Performance in the details: 
A way to make faster Ruby

Koichi Sasada
<ko1@heroku.com>

RailsClub 2015
A way to make faster Ruby

The only way I can find is:
Repeating a process.
A way to make faster Ruby: A process

1. Observe Ruby interpreter
2. Make assumption the reason of slowness
3. Consider ideas to overcome
4. Implement ideas
5. Measure the result
   • Bad/same performance → Goto 4, 3, 2 or 1
   • Good performance! → Commit it.
Koichi Sasada
A programmer from Japan
Koichi is a Programmer

• MRI committer since 2007/01
  • Original YARV developer since 2004/01
    • YARV: Yet Another RubyVM
    • Introduced into Ruby (MRI) 1.9.0 and later
  • Generational/incremental GC for 2.x
Koichi is an Employee
Koichi is a member of Heroku Matz team

Mission

Design Ruby language and improve quality of MRI

Heroku employs three full time Ruby core developers in Japan named “Matz team”
<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matz</td>
<td>Designer/director of Ruby</td>
</tr>
<tr>
<td>Nobu</td>
<td>Quite active committer</td>
</tr>
<tr>
<td>Ko1</td>
<td>Internal Hacker</td>
</tr>
</tbody>
</table>
Matz
Title collector

• He has so many (job) title
  • Chairman - Ruby Association
  • Fellow - NaCl
  • Chief architect, Ruby - Heroku
  • Research institute fellow – Rakuten
  • Chairman – NPO mruby Forum
  • Senior researcher – Kadokawa Ascii Research Lab
  • Visiting professor – Shimane University
  • Honorable citizen (living) – Matsue city
  • Honorable member – Nihon Ruby no Kai
  • ...

• This margin is too narrow to contain
Nobu
Great Patch monster

Ruby’s bug
|> Fix Ruby
|> Break Ruby
|> And Fix Ruby
Nobu
Patch monster

Commit count of MRI
Nobu
The Ruby Hero
Ko1
EDD developer

Commit number of ko1 (last 3 years)

- RubyConf 2013
- RubyKaigi 2013
- RubyCon 2012
- Ruby 2.0
- Euruko 2013

EDD: Event Driven Development
Heroku Matz team and Ruby core team
Recent achievement

Ruby 2.2

Current stable
Ruby 2.2
Syntax

• Symbol key of Hash literal can be quoted

```ruby
{"foo-bar": baz}
 #=> {:"foo-bar" => baz}
 #=> not {"foo-bar" => baz} like JSON
```

TRAP!!
Easy to misunderstand
(I wrote a wrong code, already...)
Ruby 2.2
Classes and Methods

• Some methods are introduces
  • Kernel#itself
  • String#unicode_normalize
  • Method#curry
  • Binding#receiver
  • Enumerable#slice_after, slice_before
  • File.birhtime
  • Etc.nprocessors
  • ...

...
Ruby 2.2
Improvements

• Improve GC
  • Symbol GC
  • Incremental GC
  • Improved promotion algorithm
    • Young objects promote after 4 GCs

• Fast keyword parameters

• Use frozen string literals if possible
Ruby 2.2
Symbol GC

before = Symbol.all_symbols.size
1_000_000.times{|i| i.to_s.to_sym} # Make 1M symbols
after = Symbol.all_symbols.size; p [before, after]

# Ruby 2.1
#=> [2_378, 1_002_378] # not GCed 😞

# Ruby 2.2
#=> [2_456, 2_456] # GCed! 😊
Ruby 2.2
Symbol GC Issues history

• **Ruby 2.2.0** has memory (object) leak problem
  • Symbols has corresponding String objects
  • Symbols are collected, but Strings are not collected! (leak)

• **Ruby 2.2.1** solved this problem!!
  • However, 2.2.1 also has problem (rarely you encounter BUG at **the end of process** [Bug #10933] ← not big issue, I want to believe)

• **Ruby 2.2.2** had solved [Bug #10933]!!
  • However, patch was forgot to introduce!!

• Finally, **Ruby 2.2.3** solved it!!
  • Please use newest version!!
Ruby 2.2
Fast keyword parameters

“Keyword parameters” introduced in Ruby 2.0 is useful, but slow!!

Evaluation on Ruby 2.1

Repeat 10M times

foo6(1, 2, 3, 4, 5, 6)  x30 slower  foo_kw6(k1: 1, k2: 2, k3: 3, k4: 4, k5: 5, k6: 6)
Ruby 2.2
Fast keyword parameters

Ruby 2.2 optimizes method dispatch with keyword parameters

Repeat 10M times

Ruby 2.1  Ruby 2.2

foo6(1, 2, 3, 4, 5, 6)  foo_kw6(k1: 1, k2: 2, k3: 3, k4: 4, k5: 5, k6: 6)

Execution time (sec)

x14 faster!!

But still x2 times slower compare with normal dispatch
Ruby 2.2
Incremental GC

<table>
<thead>
<tr>
<th></th>
<th>Before Ruby 2.1</th>
<th>Ruby 2.1 RGenGC</th>
<th>Incremental GC</th>
<th>Ruby 2.2 Gen+IncGC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Pause time</td>
<td>Long</td>
<td>Long</td>
<td>Short</td>
<td>Short</td>
</tr>
</tbody>
</table>
RGenGC from Ruby 2.1: Micro-benchmark

![Bar chart showing comparison between no and RGenGC]

- **no**
  - Total mark: 1699.805974 ms
  - Total sweep: 704.843669 ms

- **RGenGC**
  - Total mark: 867.740319 ms
  - Total sweep: 87.230735 ms

x2.5 faster
RGenGC from Ruby 2.1: Pause time

Most of cases, FASTER 😊

(w/o rgengc)
RGenGC from Ruby 2.1: Pause time

Several peaks 😞

(w/o rgengc)
Ruby 2.2 Incremental GC

Short pause time 😊
Heroku Matz team and Ruby core team
Next target is

Ruby 2.3
Heroku Matz team and Ruby core team
Next target is

Ruby 2.3

No time to talk about it.
Please ask me later 😊
Performance in the details: A way to make faster Ruby
Ruby’s components for users

Ruby (Rails) app

So many gems
such as Rails, pry, thin, ... and so on.

RubyGems/Bundler

Ruby interpreter

i gigantum umeris insidentes
Standing on the shoulders of giants
Ruby’s components from core developer’s perspective

- **Ruby script**
- **Parse**
- **Compile**
- **Ruby Bytecode**
- **Bundled Libraries**
- **Gem Libraries**
- **Embedded classes and methods (Array, String, ...)**
- **Threading**
- **Evaluator**
- **Object management (GC)**

Interpret on RubyVM

Ko1’s area
Basic flow to make faster Ruby

1. Observe Ruby interpreter
2. Make assumption the reason of slowness
3. Consider ideas to overcome
4. Implement ideas
5. Measure the result
   • Bad/same performance → Goto 4, 3, 2 or 1
   • Good performance! → Commit it.
Basic weapons to overcome issues

• Knowledge of computer science
  • Computer system, Programming techniques, and many others
  • From:
    • Textbook
    • Academic papers
    • Other implementation

• Feedback from users
Basic technique to improve performance

• Change the algorithm to reduce computation complexity
  • e.g.: Selection sort (O(n^2)) v.s. Quick sort (O(n log(n))

• Change the data structure to improve data locality
  • e.g.: “list” and “array”

• Remove redundant process
  • e.g.: Using cache (utilize time locality)

• Considering trade-off
  • Speed-up major cases and slow-down minor cases
  • e.g.: speed-up non-exception flow (and slow-down exception cases)

• Machine dependent technique
  • e.g.: Using assembler / CPU register directly

• …
Case studies
Ruby has many

*    *    *    *

Let’s play hangman game
Ruby has many
Ruby has many

****
Ruby has many

FU***
Ruby has many

FU***

___
Ruby has many Or Methods
Case study:
Optimize method dispatch
Ruby’s components from core developer’s perspective

- Ruby script
- Parse
- Compile
- Ruby Bytecode
- Interpret on RubyVM
- Bundled Libraries
- Gem Libraries
- Embedded classes and methods (Array, String, ...)
- Threading
- Evaluator
- Object management (GC)

Ko1’s area
Method dispatch

# Example
recv.selector(arg1, arg2)

•recv: receiver
•selector: method id
•arg1, arg2: arguments
Method dispatch
Overview

1. Get class of `recv` (`klass`)
2. Search method `body` named `selector` from `klass`
   • Method is not fixed at compile time
   • “Dynamic” method dispatch
3. Dispatch method with `body`
   1. Check visibility
   2. Check arity (expected args # and given args #)
   3. Store `PC` and `SP` to continue after method returning
   4. Build `local environment`
   5. Set program counter
4. And continue VM execution
Overview
Method search

• Search method from ‘klass’
  1. Search method table of ‘klass’
     1. if method ‘body’ is found, return ‘body’
     2. ‘klass’ = super class of ‘klass’ and repeat it
  2. If no method is given, exceptional flow
     • In Ruby language, ‘method_missing’ will be called
Overview

Checking arity and visibility

• Checking arity
  • Compare with given argument number and expected argument number

• Checking visibility
  • In Ruby language, there are three visibilities
  • can you explain each of them :-p
    • public
    • private
    • protected
Overview
Building `local environment`

• How to maintain local variables?
  → Prepare `local variables space` in stack
  → `local environment` (short `env`)
• Parameters are also in `env`
Method dispatch
Overview (again)

1. Get class of `recv` (`klass`)
2. Search method `body` `selector` from `klass`
   • Method is not fixed at compile time
   • “Dynamic” method dispatch
3. Dispatch method with `body`
   1. Check visibility
   2. Check arity (expected args # and given args #)
   3. Store `PC` and `SP` to continue after method returning
   4. Build `local environment`
   5. Set program counter
4. And continue VM execution

It seems very easy and simple! and slow...
Method dispatch

• Quiz: How many steps in Ruby’s method dispatch?
  • Hint: More complex than I explained overview
    ① 8 steps
    ② 12 steps
    ③ 16 steps
    ④ 20 steps

Answer is
About ④ 20 steps
Method dispatch
Ruby’s case

1. Check caller’s arguments
   1. Check splat (*args)
   2. Check block (given by compile time or block parameter (&block))

2. Get class of ‘recv’ (‘klass’)

3. Search method ‘body’ ‘selector’ from ‘klass’
   • Method is not fixed at compile time
   • “Dynamic” method dispatch

4. Dispatch method with ‘body’
   1. Check visibility
   2. Check arity (expected args # and given args #) and process
      1. Post arguments
      2. Optional arguments
      3. Rest argument
      4. Keyword arguments
      5. Block argument
   3. Push new control frame
      1. Store ‘PC’ and ‘SP’ to continue after method returning
      2. Store ‘block information’
      3. Store ‘defined class’
      4. Store bytecode info (iseq)
      5. Store recv as self
   4. Build ‘local environment’
   5. Initialize local variables by ‘nil’
   6. Set program counter
   5. And continue VM execution

(*) Underlined items are additional process
Ruby’s case
Complex parameter checking

• “def foo(m1, m2, o1=..., o2=..., p1, p2, *rest, &block)”
  • m1, m2: mandatory parameter
  • o1, o2: optional parameter
  • p1, p2: post parameter
  • rest: rest parameter
  • block: block parameter

• From Ruby 2.0, keyword parameter is supported
Method dispatch

1. Check caller’s arguments
   1. Check splat (*args)
   2. Check block (given by compile time or block parameter (&block))

2. Get class of ‘recv’ (‘klass’)

3. Search method ‘body’ ‘selector’ from ‘klass’
   • Method is not fixed at compile time
   • “Dynamic” method dispatch

4. Dispatch method with ‘body’
   1. Check visibility
   2. Check arity (expected args # and given args #) and process
      1. Post arguments
      2. Optional arguments
      3. Rest argument
      4. Keyword arguments
      5. Block argument
   3. Push new control frame
      1. Store ‘PC’ and ‘SP’ to continue after method returning
      2. Store ‘block information’
      3. Store ‘defined class’
      4. Store bytecode info (iseq)
      5. Store recv as self
   4. Build ‘local environment’
   5. Initialize local variables by ‘nil’
   6. Set program counter
   5. And continue VM execution

Complex and Slow!!!
Method dispatch overhead is big especially on micro-benchmarks 😊

Fib

\[\text{Others} < \text{Obj} < \text{Insn} < \text{VM} < \text{Cfunc} < \text{NotRuby}\]

Pentomino

\[\text{Others} < \text{Obj} < \text{Insn} < \text{VM} < \text{Cfunc} < \text{NotRuby}\]

OS: Linux 2.6.31 32-bit
CPU: IntelCore2Quad 2.66GHz
Mem: 4GB
C Compiler: GCC 4.4.1, -O3
Profiled by Oprofile

ruby 1.9.3dev (2010-05-26)
Profiled by Mr. Shiba
Speedup techniques for method dispatch

1. Specialized instructions
2. Method caching
3. Caching checking results
4. Special path for `send’ and `method_missing’
Optimization
Specialized instruction (from Ruby 1.9.0)
• Make special VM instruction for several methods
  • +, -, *, /, ...

```ruby
def opt_plus(recv, obj)
  if recv.is_a?(Fixnum) and obj.is_a?(Fixnum) and Fixnum#+ is not redefined
    return Fixnum.plus(recv, obj)
  else
    return recv.send(:+, obj) # not prepared
  end
end```
Optimization
Method caching (from Ruby 1.9.0)

• **Eliminate method search overhead**
  • Reuse search result
  • Invalidate cache entry with VM stat

• Two level method caching
  • Inline method caching
  • Global method caching
Optimization
Caching checking results (from 2.0.0)

• Idea: Visibility and arity check can be skipped after first checking
  • Store result in inline method cache

1. Check caller’s arguments
2. Search method `body` `selector` from `klass`

3. Dispatch method with `body`
   1. Check visibility and arity
      1. Cache result into inline method cache
   2. Push new control frame
   3. Build `local environment`
   4. Initialize local variables by `nil`

Second time

First time
Evaluation result
Micro benchmarks

Faster than first date

Speedup ratio

trunk 2012/10/13

Faster than first date

trunk 2012/10/31
Case study
Faster keyword parameters
Keyword parameters from Ruby 2.0

```ruby
# def with keywords

def foo(a, b, key1: 1, key2: 2)
  ...
end

# call with keywords

foo(1, 2, key1: 123, key2: 456)
```
Slow keyword parameters

Evaluation on Ruby 2.1

foo6(1, 2, 3, 4, 5, 6)
foo_kw6(k1: 1, k2: 2, k3: 3, k4: 4, k5: 5, k6: 6)

Repeat 10M times

Execution time (sec)

x30 slower
Why slow, compare with normal parameters?

1. Hash creation
2. Hash access

```ruby
def foo(h = {})
  k1 = h.fetch(:k1, v1)
  k2 = h.fetch(:k2, v2)
  ...
end
foo(k1: 1, k2: 2)
```

```ruby
def foo(k1: v1, k2: v2)
  ...
end
foo(k1: 1, k2: 2)
```
Optimization technique of keyword parameters from Ruby 2.2

• Key technique
  → Pass “a keyword list” instead of a Hash object

Check “Evolution of Keyword parameters” at Rubyconf portugal'15
Result: Fast keyword parameters (Ruby 2.2.0)

Ruby 2.2 optimizes method dispatch with keyword parameters

foo6(1, 2, 3, 4, 5, 6)

foo_kw6(k1: 1, k2: 2, k3: 3, k4: 4, k5: 5, k6: 6)

x14 faster!!
(best case)

Repeat 10M times

Ruby 2.1  
Ruby 2.2

But still x2 times slower
compare with normal dispatch
Case study
Garbage collection

http://www.flickr.com/photos/circasassy/6817999189/
Ruby’s components from core developer’s perspective

- Ruby script
- Parse
- Compile
- Ruby Bytecode
- Evaluator
- Threading
- Embedded classes and methods (Array, String, ...)
- Object management (GC)
- Bundled Libraries
- Gem Libraries

Interpret on RubyVM

Ko1’s area
Automatic memory management
Basic concept

• Garbage collector recycled “unused” objects automatically
Mark & Sweep algorithm

1. Mark reachable objects from root objects

2. Sweep unmarked objects (collection and de-allocation)

Collect unreachable objects
Generational GC (GenGC) from Ruby 2.1.0

- Weak generational hypothesis:
  
  “Most objects die young”

→ Concentrate reclamation effort only on the young objects
Generational hypothesis

Object lifetime in RDoc
(How many GCs surviving?)

95% of objects dead by the first GC
Generational GC (GenGC)

• Separate young generation and old generation
  • Create objects as young generation
  • Promote to old generation after surviving n-th GC
• Usually, GC on young space (minor GC)
• GC on both spaces if no memory (major/full GC)
GenGC [Minor M&S GC] (1/2)

- Mark reachable objects from root objects.
  - Mark and promote to old generation
  - Stop traversing after old objects

→ Reduce mark overhead
- Sweep not (marked or old) objects

- Can’t collect Some unreachable objects

Don’t collect old object even if it is unreachable.
GenGC [Minor M&S GC] (2/2)

- Mark reachable objects from root objects.
  - Mark and **promote to old generation**
  - Stop traversing after old objects

→ **Reduce mark overhead**
- Sweep not (marked or old) objects

- Can’t collect Some unreachable objects

Don’t collect old object even if it is unreachable.
GenGC [Major M&S GC]

- Normal M&S
- Mark reachable objects from root objects
  - Mark and promote to old gen
- Sweep unmarked objects
  - *Sweep all unreachable (unused) objects*
RGenGC from Ruby 2.1.0
Performance evaluation (RDoc)

About x15 speedup!

Total mark time (ms)
Total sweep time (sec)

- w/o RGenGC
- RGenGC

* Disabled lazy sweep to measure correctly.
RGenGC from Ruby 2.1.0
Performance evaluation (RDoc)

* 12% improvements compare with w/ and w/o RGenGC
* Disabled lazy sweep to measure correctly.
Summary
Summary
Repeating “Basic flow” is my daily job

1. Observe Ruby interpreter
2. Make assumption the reason of slowness
3. Consider ideas to overcome
4. Implement ideas
5. Measure the result
   • Bad/same performance → Goto 4, 3, 2 or 1
   • Good performance! → Commit it.
Ruby/MRI is getting better and better.
Thank you for your attention

Koichi Sasada
<ko1@heroku.com>