Fukuoka RubyKaigi 02 (2017/11/25)

Rubyにおける トレース機構の刷新

クックパッド株式会社
笹田 耕一
ko1@cookpad.com

cookpad

今日のトピック

- Ruby 2.5 では、TracePoint を「使ってない時」に高速化
  - 使ってるときも、そこそこ速い…？
- 高速化を考えるとき、どういうことを考えているのか、という話

笹田耕一

http://atdot.net/~ko1/

- プログラマ
  - 2006-2012 大学教員
  - 2012-2017 Heroku, Inc.
  - 2017- Cookpad Inc.
- 仕事：MRI 開発
  - MRI: Matz Ruby Interpreter
  - コアパート
    - VM, Threads, GC, etc

最近の仕事

- Feature #14045: Lazy Proc allocation for block parameters https://bugs.ruby-lang.org/issues/14045

```
def iter_yield
  yield
end

def iter_pass(&b)
  iter_yield(&b)
end
```

	r60392 (sec)	Modified (sec)
iter_pass * 10M	2.8	0.7

だいたい4倍くらい高速化

背景：TracePoint

- TracePoint (2.0 より前は set_trace_func)
  - Ruby プログラムにフックを仕掛ける仕組み
  - **いろいろなタイミング**でフックをかけることが可能
  - デバッガやトレーサに利用
  - カバレッジでも利用

:line:: execute code on a new line
:class:: start a class or module definition
:end:: finish a class or module definition
:call:: call a Ruby method
:return:: return from a Ruby method
:c_call:: call a C-language routine
:c_return:: return from a C-language routine
:raise:: raise an exception
:b_call:: event hook at block entry
:b_return:: event hook at block ending
:thread_begin:: event hook at thread beginning
:thread_end:: event hook at thread ending
:fiber_switch:: event hook at fiber switch

TracePoint の利用例（1）

```
TracePoint.new(:line){|tp|
  puts "trace> #{tp.path}:#{tp.lineno}"
}.enable do
  x = 1
  y = 2
  p x + y + 3
end
```

```
# output
trace> t.rb:4
trace> t.rb:5
trace> t.rb:6
6
```

TracePoint の利用例（2）

```
def foo
  yield
end

sp = ''
TracePoint.new(:call, :b_call){|tp|
  puts "#{sp}->#{tp.method_id}@#{tp.path}:#{tp.lineno}"
  sp << '  '
}.enable do
  TracePoint.new(:return, :b_return){|tp|
    sp.sub!('  ', '')
    puts "#{sp}<-#{tp.method_id}#{tp.path}:#{tp.lineno}" ¥
         "with #{tp.return_value.inspect}"
  }.enable do
    foo do
      1
    end
  end
end
```

```
# output
->@t.rb:9
  ->@t.rb:13
    ->foo@t.rb:1
      ->@t.rb:14
      <-t.rb:15 with 1
    <-foot.rb:2 with 1
  <-t.rb:14 with 1
```

TracePoint の悪用例（1）

```
TracePoint.new(:line){|tp|
  sleep 1
}.enable do
  x = 1
  y = 2
  p x + y + 3
end
```

```
# output
（3秒後）
6
```

TracePoint の悪用例（2）

```
TracePoint.new(:c_return){|tp|
  tp.return_value.downcase! ¥
    if tp.method_id == :upcase
}.enable do
  p 'foo'.upcase
end
```

メソッド名が :upcase なら
結果を downcase! しちゃう

```
# output
"foo"
```

背景：TracePointの実装

- Trace を呼び出す可能性がある場所でフラグセンス
  - 例
    - if (trace_flag & RUBY_TRACE_EVENT_LINE) {
      call_trace_hook(LINE_EVENT)
      }
- 問題：ふつーは TracePoint 使わないのに分岐は増える

高速化の一般論：偏りを利用

- 処理には偏りがある
  - 例：処理 A は 9 割、処理 B は1割
  - このとき、処理 A を高速化し、B は遅いままにする
- この一般論の適用例：例外処理
  - 例外処理はきっと「例外的」（滅多に動かない）　（はず…）

  → YARV では例外発生しなければ速い、発生したら遅い、ように実装

背景：TracePoint 利用の偏りを利用

- どーせ TracePoint は普段誰も使わない（偏り）
  → 使わない時に速く設計（使うときに多少遅くても気にしない）
- VMにtrace命令を導入（Ruby 1.9）
  - フックを起動する場所に trace 命令を埋め込み、フラグセンス
  - TracePoint を使わない時は排除可能（no overhead）
    - RubyVM::InstructionSequence.compile_option = {trace_instruction: false} で抑制（1割程度高速化）

trace 命令付きRubyの命令列

```
x=1
y=2
p x+y+3
```

命令列へ変換

```
# Ruby 2.4
0000 trace            1           (   2)
0002 putobject        1
0004 setlocal         x, 0
0007 trace            1           (   3)
0009 putobject        2
0011 setlocal         y, 0
0014 trace            1           (   4)
0016 putself
0017 getlocal         x, 0
0020 getlocal         y, 0
0023 send             :+
0027 putobject        3
0029 send             :+
0033 send             :p
0037 leave
```

背景：でも、誰も disable しない

- でも、デフォルトは enable
  - TracePoint を基本使えるようにするため
- → 誰も disable しない
  - 「1割速くなってもなあ」
  - 「こんな長いオプション知らないよ」
  - 「そもそも trace 命令とは？」
  - 「そもそも TracePoint って？」
- TracePoint を使わない時、trace 命令のせいでちょっと重い
  - VM 命令ディスパッチのオーバヘッドがかかるため

提案：TRACE命令をやめよう

HTTPS://BUGS.RUBY-LANG.ORG/ISSUES/14104

“REMOVE `TRACE` INSTRUCTIONS”

提案：trace 命令をやめよう

- TracePoint を使わない時は一切オーバヘッドがかからないように
- TracePoint を使うときは、命令を書き換えることでサポート
  - **ヒープに存在するすべての命令列**を書き換え
  - TracePoint を有効にするタイミングで大きなオーバヘッド
  - でも、使うこと滅多にないから良いよね？

Rubyの命令列

```
x=1
y=2
p x+y+3
```

命令列へ変換

```
# Ruby 2.4
0000 trace            1     (   2)
0002 putobject        1
0004 setlocal         x, 0
0007 trace            1     (   3)
0009 putobject        2
0011 setlocal         y, 0
0014 trace            1     (   4)
0016 putself
0017 getlocal         x, 0
0020 getlocal         y, 0
0023 send             :+
0027 putobject        3
0029 send             :+
0033 send             :p
0037 leave
```

```
x=1
y=2
p x+y+3
```

Rubyの命令列

```
# Ruby 2.4
0000 trace            1           (   2)
0002 putobject        1
0004 setlocal         x, 0
0007 trace            1           (   3)
0009 putobject        2
0011 setlocal         y, 0
0014 trace            1           (   4)
0016 putself
0017 getlocal         x, 0
0020 getlocal         y, 0
0023 send             :+
0027 putobject        3
0029 send             :+
0033 send             :p
0037 leave
```

```
# Ruby 2.5
0000 putobject        1           (   2)[Li]
0002 setlocal         x, 0
0005 putobject        2           (   3)[Li]
0007 setlocal         y, 0
0010 putself                      (   4)[Li]
0011 getlocal         x, 0
0014 getlocal         y, 0
0017 send             :+
0021 putobject        3
0023 send             :+
0027 send             :p
0031 leave

# trace 命令がない → 命令列長が短い
# 各行のところにイベント情報がある
```

```
x=1
y=2
p x+y+3
```

Rubyの命令列・トレース時

```
# Ruby 2.5
0000 putobject      1           (   2)[Li]
0002 setlocal       x, 0
0005 putobject      2           (   3)[Li]
0007 setlocal       y, 0
0010 putself                    (   4)[Li]
0011 getlocal       x, 0
0014 getlocal       y, 0
0017 send           :+
0021 putobject      3
0023 send           :+
0027 send           :p
0031 leave
```

```
# Ruby 2.5 / Trace on!
0000 trace_putobject 1          (   2)[Li]
0002 setlocal       x, 0
0005 trace_putobject 2          (   3)[Li]
0007 setlocal       y, 0
0010 trace_putself              (   4)[Li]
0011 getlocal       x, 0
0014 getlocal       y, 0
0017 send           :+
0021 putobject      3
0023 send           :+
0027 send           :p
0031 leave
```

「すべての命令列の各命令に対して」、「対応するtrace命令」に変更

実現方法

- すべての命令に対して、”trace_...” という命令を用意
  - VM生成系が自動的に作成（命令は手作業で作らない→バグ排除）
  - VMの命令数が2倍に
- TracePoint有効時に、ObjectSpace.each_object と同じ機能ですべての命令列を探し、”trace_...” 命令に変換

- この辺はだいたい二日くらい（自慢）
- デバッグやチューニングで2週間くらい

長所

- 長所　TracePoint 使わない時に速い
  - trace 命令が無くなったので速い
  - trace 命令がなくなったのでちょっと省メモリ

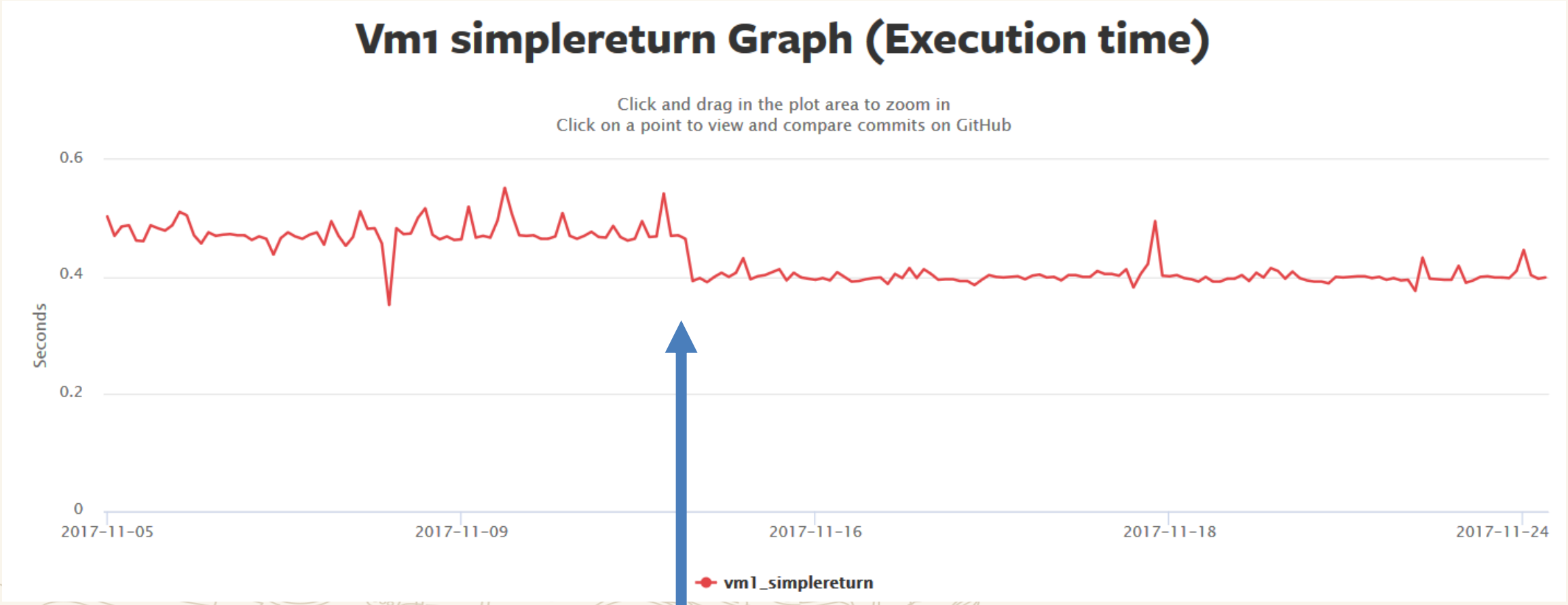
偏りを利用
だいたいオッケー

評価
マイクロベンチマーク

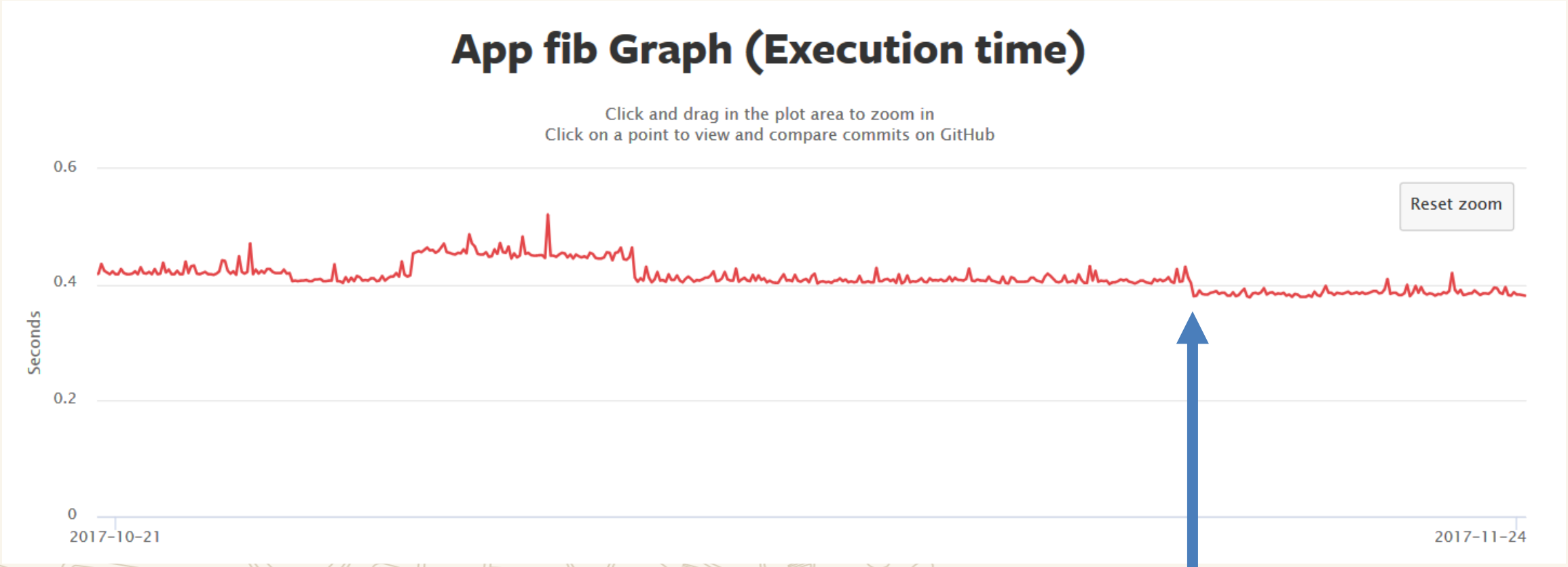
	Trunk (sec)	Modified (sec)	Speedup
Trace off	9.1	6.6	↑37%
Trace on	108.2	93.4	↑1.16%

```
require 'benchmark'

def foo n
end

N = 100_000_000
Benchmark.bm(10){|x|
  x.report('trace off'){
    N.times{
      foo(10)
      foo(10)
      foo(10)
    }
  }
  x.report('trace on'){
    TracePoint.new{}.enable
    N.times{
      foo(10)
      foo(10)
      foo(10)
    }
  }
}
```

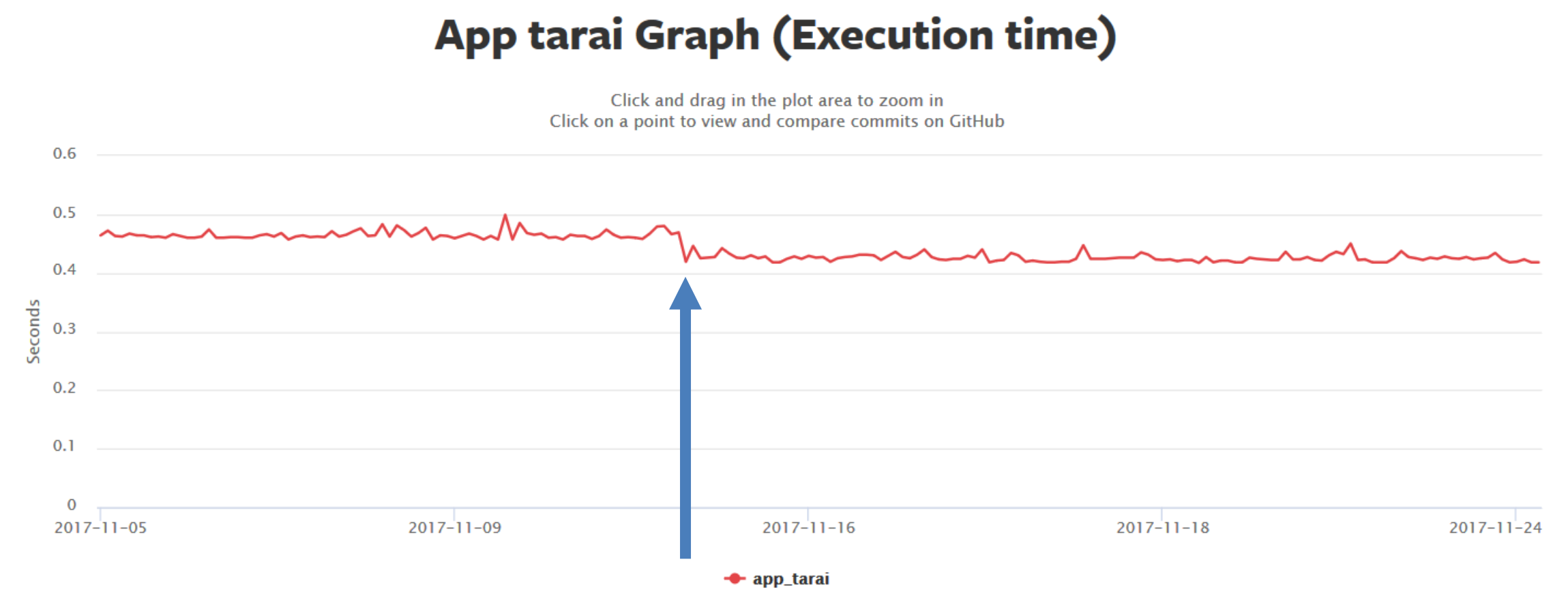
Ruby Long Running Benchmarks simplereturn



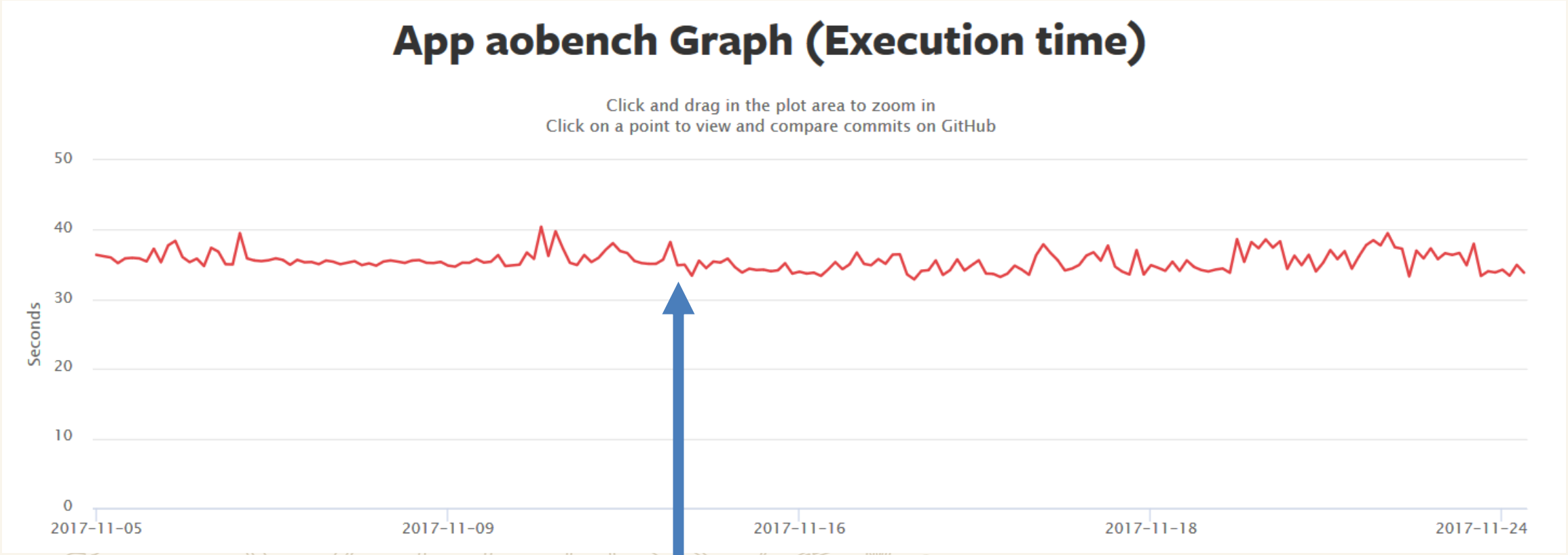
Ruby Long Running Benchmarks Fib



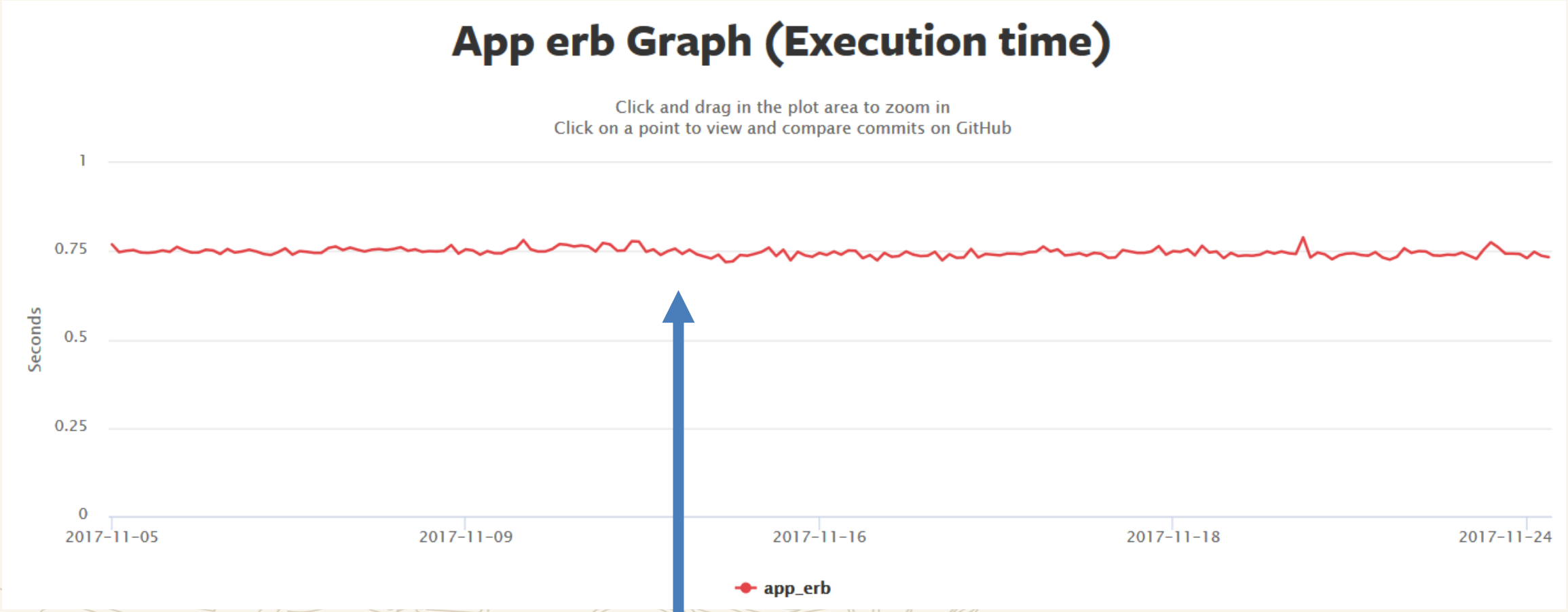
Ruby Long Running Benchmarks tarai



Ruby Long Running Benchmarks aobench



Ruby Long Running Benchmarks erb



Optcarrot:
A NES Emulator for Ruby Benchmark

	FPS (higher is better)	
r60762 （この変更前）	36.26	-
r60763 （この変更後）	39.92	10% improved

短所

- TracePoint 使うときに遅い
  - (1) enable / disable が遅い
  - (2) 各フックが遅い

もうちょっとチューニングが必要

チューニング
TracePoint の enable/disable が遅い問題

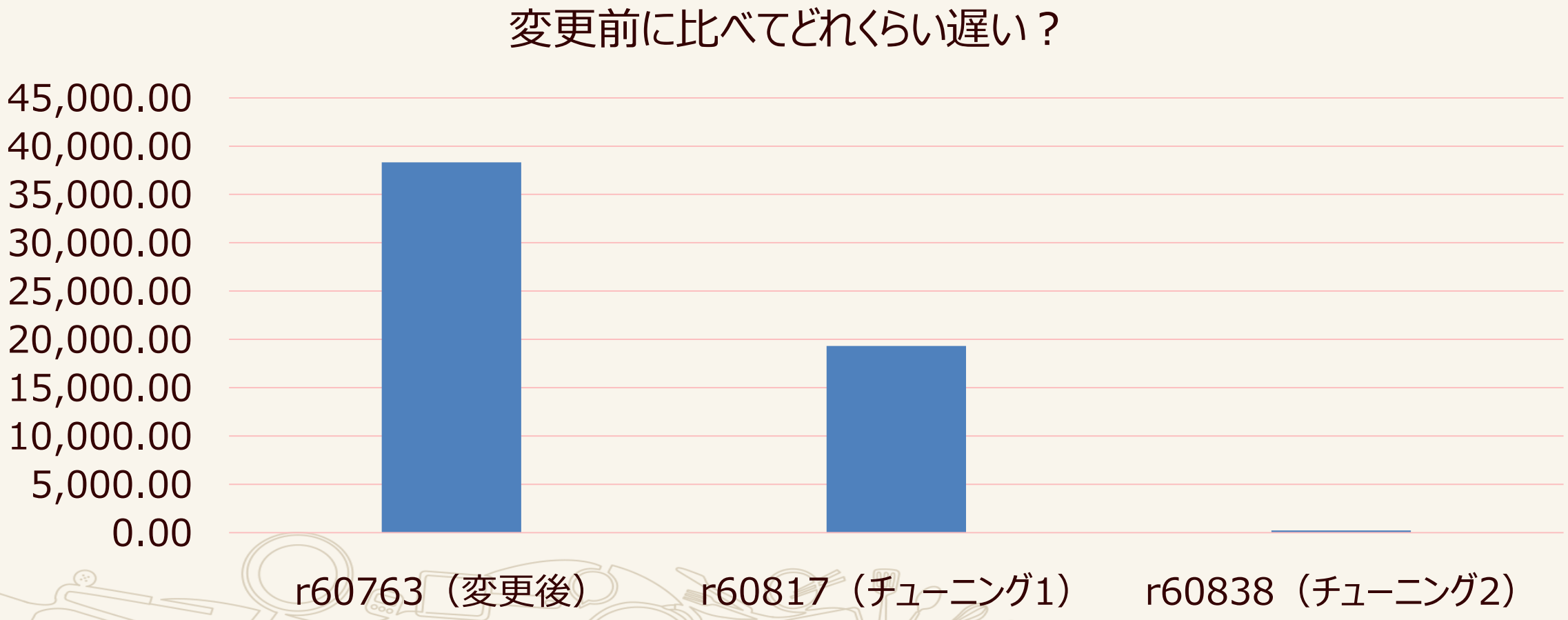
- enable/disable のたびに、全命令列を走査して命令書き換え
  - 誰も頻繁に on/off の切り替えをやらないだろう、と予想
    → power-assert がやってたらしい
- チューニング1：disable の遅延 r60817
  - trace_... 命令が、トレースが不要なタイミングで呼び出された→自分自身を trace_... なしに書き換え
- チューニング2：enable のスキップ r60838
  - すでに trace_... に変換されたものの場合、変換処理をスキップ
  - ObjectSpace.each_object 相当の時間はやっぱりかかる

TracePoint
enable/disable を繰り返すベンチマーク

	Execution time (sec)	Ratio compared with r60762 (x slower)
r60762（変更前）	0.02	-
r60763（変更後）	766.60	38,330.0
r60817（チューニング1）	386.33	19,316.5
r60838（チューニング2）	4.54	227.0

※0.1M 回の on/off を実行
※100行のメソッド1000個用意

TracePoint
enable/disable を繰り返すベンチマーク



各フックが重い→命令情報取得が重い

- “trace_…” 命令実行時に、自分の命令が行うべき情報を取得

```
# Ruby 2.5 / Trace on!
0000 trace_putobject 1          (   2)[Li]
0002 setlocal        x, 0
0005 trace_putobject 2          (   3)[Li]
0007 setlocal        y, 0
0010 trace_putself              (   4)[Li]
0011 getlocal        x, 0
0014 getlocal        y, 0
0017 send            :+
0021 putobject       3
0023 send            :+
0027 send            :p
0031 leave
```

```
# 命令情報テーブル
0000 2行目 Lineイベント
0005 3行目 Lineイベント
0010 4行目 Lineイベント
```

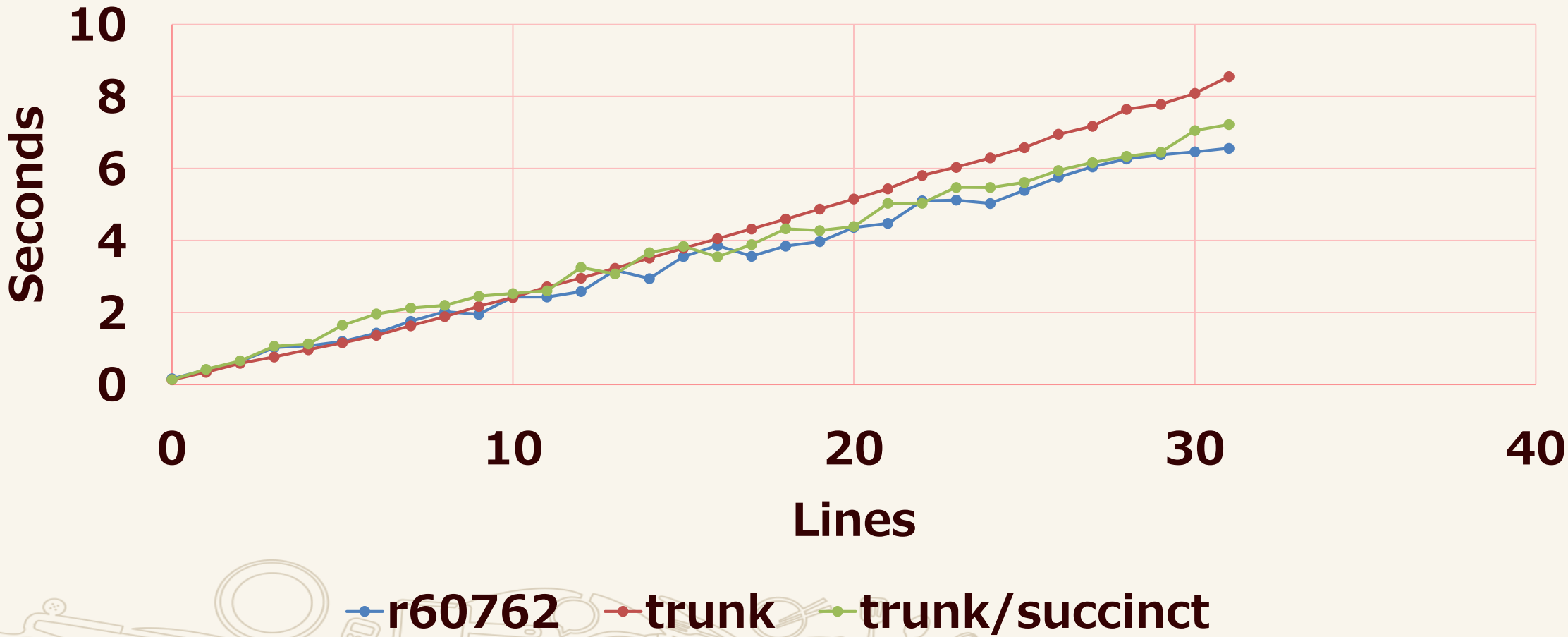
実行ごとにプログラムカウンタから検索
（O(n) で重い）

※trace 命令は Line イベントであることを
命令オペランドに持たせていたので検索が不要だった

解決策：簡潔データ構造の利用
（proposed by クックパッド・遠藤さん）

- 簡潔データ構造
  - あるビット列について、n 番目までに 1 が何個あったかを答える、速くて（O(1) ）コンパクトなデータ構造
  - 詳細はぐぐってね
- 素直に応用可能
  - 今回は「n 番目 → プログラムカウンタ」、「1 の個数→命令情報テーブルのインデックス」と対応づけ

ベンチマーク
（簡潔データ構造の利用）



今後の発展（1）

- その他の利点（1）柔軟なオンオフ
  - 命令書き換えなので、特定の箇所だけtraceの有効化・無効化が可能
- 「特定の箇所」の例
  - あるファイルの中だけ（例：特定ファイルのトレース）
  - あるメソッドの中だけ（例：特定メソッドにブレイクポイント）
  - ある行だけ（例：特定行にブレイクポイント）
- Ruby 2.5 では入りません
  - API の設計が難しい
  - 誰か考えませんか？

今後の発展（2）

- その他の利点（2）トレース追加が容易に
  - trace 命令を挿入しないため、オーバヘッドが無いから
- 「他にも TracePoint が欲しい！」という要望に答えられる
  - 例：callee じゃなくて caller （呼び出す直前）にフックしたい
  - 例：インスタンス変数/定数アクセスをフックしたい
- Ruby 2.5 では入りません
  - ユースケース大事
  - 誰か考えませんか？

振り返って
もう一度、（昔の）Ruby と向き合う

- 九州Ruby会議#1 2008/12「Ruby 1.9.1 に期待できること」
  - もう trace 命令はあったらしい…
- そもそもなんで trace 命令にしたの？
  - 楽だったから
  - 基本的に、実行時の命令書き換えは最後の手段だと思っていた
    （C への変換とかやりたかった）
- 約10年越しにキーノートのネタができた

今日のトピック

- Ruby 2.5 では、TracePoint を「使ってない時」に高速化
  - 使ってるときも、そこそこ速い…？
- 高速化を考えるとき、どういうことを考えているのか、という話
- 面白そうじゃないですか？
  - Ruby Hack Challenge も見てね
    http://techlife.cookpad.com/entry/2017/09/29/224024

Thank you for your attention

Koichi Sasada
<ko1@cookpad.com>

cookpad