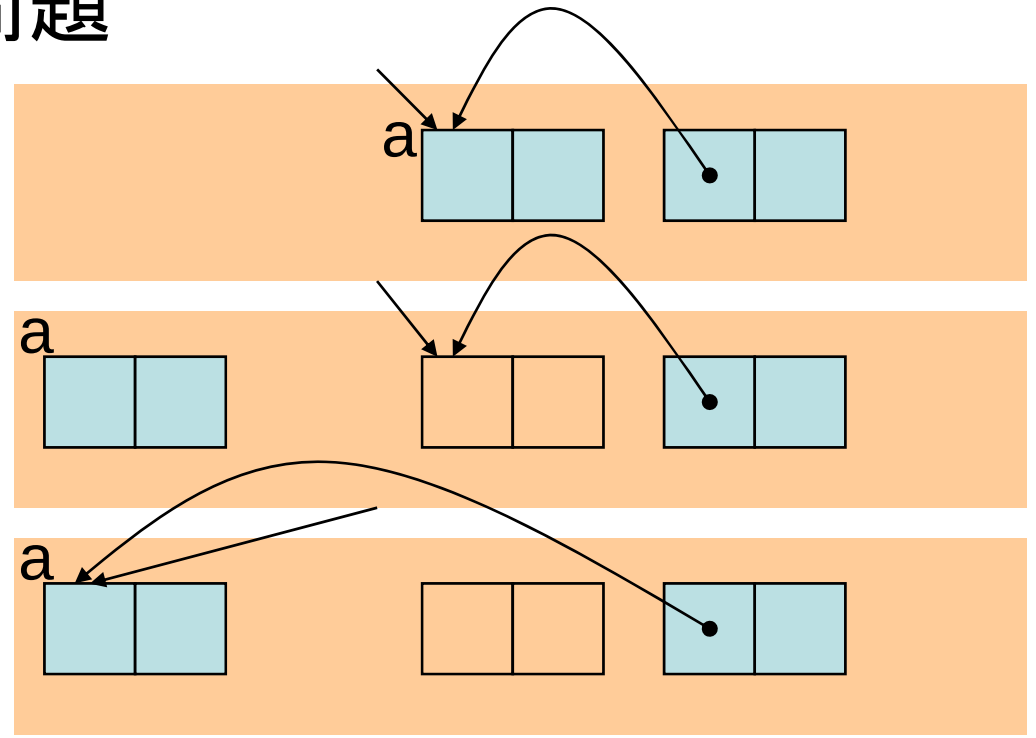


フラグメンテーション



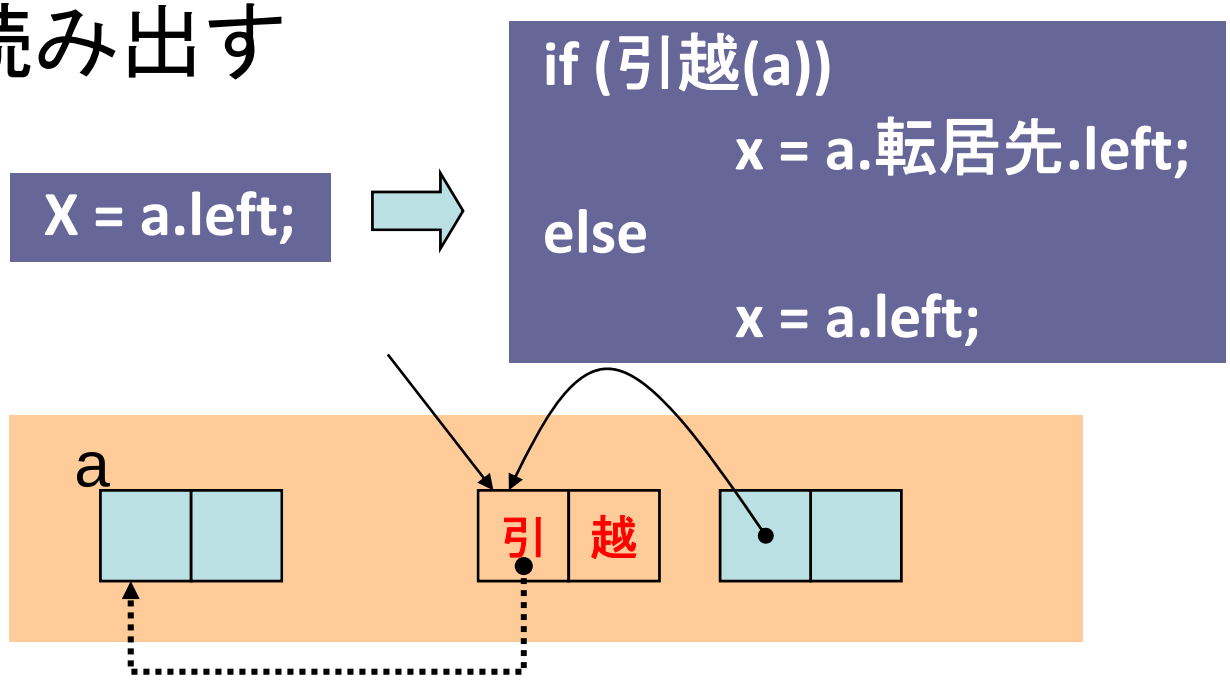
コンパクションの分割

- コンパクションはオブジェクトを移動させる
- 移動したオブジェクトを指すポインタも更新しないと問題



リードバリア

- オブジェクトのスロット読み出し時に
  オブジェクトが移動していたら，
  移動先から読み出す



リードバリアの欠点

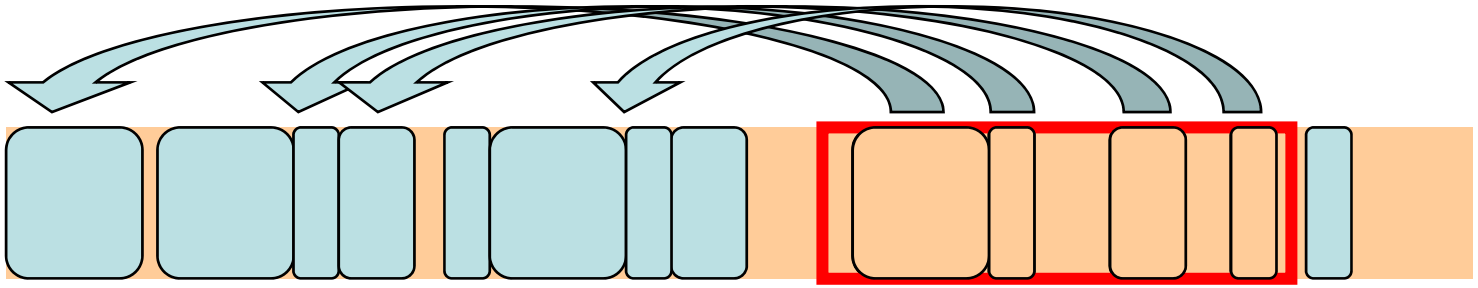
- オーバヘッド大
  - 読み出し操作の回数 >> 書込み操作の回数
  - 手続き型言語でも当てはまる
- システムの信頼性低下の場合も
  - バリアの挿入が自動でできない場合
    - CやC++で書かれたライブラリ
  - バリアの挿入箇所が膨大
    ⇒バリア挿入忘れ

複製に基づく実時間コンパクション[話者ら ’08]

- Replication GC[Nettlesら ’93]を応用
  - コンパクションの進行中は，移動元と移動先の両方が最新の状態を保持する
  - アプリケーションはどちらにアクセスしてもよい
- リードバリアなし
  - 書き込みバリア
  - ポインタ比較演算にバリア

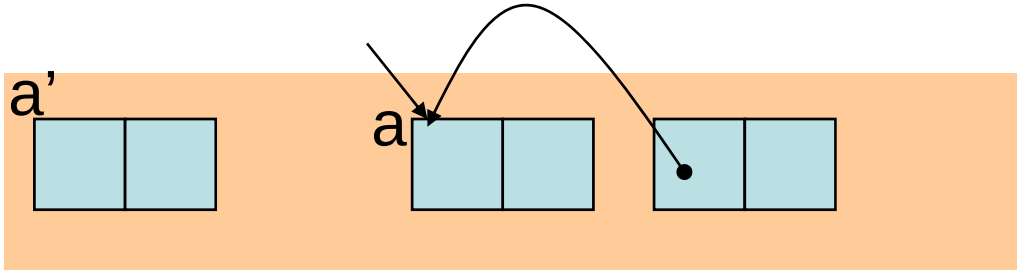
不完全なコンパクション

- 連続した空き領域を作ることが目的
- フラグメンテーションの激しい箇所のオブジェクトを移動させる

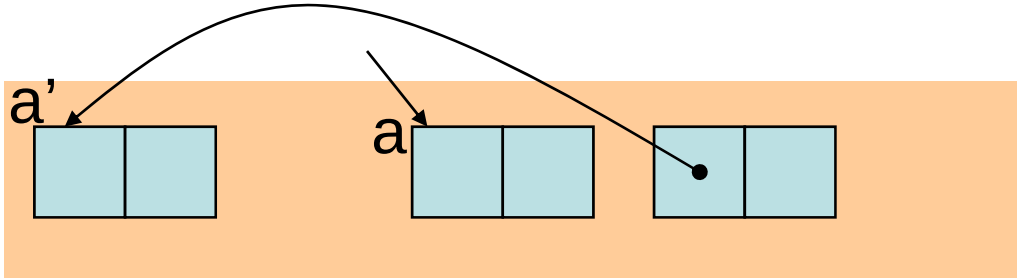


コンパクションの流れ

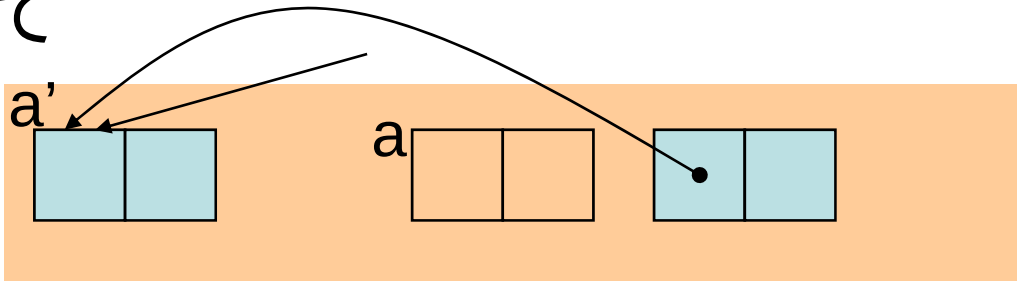
1. オブジェクトを複製



2. ヒープをスキャンして
   ポインタ更新

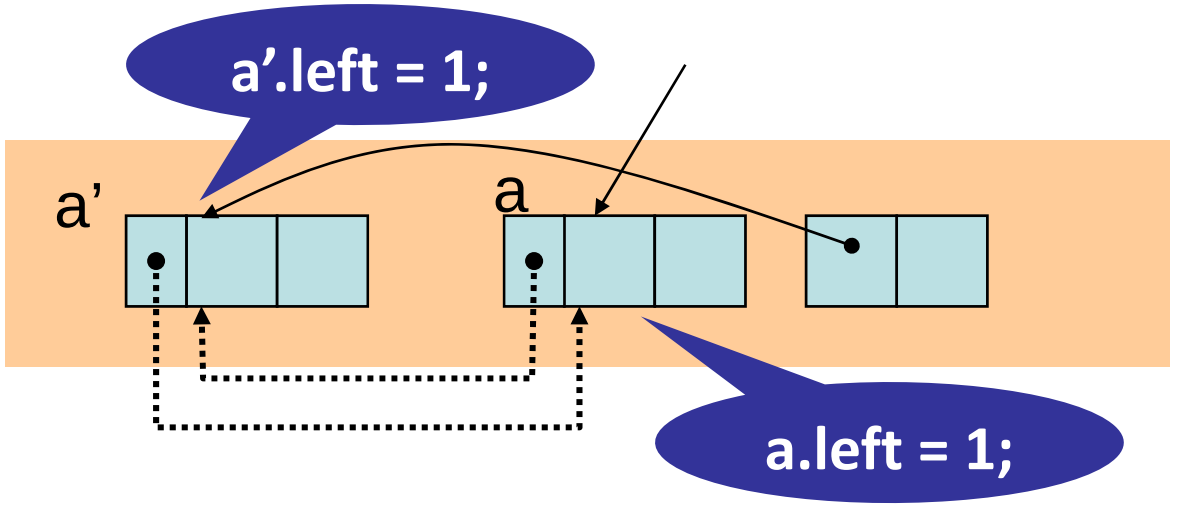


3. ルートをスキャンして
   ポインタ更新



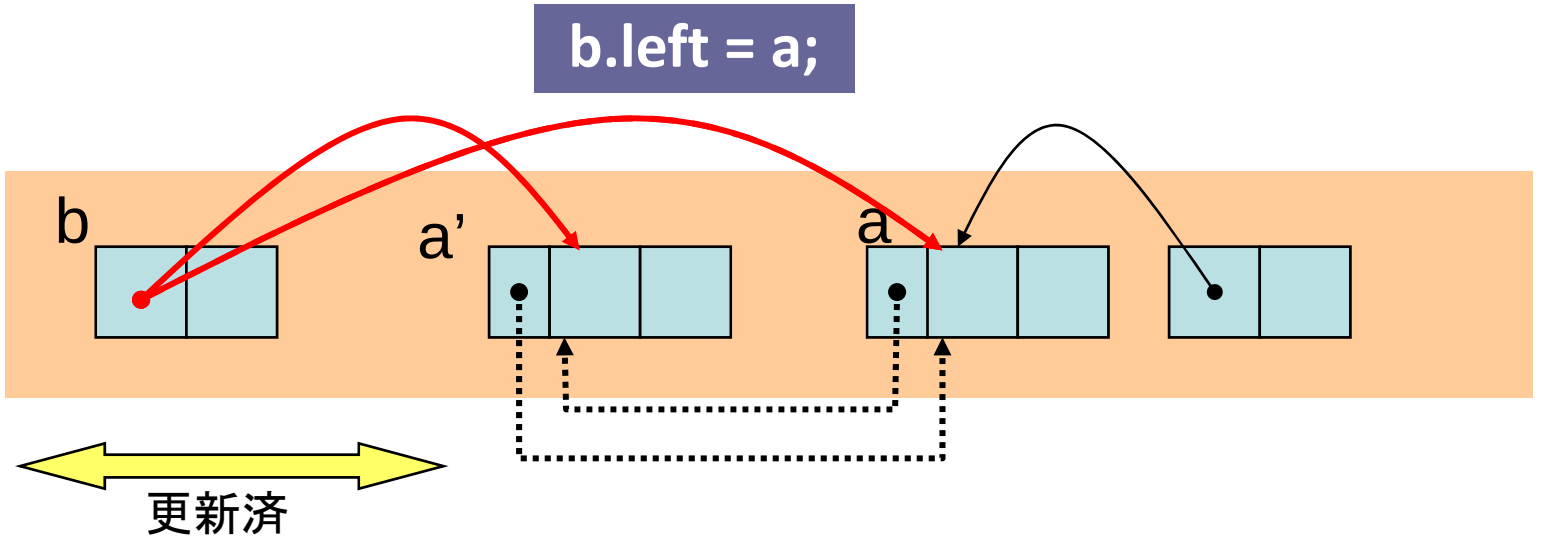
書き込みバリア（1）

- 書き込みの複製
  - 片方に書き込んだら他方にも書き込みを反映



書き込みバリア(2)

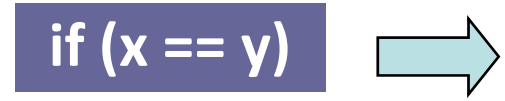
- 書き込み時のポインタ更新
  - ポインタ更新済みの領域に書こうとしたポインタを更新



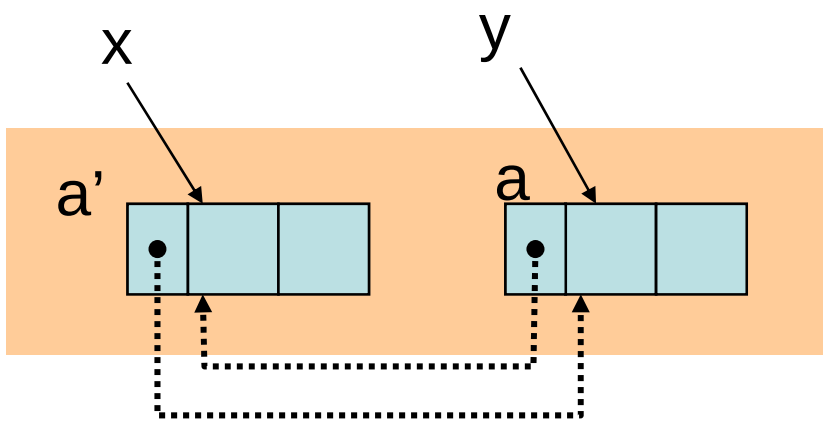
ポインタ比較演算のバリア

- 同じオブジェクトの実体が二つある
- 「==」演算に注意
- 定数との比較はバリア不要
  - NULLチェック

```
if (x == y)
```



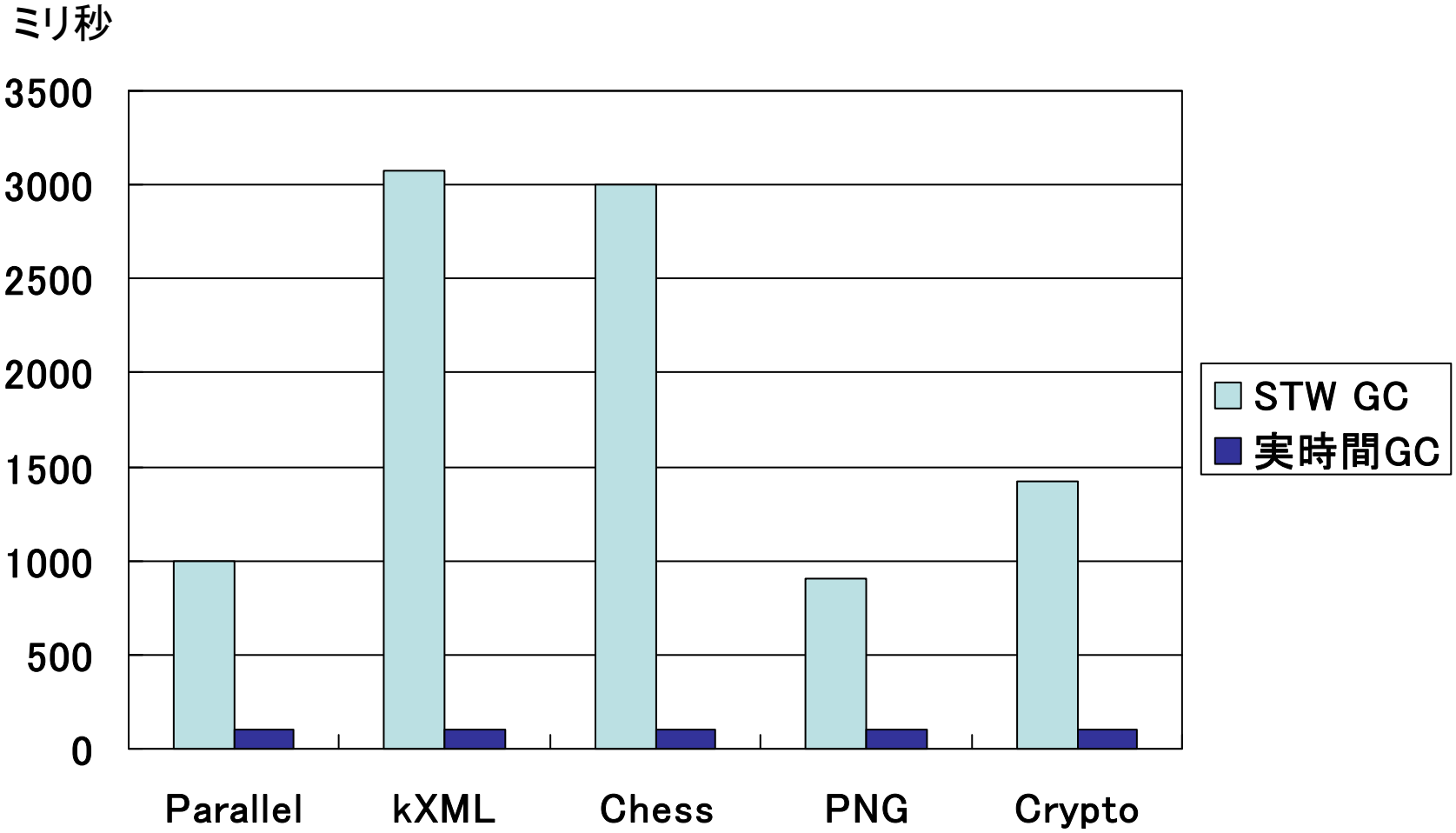
```
if (x == y ||
    (引越(x) &&
    x.転居先 == y))
```



性能測定

- 実験環境
  - Java VM: KVM CLDC 1.1
    - Javaのサブセット
    - Sun のリファレンス・インプリメンテーション
      - Community license
    - 携帯電話のJava（iアプリ）
  - CPU: Core2Duo 6400 (3.13GHz, Cache 2MB)
    - ARMプロセッサでの計測を進行中…
  - OS: Linux 2.6
- ベンチマークプログラム
  - GrinderBench: 携帯電話用Javaベンチマーク

分割したGC1回の時間

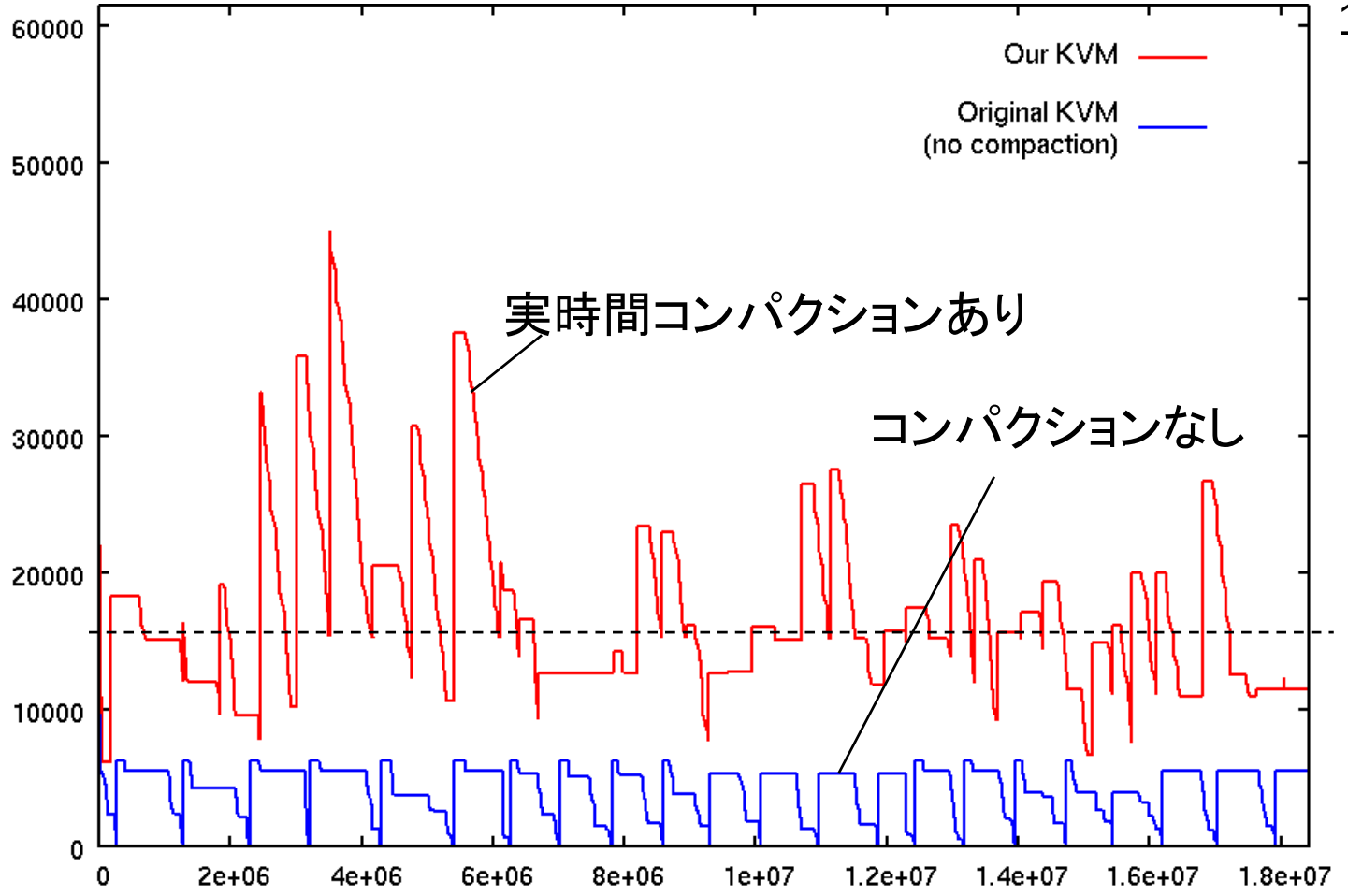


ヒープサイズ：5MB

フラグメンテーションの評価

連続空き領域（単位：バイト）

ヒープ全体に対して



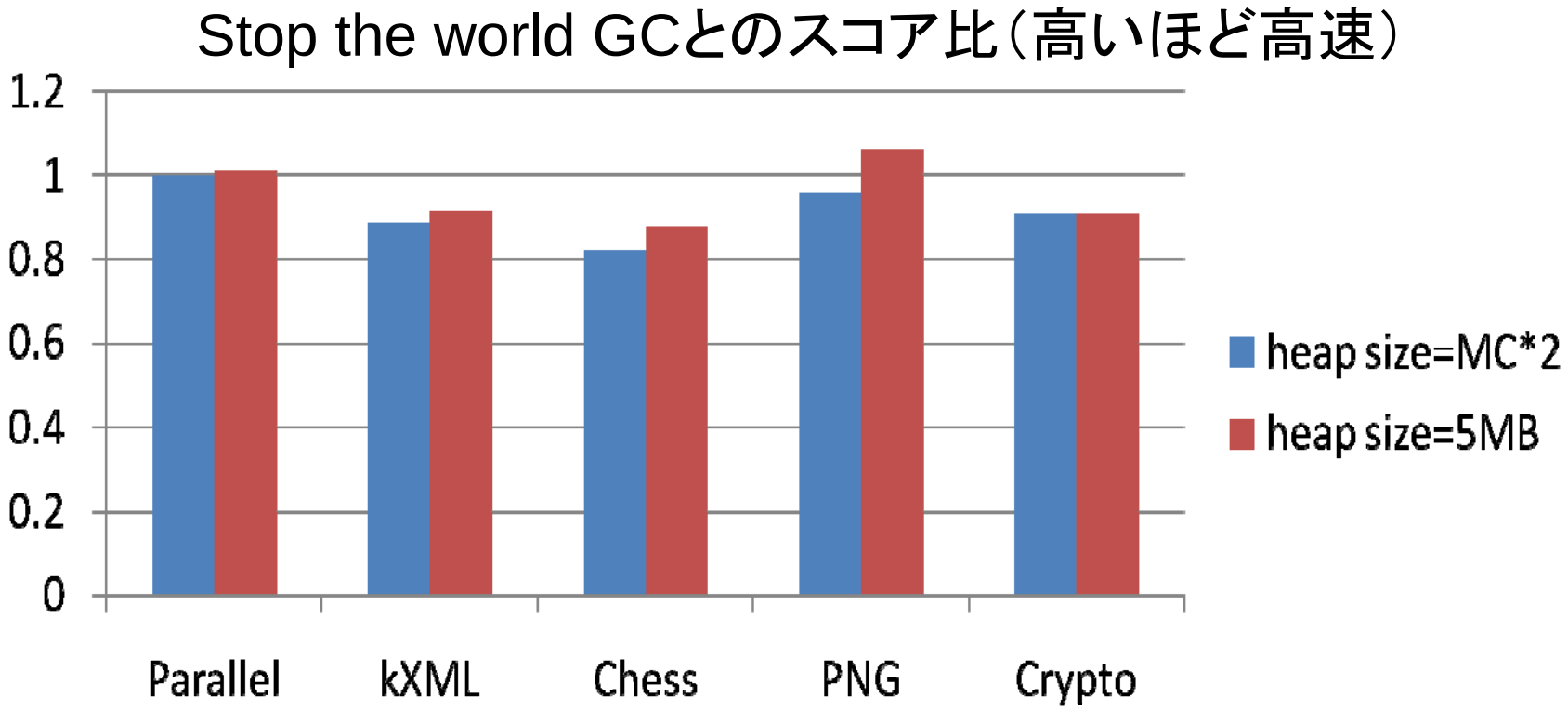
100%

約25%
（GC開始）

時間

Chessベンチマーク

ベンチマークスコア


Stop the world GCとのスコア比（高いほど高速）
1.2
1
0.8
0.6
0.4
0.2
0
Parallel
kXML
Chess
PNG
Crypto
heap size=MC*2
heap size=5MB

現状

- 現実味のある評価やデモの準備中
  - ARMプロセッサの乗ったボードでの性能評価
  - 本物の携帯電話（PDAのようなもの）に実装して市販ゲームでデモ

まとめ

- 組込み用のJava処理系に実時間GCを実装
  - スナップショットGC
  - リターンバリア
  - 複製に基づくコンパクション
- PC上ではそれなりの実験結果