Implementation Details of Ruby 2.0 VM

Koichi Sasada
Disclaimer

• (As you can see) I can speak English little.
  – It’s my 8th RubyConf
  – 7th time disclaimer

• Ask me an questions in 日本語, Japanese (WELCOME!), Ruby or SLOW English

• All of I want to say is on the screen. You can read them.
Who am I?

• Koichi Sasada (笹田耕一)
  – Matz team at Heroku, Inc.
    • Full-time CRuby development
  – CRuby/MRI committer
    • Virtual machine (YARV) from Ruby 1.9
    • YARV development since 2004/1/1
  – 2.0 Release manager assistant
    • Organizing feature request
      – 200 tickets → 100 tickets last week!
    • Over 150 mails to ruby-core/ruby-dev
      – Sorry for spamming
Ruby 2.0

20\textsuperscript{th} Anniversary Release of Ruby language
Ruby 2.0 Release policy

• Compatibility (Ruby level)
• Compatibility (Ruby level)
• Compatibility (Ruby level)
• Usability
• Performance

ADD (Anniversary Driven Development)
Ruby 2.0 Roadmap

2012/Aug
“Big-feature” freeze (was ignored)

2012/Oct
Feature freeze

2012/Nov
Now Preview1

2012/Dec
Preview2

2012/Dec
X’mas Code freeze

2013/Jan
RC1

2013/Feb
RC2

2013/2/24
Ruby 2.0 Release (20th anniversary)

“[ruby-core:40301] A rough release schedule for 2.0.0” and Endo-san’s (release manager) leak
Introduction of Ruby 2.0 features

What is introduced?
You can read them at NEWS file
goto :next_topic

Change the title of this presentation to...
Lecture series of Computer Science

How to make interpreter?

#3 Method dispatch

Prof. Koichi Sasada (*1)
Akihabara University (*2)

*1: Prof. means ...

*2: Of course, joking. No such University 🎈
Review slide

Requirement and Assumption

• You need to finish “Ruby language basic” course
• This course uses “Ruby” language/interpreter
  – One of the most popular languages
  – Used in world-wide programming
    • Web application
    • Text processing
    • and everything!!
  – CRuby
    • Ruby has many alternative implementations
    • CRuby has their own VM
Review slide
How to implement virtual machine?

• Execute instructions
  – Execute compiled instructions (bytecodes)
  – Pointed by “Program counter” (PC)

• Stack machine architecture
  – All of values on the stack
  – Stack top is pointed by “Stack pointer” (SP)

  • V.S. Register machine architecture
    • Advantages and disadvantages
Review slide
Stack machine execution (basic)

Ruby Program
a = b + c

YARV Instructions
getlocal b
getlocal c
send +
setlocal a

YARV Stack
PC
SP
local env
b+c
b
c

Animation with Powerpoint
[Advanced] Optimization techniques

- Peephole optimizations (compiler technique)
  - Reduce instruction number
- Make macro instructions
  - Operand unification
  - Instruction unification
- Direct threading
  - Using GCC specific feature
- Stack caching
  - n-level stack caching
  - Impact on CPU’s branch prediction
[Advanced] VM generator

VM generator enables flexible VM building

- Dis-assembler
- Assembler
- Documents

VM Instruction Description

Virtual Machine

Compiler (Optimizer)
Today’s lecture:
Method dispatch

# Example
recv.selector(arg1, arg2)

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

1. Evaluate `recv`
2. Evaluate `arg1` and `arg2`
3. Method dispatch (`selector`)

# Ruby’s disassembled bytecodes of Ruby 2.0 trunk

0016 getlocal recv, 0 # 1 receiver
0019 getlocal arg1, 0 # 2 arg1
0022 getlocal arg2, 0 # 2 arg2
0025 send <callinfo!mid:selector, argc:2, ARG'S_SKIP>
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

BasicObject
 Kernel
 Object
 C1
 C2

Each Class has method table
selector: body ...
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`
Overview
Building `local environment`

Stack before method dispatch:
- `arg2`
- `arg1`
- `recv`

Stack after method dispatch:
- `loc2`
- `loc1`
- `arg2`
- `arg1`
- `recv`

`sp`: Pointer to stack
`ep`: Environment pointer
`env`: Pointer to all local variables

All local variables are accessible with `ep`:
(ep[0], ep[-1], ep[-2], ...)

`recv` argument

Method dispatch
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

**About 7 steps**

It seems very easy and simple! and slow...
Method dispatch
Ruby’s case

• Quiz: How many steps in Ruby’s case?
  – 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

... simple?
Ruby’s case

4. Dispatch method with `body`

• Previous explanation is for Ruby methods
  – `body’ (defined as rb_method_definition_t in method.h) has several types at least the following two types:
    • Method defined by Ruby code
    • Method defined by C function (in C-extension)

• Quiz: How many method types in CRuby?
  – Hint: At least 2 types (Ruby method and C method)
    ① 3 types
    ② 6 types
    ③ 9 types
    ④ 11 types

Answer is
About ④ 11 types
Ruby’s case
Method types

1. **VM_METHOD_TYPE_ISEQ**: Ruby method (using `def` keyword)
2. **VM_METHOD_TYPE_CFUNC**: C method
3. **VM_METHOD_TYPE_ATTRSET**: defined by `@attr_accessor`
4. **VM_METHOD_TYPE_IVAR**: defined by `@attr_reader`
5. **VM_METHOD_TYPE_BMETHOD**: defined by `define_method`
6. **VM_METHOD_TYPE_ZSUPER**: used in internal
7. **VM_METHOD_TYPE_UNDEF**: `undef`ed method
8. **VM_METHOD_TYPE_NOTIMPLEMENTED**: not implemented
9. **VM_METHOD_TYPE_OPTIMIZED**: optimization
10. **VM_METHOD_TYPE_MISSING**: method_missing type
11. **VM_METHOD_TYPE_CFUNC_FRAMELESS**: optimization two

There are 11\textsuperscript{th} different method dispatch procedure
(dispatch by switch/case statement)
Quiz: I introduce (virtual) registers `pc’, `sp’ and `ep’. How many registers in virtual machine (in Ruby 1.9.x)?

① 4 registers
② 6 registers
③ 9 registers
④ 11 registers

Answer is
About ④ 11 registers
↓
Need to store/restore 11 registers each method call
Ruby’s case
Store registers

- Introduce “control frame stack” to store registers
  - To store `pc', `sp', `ep' and other information, VM has another stack named “control frame stack”
  - Not required structure, but it makes VM simple → Easy to maintain

```c
/* 1.9.3 */
typedef struct {
  VALUE *pc; /* cf[0] */
  VALUE *sp; /* cf[1] */
  VALUE *bp; /* cf[2] */
  rb_iseq_t *iseq; /* cf[3] */
  VALUE flag; /* cf[4] */
  VALUE self; /* cf[5] / block[0] */
  const rb_method_entry_t *me; /* cf[10] */
} rb_control_frame_t;

/* 2.0 */
typedef struct {
  VALUE *pc; /* cf[0] */
  VALUE *sp; /* cf[1] */
  VALUE *bp; /* cf[2] */
  rb_iseq_t *iseq; /* cf[3] */
  VALUE flag; /* cf[4] */
  VALUE self; /* cf[5] / block[0] */
  rb_iseq_t *block_iseq; /* cf[9] */
  VALUE proc; /* cf[9] */
  const rb_method_entry_t *me; /* cf[10] */
} rb_control_frame_t;
```

11 regs

10 regs
reduced, but many yet
Ruby 1.9 VM stacks structure

```
def foo
  bar{
    ...
  }
end
```

```
def bar
  yield
end
```

```
def foo
  bar
  yield
end
```

```
def foo
  yield
end
```

```
def bar
  yield
end
```

rb_thread_t::cfp
points current control frame
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
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

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

Fib

Pentomino

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
Homework

• Report about “Method Dispatch speedup techniques”
  1. Analyze method dispatch overhead on your favorite application
  2. Survey method dispatch speed-up techniques
  3. Propose your optimization techniques to improve method dispatch performance
  4. Implement techniques and evaluate their performance

• Deadline: 2012/11/15 (Thu) 23:59 JST
• Submit to: Koichi Sasada <ko1@rvm.jp>
• This report is important for your grade of this course!
Lecture was finished 😊

Presentation is not finished
Report
“Optimization techniques for Ruby’s method dispatch”

Koichi Sasada
Speedup techniques for method dispatch

1. Specialized instructions
2. Method caching

3. Caching checking results
4. Frameless CFUNC method
5. Special path for `send` and `method_missing`

Introduced techniques from Ruby 2.0
Today’s main subject 😊

Note that these optimizations may not be my original.
Method dispatch overheads

1. Check caller’s arguments
2. Search method `body’ `selector’ from `klass’
3. Dispatch method with `body’
   1. Check visibility and arity
   2. Push new control frame
   3. Build `local environment’
   4. Initialize local variables by `nil’
Optimization
Specialized instruction (from 1.9)

• Make special VM instruction for several methods
  \(- +, -, *, /, \ldots\)

```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
Specialized instruction

• Pros.
  – Eliminate all of dispatch cost (very effective)

• Cons.
  – Limited applicability
    • Limited classes, limited selectors
    • Tradeoff of VM instruction numbers
  – Additional overhead when not prepared class
Optimization

Method caching

• **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)

• 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`
Optimization
Frameless CFUNC (from 2.0)

• Introduce “Frameless” CFUNC methods
  – Idea: Several CFUNC doesn’t need method frame
      • For example, String#length doesn’t need method frame. It only return the size of given String

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

3. Dispatch method with `body`
   1. Check visibility and arity
   2. **Push new control frame**
   3. **Build `local environment’**
   4. **Initialize local variables by `nil’**
Optimization
Eliminate frame building (from 2.0)

• Compare with specialized instruction
  – Pros.
    • You can define unlimited number of frameless methods
  – Cons.
    • A bit slow compare with specialized instruction

• Note that evaluation result I will show you doesn’t include this technique
Optimization
Special path for `send` and `method_missing` (from 2.0)

Before
- send(:foo)
  - invoke send method (cfunc)
  - invoke `foo` method

After
- send(:foo)
  - search `foo`
  - invoke `foo` method
Evaluation results

Applications

Faster than first date

Speedup ratio

0
0.2
0.4
0.6
0.8
1
1.2
1.4


app_aobench
app_erb
app_factorial
app_fib
app_mandelbrot
app_pentomino
app_raise
app_strconcat
app_tak
app_tarai
app_uri

trunk 2012/10/13
trunk 2012/10/31
Future work

• Restructure “method frame”
  – Reduce required information per frame
• Improve “yield” performance
  – Using something cached
Conclusion
Method dispatch speed-up

• Ruby’s method dispatch is nightmare
  – Too complex

• Speedup upto 50% at simple method dispatch with new optimizations

• Need more effort to achieve performance improvements
Other optimizations from 2.0

- Introducing Flonum (only on 64bit OSs)
- Lightweight Backtrace capturing
- Re-structure VM stacks/ISeq data

- Bitmap marking garbage collection (by nari3)
- “require” performance (not by me)
Introducing Flonum
(only on 64bit CPU)

• Problem: Float objects are not immediate on Ruby 1.9
  – It causes GC overhead problem
• To speedup floating calculation, represent Float object as immediate object
  – Specified range Float objects are represented as immediate object (Flonum) like Fixnum
    • $1.72723e^{-77} < |f| < 1.15792e^{77}$ (approximately) and $+0.0$
    • Out of this range and all Floats on 32bit CPU are allocated in heap
  – No more GCs! (in most of case)
  – Flonum and old Float are also Float classes
  – Proposed by [K.Sasada 2008]
  – On 64bit CPU, object representation was changed
Benchmark results
Flonum: Float in Heap (1.9 or before)

All of Float object are allocated in heap

Data structure in heap contains IEEE754/double

On 64bit CPU

- HEAD
  - T_FLOAT
  - Float
  - etc

- VALUE

- IEEE754 Double

8B x 6w = 48 byte for Float object
Flonum: Encoding
IEEE754 double floating number

64bit double

\[-1^s2^{e-1024}m \quad (m = 1 + \sum_{i=0}^{51} \frac{b_i}{2^{52-i}})\]
Flonum: Range

<table>
<thead>
<tr>
<th>b63</th>
<th>b60-b62</th>
<th>b60-b52</th>
<th>b51-b0</th>
</tr>
</thead>
</table>

Check if e (b52 to b62) is within 768 to 1279, then it can be represented in Flonum.

This check can be done with b60-b62.

(+0.0 (0x00) is special case to detect)
Flonum: Encoding

IEEE754 double

Ruby’s Flonum

Flonum representation bits (2 bits)
#define FLONUM_P(v) ((v&3) == 2)

☆ +0.0 is special case (0x02)
Flonum:
Object representation on VALUE

<table>
<thead>
<tr>
<th></th>
<th>Non Flonum</th>
<th>Flonum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixnum</td>
<td>...xxxx xxx1</td>
<td>...xxxx xxx1</td>
</tr>
<tr>
<td>Flonum</td>
<td>N/A</td>
<td>...xxxx xx10</td>
</tr>
<tr>
<td>Symbol</td>
<td>...xxxx 0000 1110</td>
<td>...xxxx 0000 1100</td>
</tr>
<tr>
<td>Qfalse</td>
<td>...0000 0000</td>
<td>...0000 0000</td>
</tr>
<tr>
<td>Qnil</td>
<td>...0000 0100</td>
<td>...0000 1000</td>
</tr>
<tr>
<td>Qtrue</td>
<td>...0000 0010</td>
<td>...0001 0100</td>
</tr>
<tr>
<td>Qundef</td>
<td>...0000 0110</td>
<td>...0011 0100</td>
</tr>
<tr>
<td>Pointer</td>
<td>...xxxx xx00</td>
<td>....xxxx x000</td>
</tr>
</tbody>
</table>

57
Lightweight Backtrace capturing

- Backtrace is Array of String objects
  - [“file:lineno method”, ...]
- Idea: Capture only ISeqs and translate to String (file and line) only when it is accessed
  - Backtrace information may be ignored
After Ruby 2.0

What we should do?
Performance overhead

**rdoc**

- VM
- Method
- Others
- Cfunc
- MM
- Others

**Rails**

- VM
- Others
- NotRuby
- MM
- Others
- Cfunc

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

rudy 1.9.3dev (2010-05-26)
Profiled by Mr. Shiba
VM techniques

• On Rails and other applications, VM is not an bottleneck

• On Mathematic, Symbolic computation, VM is matter
  – To speedup then, we need compilation framework
    • 2.0?
Object Allocation and garbage collection

• Lightweight object allocation
  – Sophisticate object creation
  – Create objects in non-GC managed area

• Sophisticate Garbage collection
  – Per-type garbage collection
  – Generational garbage collection
    • Introduce write barriers with dependable techniques
Parallelization

- Multiple processes
- Multiple VMs
- Multiple Threads

No time and space to discuss about them!
Conclusion

Our challenge has just begun!!

俺たちの戦いはまだ始まったばかりだ!
Thank you for your attention

Koichi Sasada
Heroku, Inc.
ko1@heroku.com