# Rubyのメモリ管理の改善

*Improving Memory Management for Ruby*

東京大学大学院情報理工学系研究科創造情報学専攻
笹田 耕一
sasada@ci.i.u-tokyo.ac.jp, ko1@rvm.jp

東京大学
情報理工学系研究科
創造情報学専攻

# agenda

- 謝辞
- 貢献のまとめ
- 背景
- 現状のRuby処理系
- 改善手法の検討
- 実装
- 評価
- まとめ

# 謝辞


- 本研究では，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.
- 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.
- 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.

# 本発表の貢献のまとめ

- **Ruby実装の『制約』を元にメモリ管理改善を検討**
  - ヒープサイズを大きくしてスループット向上
  - OSと簡単に協調するヒープサイズ調整，スイープの改善
  - メモリ管理機構の実装を一新
  - 予備評価により，ある程度の成果を確認
- **来年リリースする Ruby 1.9.2 のメモリ管理へ**
  - メモリ周りでいい評判がないのでなんとかしたい
  - あまりドラスティックな変更はできない
- **現在進行中の仕事**
  - まだ，全ての実装が終わっていないため，最後までまとまっていません．ご意見募集．

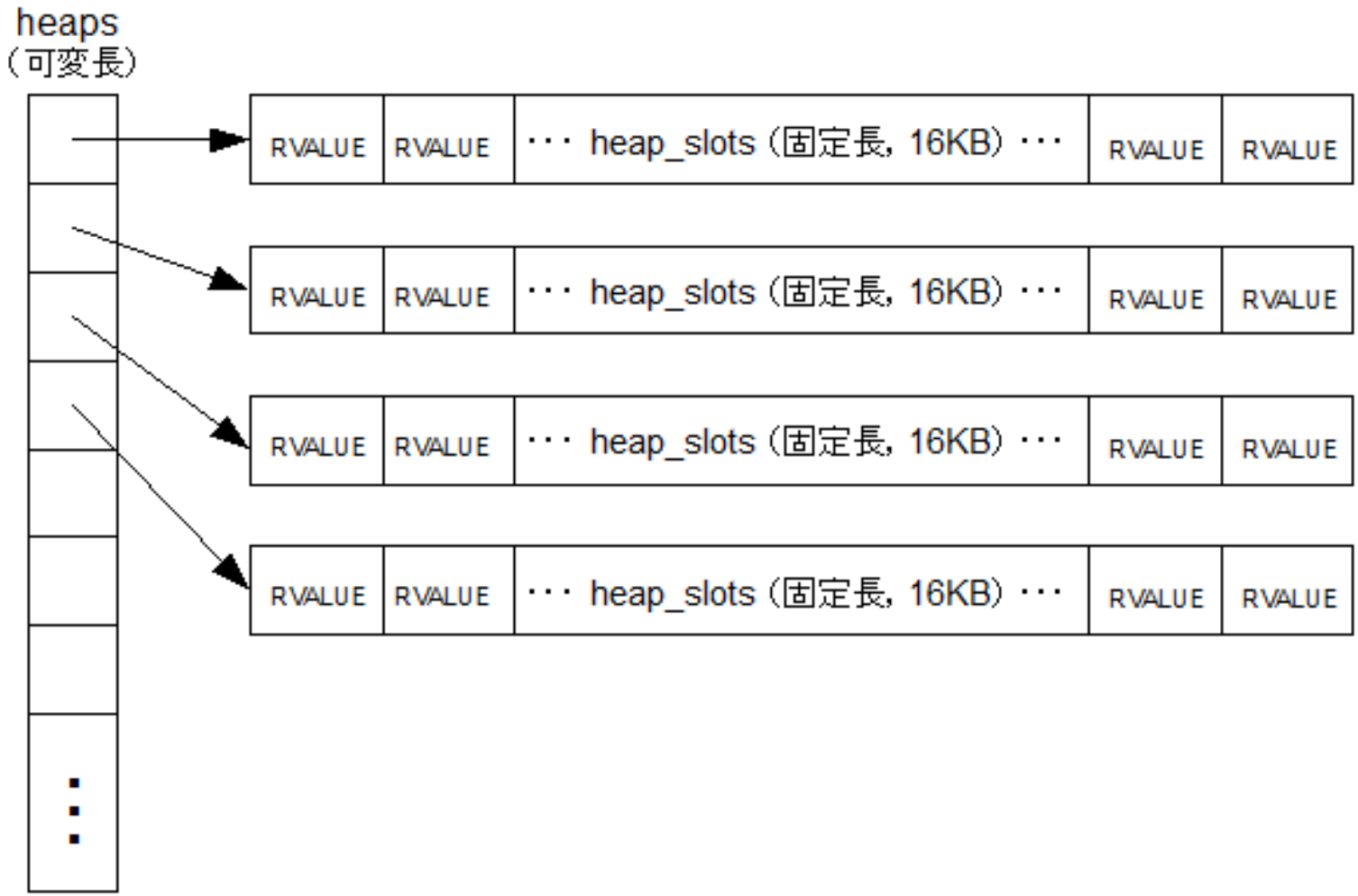
# 背景

- オブジェクトスクリプト言語Ruby
  - Ruby 1.9.1: 2009-01, Ruby 1.9.2: 2010-??
  - Ruby on Rails などで世界中で大ヒット
  - ! テキスト処理簡単言語
- Ruby処理系
  - インタプリタ
  - Cによる実装
  - VM，実行時にバイトコードコンパイル [笹田2005]
  - **GC：保守的なマークアンドスイープ**
    - オブジェクト操作は，ただのポインタ操作
    - 性能より，処理系および拡張ライブラリの作りやすさを優先

# 背景
# Rubyのメモリ管理

- RVALUE が Ruby オブジェクトを表現
  - 固定長 (5word)
  - 収まらない部分は malloc(), free() で管理
    - この点は本研究の対象外
      - Boehm GC で全部管理すればどうなる？
      - REE (Ruby Enterprise Edition) は tcmalloc を利用
- RVALUE は固定長配列(16KB) heap_slotsで管理
  - 以前は可変長→オブジェクトフラグメンテーション対策
  - 32bit マシンでは約200，64bitマシンでは約100 Object
- heap_slots は heaps という配列で管理
  - アドレスで整列

# Rubyのメモリ管理
# heaps と heap_slots



# Rubyのメモリ管理
# GCの実装

- マークフェーズ
  - 保守的にマシンスタックなどのルートから辿りマーク付
  - ポインタをRVALUEへのポインタかどうかを判別
    - is_pointer_to_heap() という関数で実装
    - どれかの heap_slots の範囲内かチェック
- スイープフェーズ
  - ヒープを全て辿って後始末，もしくはfreelist構築

# Rubyのメモリ管理
# 制約

- ソースコード，バイナリの互換性は厳守
  - 既存のソースコードを活かすため
  - 研究としては，取らないという選択肢もあるが…
- 移動が出来ない
  - コンパクションが出来ない
  - コピーが出来ない
- ライトバリアが出来ない
  - 世代別GCが実装できない
  - インクリメンタルGCが実装できない

# 関連研究
# Rubyのメモリ管理への挑戦

- Ruby処理系の拡張ポリシの変更が必要
  - 世代別GC [木山2001, 木山2002]
    → ライトバリアが必要
  - Mostly compaction [鵜川2009]
    → 効率的な実装にはいくつかの仮定が必要
  - インクリメンタルGC [相川2009]
    → ライトバリアが必要（ツールによるサポートの提案）
  - DDmalloc [Inoue2009]
    → たまに再起動が必要（ウェブアプリ的にはOK）
- そうでない工夫
  - ビットマップマーキング [中村2009]

# 検討
# GCオーバヘッドの検討

- GCにかかる実行時間Tgcについて検討
  - モデル
    - 生成するオブジェクトの数N
    - ヒープサイズH（オブジェクトの数）
    - 生きているオブジェクトの数：L（簡単のために固定）
  - マーク時間　　Tm = Wm*L
  - スイープ時間　Ts = Ws*H
  - GC回数　　Ngc= N / (H-L)
  - 総GC時間　　Tgc = N/(H-L) * (Wm * L + Ws * H)
    N/(1-L/H) * (L/H * Wm + Ws)

# 方針

**ヒープを大きくすると，総GC実行時間が下がる**
→ **できるだけヒープを大きく取りスループット向上**

- 問題
  1. 勝手にヒープを大きくすると，他のプロセスに迷惑がかかるかもしれない
  2. ヒープを大きくするとオブジェクトフラグメンテーションにより，必要なときに解放できない
  3. 広い区間をたどるので，スイープ時のメモリアクセス局所性が低下してしまう

# 検討
# 他プロセスと協調するヒープサイズ調整

- OSのメモリ状況を知る方法はOSに問い合わせる
  - /proc などで見ると，詳細情報は取れるが，正しくバランスさせるのは困難
- 自プロセスのメモリのページアウトを監視
  - ページアウトが1度でも発生したらメモリ逼迫と判断
  - POSIX の getrusage() システムコールで取得
    - **＊ 今回はこの方針でやってみたて成果が出るのだけど，ページキャッシュなど考えるとまだ検討が必要…**

- 関連研究沢山 [Grzegorczyk07, Yang04, Zhao09]
  - そもそも，ページアウトを起こさせない仕組みを検討
  - OS，ハードウェアに特殊な仕組みを要求

# 検討
# オブジェクトフラグメンテーション対策

1. heap_slots のサイズを 16KB から 4KB に変更
   - 断片化の可能性は減少
   - is_pointer_to_heap() などが若干高負荷に
2. 短寿命オブジェクトのみに広いヒープを用意
   - その他のオブジェクトは
   - オブジェクトの寿命予測を利用
   - 関連研究沢山 [Barrett93, Seidl98, Blackburn07,…]
     - 静的メモリ管理の効率化のために利用
     - GC では，オブジェクトの pre- tenuring に利用
     - **今回は「短寿命オブジェクト」だけを知りたい**のは新しいかも
   - プロファイルベースが必要か？　負荷は大丈夫？
   - 本研究では実装していないため，未検証

# 検討
# スイープの実行時間削減

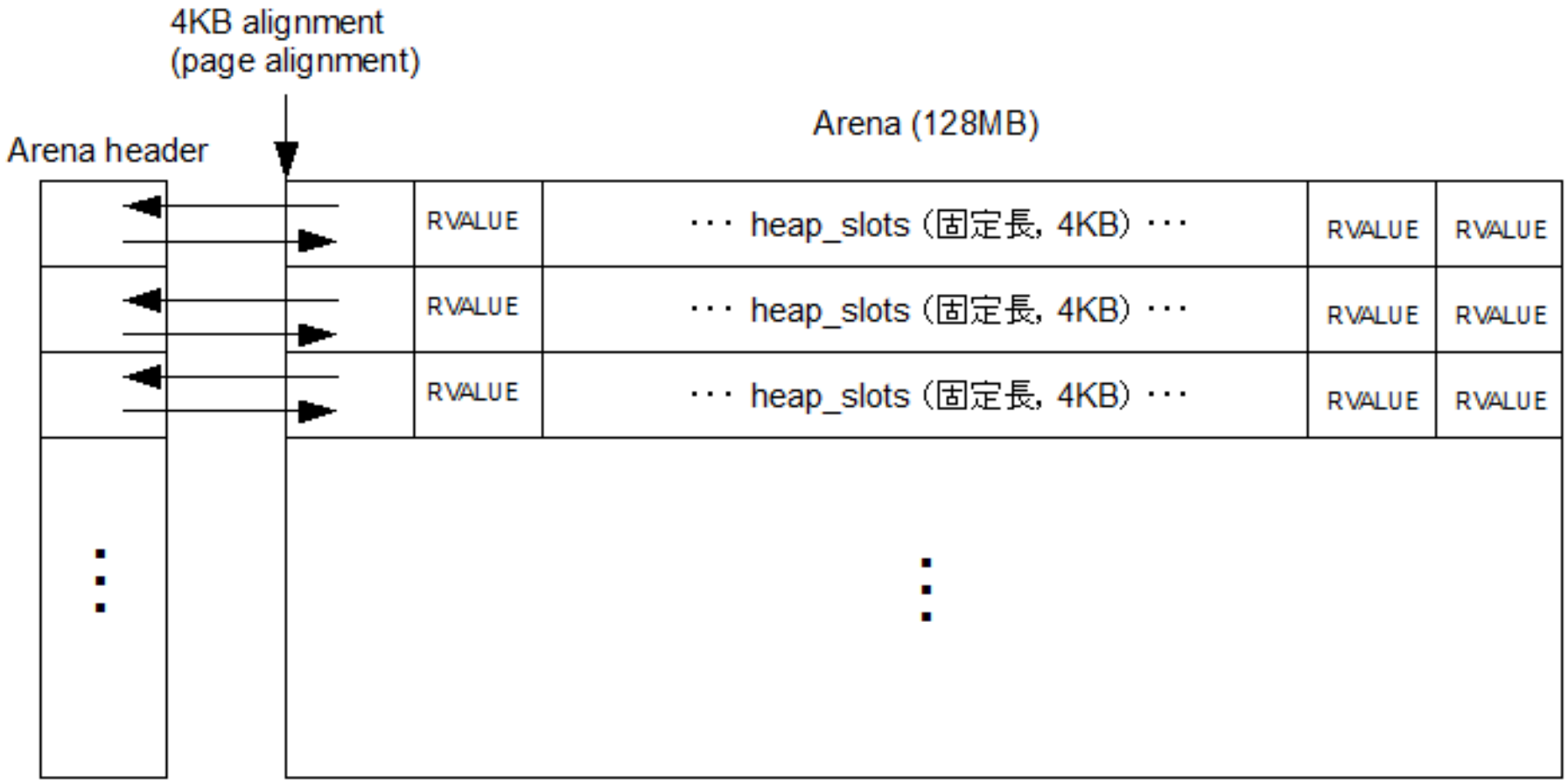
- heap_slots ごとに，2種類のカウンタを用意
  - MC - マークカウンタ：生きているObjectの数
  - FC - ファイナライザカウンタ：後始末必要なObjectの数
  - MC はマークフェーズで更新（若干の負荷増）

- MC, FCともに 0 なら，heap_slots ごと消去可能
  → heap_slots 内を一切触らずに消去が可能
  → メモリアクセスの局所性向上

# 実装

1. heap_slots の管理方法に Arena を導入
   - mmap() を利用してArenaを一気に確保（128MB）
     - アクセスされない限り実メモリを消費しない
   - heap_slots のサイズを 4KB （ページサイズ）に
     - heap_slots が 4KBアラインメントになって色々効率化
   - madvise() で heap_slotsごとに実メモリを直接解放
     - free() と比べて直接返却
   - heap_slots 追加，消去ポリシを変更
2. マークカウンタを効率的に実装
3. ページアウト監視を getrusage() を使って実装

# 実装
# Arena の導入


4KB alignment
(page alignment)
Arena header
Arena (128MB)
RVALUE
··· heap_slots (固定長, 4KB) ···
RVALUE
RVALUE

# 余談
# 本研究の最初の思惑と挫折

- Arena構造を導入して明示的なメモリ管理をすれば，メモリ管理全般が高速化するのでは？ → **しなかった**
- 期待
  - madvise() で即座に OS に返却することで，システムのスループットが向上するのでは
  - 4KB 管理でいろいろ効率化が出来るのでは
- 実際
  - madvise() はそれなりに遅い（ページテーブル弄るし，システムコールだし）ので，多発するとかなり遅かった
  - is_pointer_to_heap() とかはボトルネックじゃなかった
  - 4KBにしてフラグメンテーション解消があまり出来なかった

# 実装
# マークカウンタ

- 4KBアラインメントを利用して，heap_slots 固有のヘッダに対して簡単アクセス
  - (RVALUEへのポインタ) & ~(4KB)
  - ビットマップマーキングへの応用も簡単→今後の課題

- スイープ時，マークカウンタが 0 なら後始末や freelist への追加を行わずに即座に解放

  → 局所性の向上
- ただ，局所性の向上だけならLazy sweepでもいいのでは？ → 今後の課題

# 実装
# ヒープサイズの調整

- getrusage() により，メジャーフォールト数記録
  - heap_slots を要求する（使い切る）ごとにチェック
  - 前回の調査と比べて1回でも発生していればGC起動
- GCにより不要なheap_slotsが発生
  1. まずは何もしないで放っておく
  2. メジャーフォールト発生時，放っておいたheap_slots を madvise() で解放してまわる

  → 無駄な madvise() 回数を削減，必要なときだけ解放

# 評価

- 評価環境
  - ハードウェア
    - CPU: Intel Xeon E5335 2.00GHz (4 core) x 2 = 8 core
    - Memory: 2GB
  - ソフトウェア
    - OS: GNU/Linux 2.6.26-2-amd64
    - Cコンパイラ: gcc version 4.3.2 (Debian 4.3.2-1.1)
    - Ruby: ruby 1.9.2dev (2009-10-26 trunk 25491) [x86_64-linux]

# マイクロベンチマーク

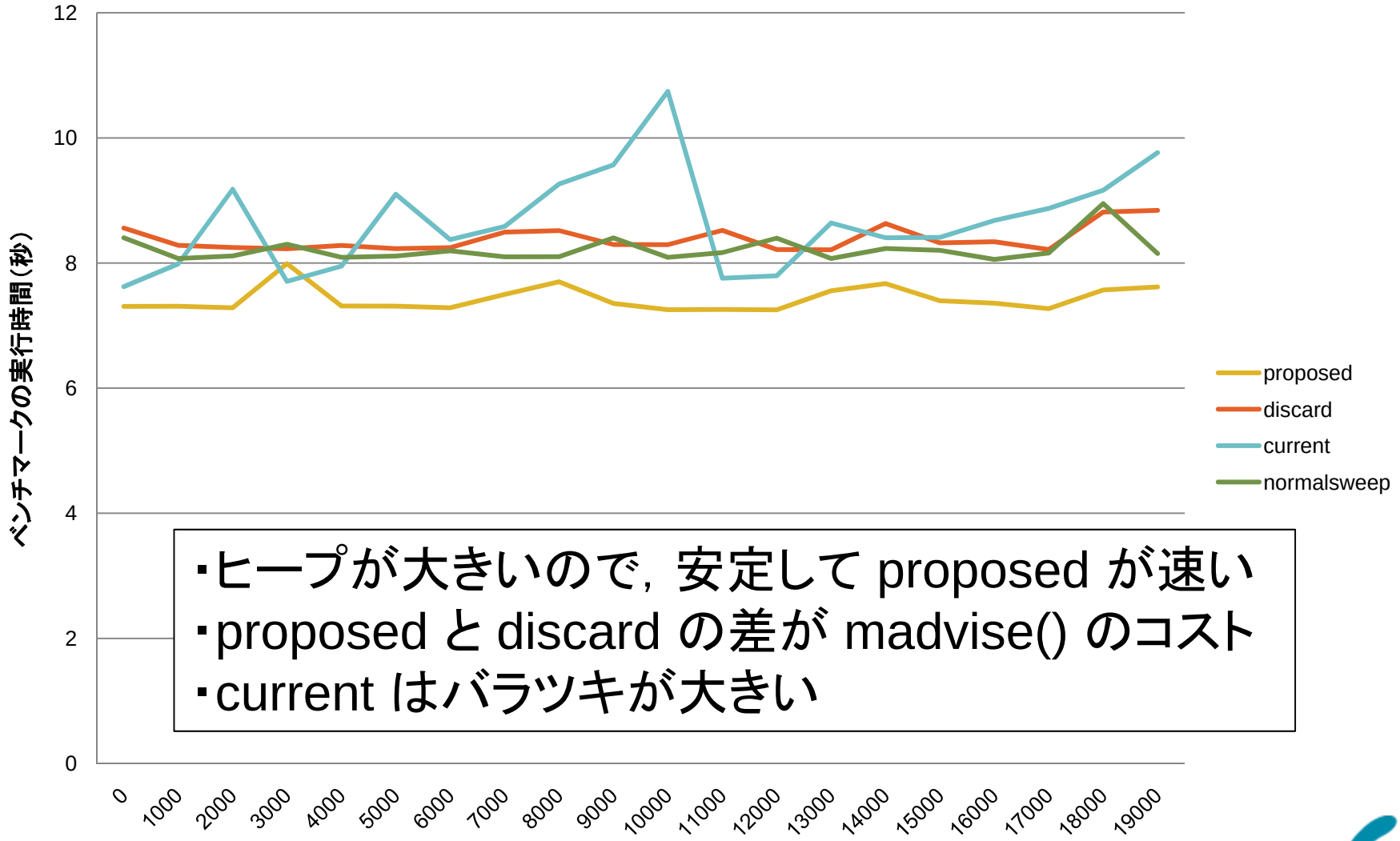
- 単純な浮動小数点数計算 → (2)
  - オブジェクト寿命予測をしていないから，短寿命オブジェクトを沢山生成したいため
  - ファイナライザカウンタを作っていないから，後始末の不要なオブジェクトを沢山生成したいため
  - 長寿命オブジェクトの数は調整可能にする → (1)

```
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
```

# 評価対象

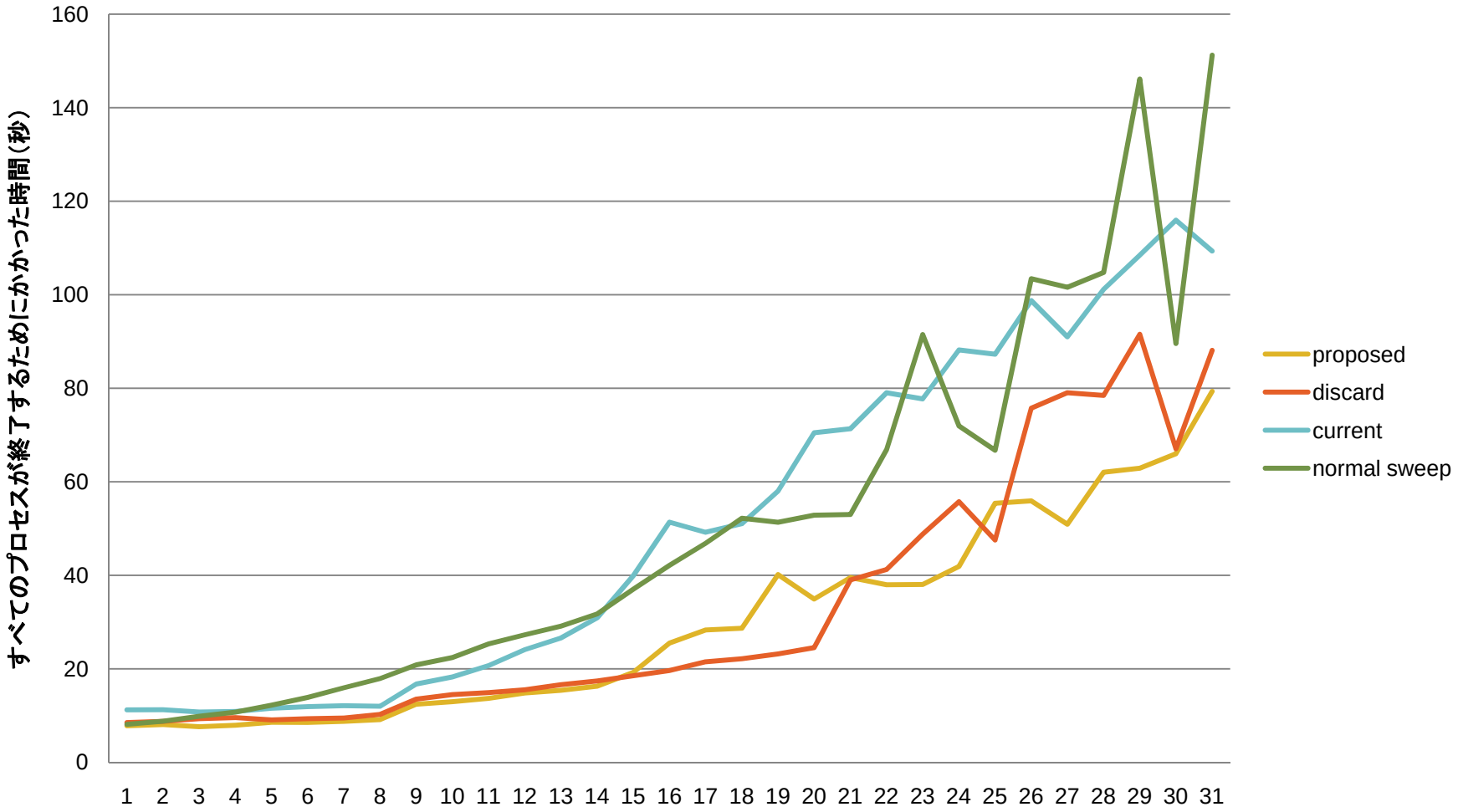
- proposed
  - Arena有り，マークカウント有り，page out 監視有り
  - 不要ページはpage out起こるまで解放しない
- discard
  - Arena有り，マークカウント有り，page out 監視有り
  - 不要ページは即座にmadivse()でOSに解放
- normalsweep
  - Arena有り，page out 監視有り
  - 従来通り，sweep で全部辿る（局所性悪化）
- current
  - 従来のインタプリタ

# 評価
# 長寿命オブジェクトの数と実行時間

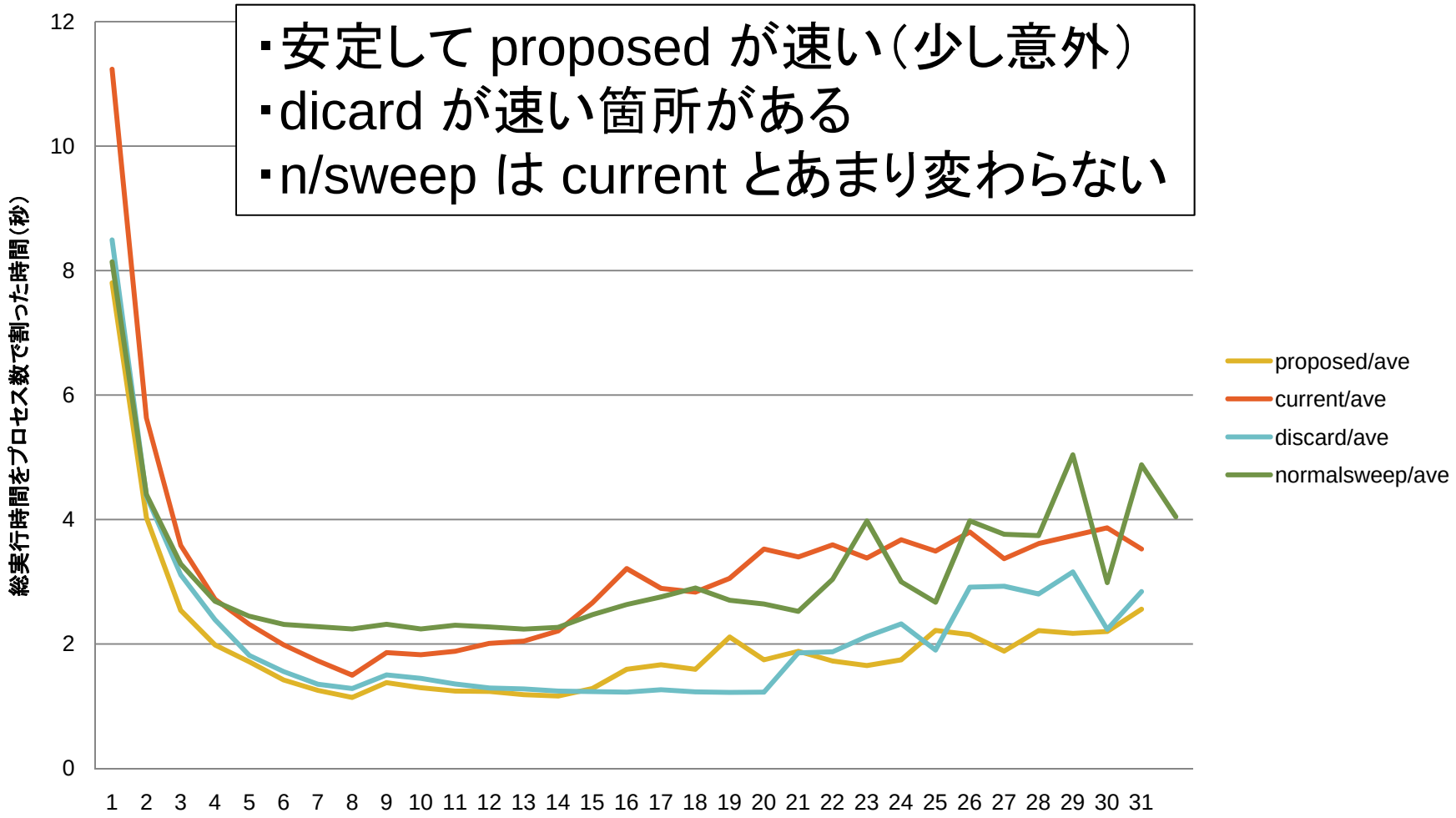



- ヒープが大きいので，安定して proposed が速い
- proposed と discard の差が madvise() のコスト
- current はバラツキが大きい

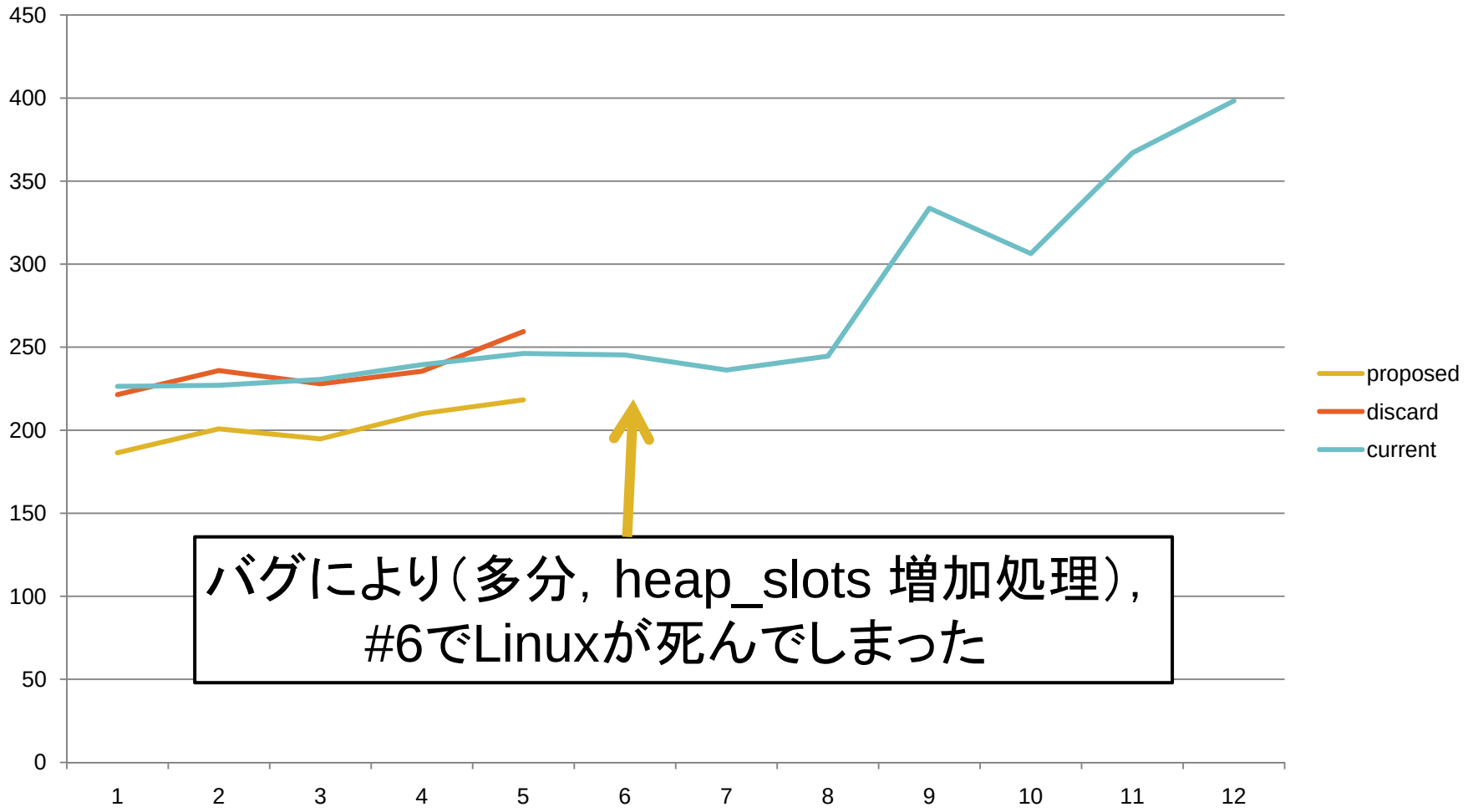
# 評価
# 複数プロセス同時実行（総実行時間）



# 評価
# 複数プロセス同時実行（平均時間）



# aobench での評価
# 現実的なレンダリングアプリケーション



# 考察

- ヒープサイズ大きくすれば速かった（当然）
- madvise() 回数削減は有効
- スイープ時にメモリを触らないようにする工夫は有効
  - Lazy sweep でも同様に有効かも？
  - Parallel sweep 実装は容易そう．比べてどうか？
- ページアウト監視は意外に有効かも？
  - ファイル操作を伴うプログラムが無い，同じ挙動をするものばかりなので，さらに検討が必要
- aobenchが駄目だったのは，結局フラグメンテーション解消は出来ていないのでは？
  - 寿命予測とか有効に働くだろうか？
  - テンポラリオブジェクト作成APIを作るのはどうか？

# 今後の課題

- 短寿命オブジェクトの発見手法の実現
  - プロファイルベースか？
  - 短寿命オブジェクトと明示的に示すAPIの設計？
- ファイナライザカウンタ関連
  - 後始末不要なデータ構造に変更
- ページアウト監視で良いのかの検討
  - メモリ消費自粛期間を設けるべきだったかも
- 開発者の立場から：Ruby 1.9.2 に向けて
  - mmap(), madvise() による Arena は取り入れたい
  - ヒープサイズ決定アルゴリズムは修正できそう
  - lazy sweep, bitmap marking は手軽なのでやりたい

# まとめ
# Rubyのメモリ管理の改善

- 制約から，ヒープサイズを大きくする方針で検討
  - ページアウトを監視してOSと協調するサイズ調整
  - マークカウンタによる，スイープ時の局所性向上
  - 4KB単位で管理することで断片化対策
- Arenaによるメモリ管理機構の一新
  - mmap(), madvise() によるOSと直接メモリやりとり
  - getrusage() によるページアウト監視
- 予備評価により，ある程度の成果を確認
  - ある程度効果を確認
  - さらなる実装，評価が必要