ORACLE®

What makes TruffleRuby run Optcarrot 9 times faster than MRI?

Petr Chalupa
Principal Member of Technical Staff
Oracle Labs
February 4, 2017

ORACLE

Safe Harbor Statement

The following is intended to provide some insight into a line of research in Oracle Labs. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. Oracle reserves the right to alter its development plans and practices at any time, and the development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.

ORACLE

Outline

1. Optcarrot
2. TruffleRuby
3. Optimizations

Optcarrot

Nintendo Entertainment System emulator

Optcarrot

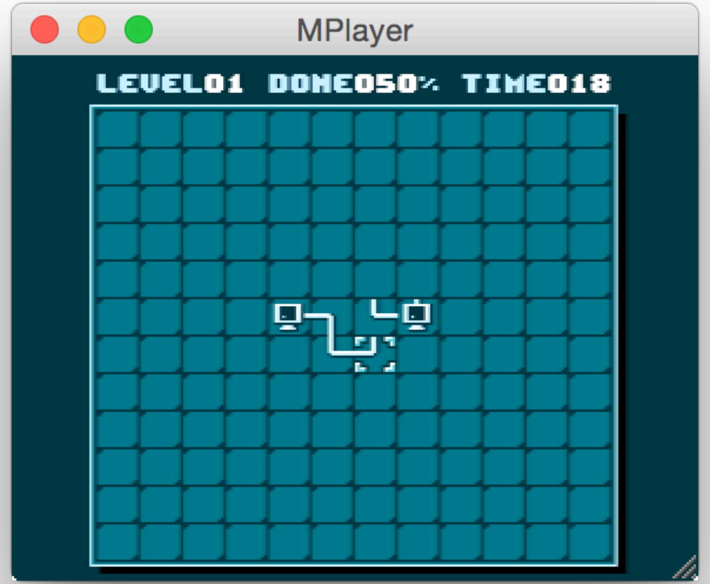
- **NES** Emulator
  - 8-bit, CPU, PPU, 2kB RAM, 2kB VRAM
  - Released in 1983
  - github.com/mame/optcarrot
- A benchmark created to drive Ruby MRI 3x3 improvements
- It runs the Lan Master game

Nintendo Entertainment System

https://en.wikipedia.org/wiki/Nintendo_Entertainment_System#/media/File:NES-Console-Set.jpg

Lets play

- Using:
  - MRI
  - TruffleRuby



Lan Master

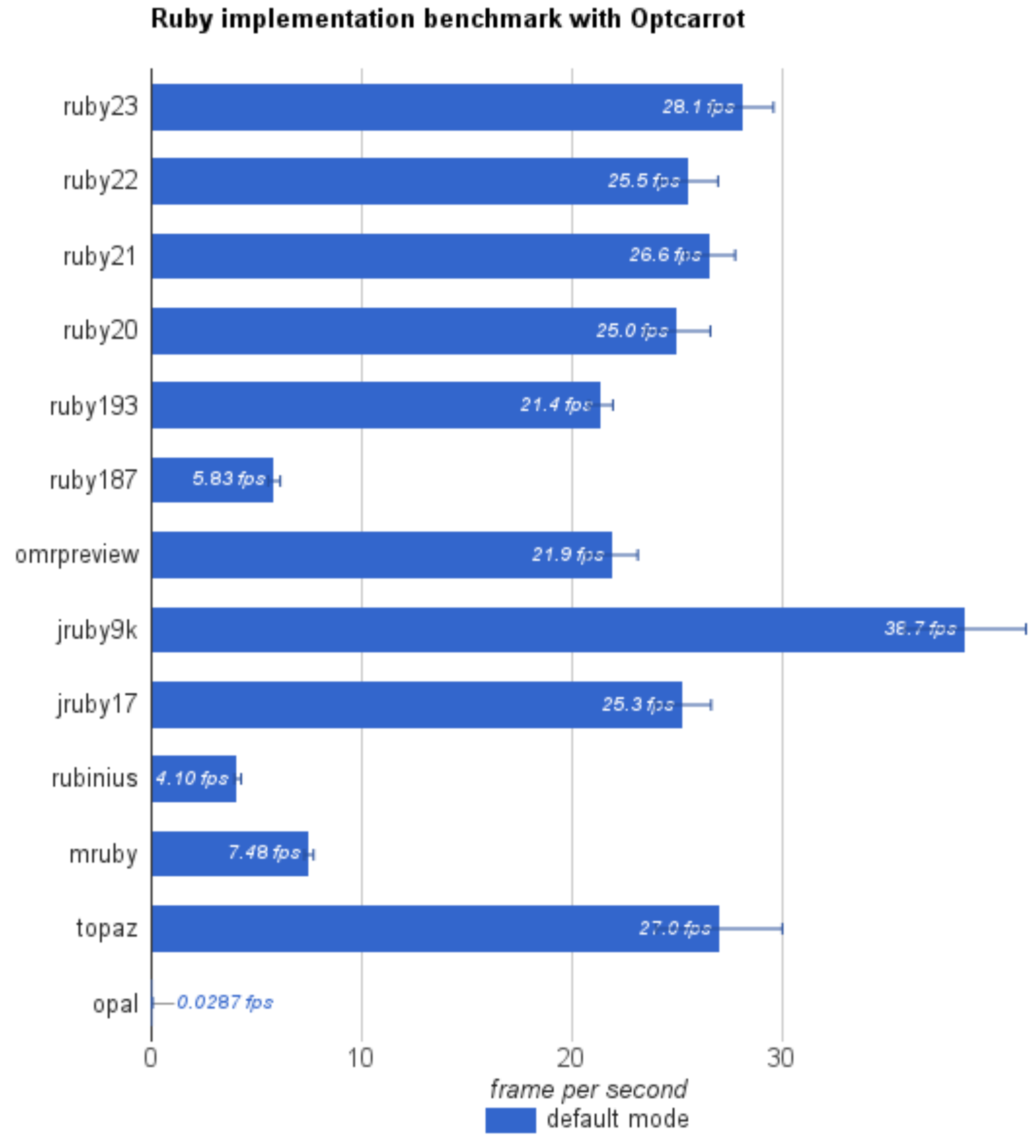
MRI 2.4.0

TruffleRuby

Results

Published results

- Without TruffleRuby
- 180 frames



https://raw.githubusercontent.com/mame/optcarrot/master/doc/benchmark-default.png

Benchmarking

- Implementations:
  - MRI 2.0
  - MRI 2.4
  - JRuby, 9.0 indy server
  - TruffleRuby, GraalVM 0.19
- Options
  - 6000 frames
  - Headless

ORACLE

Benchmarking

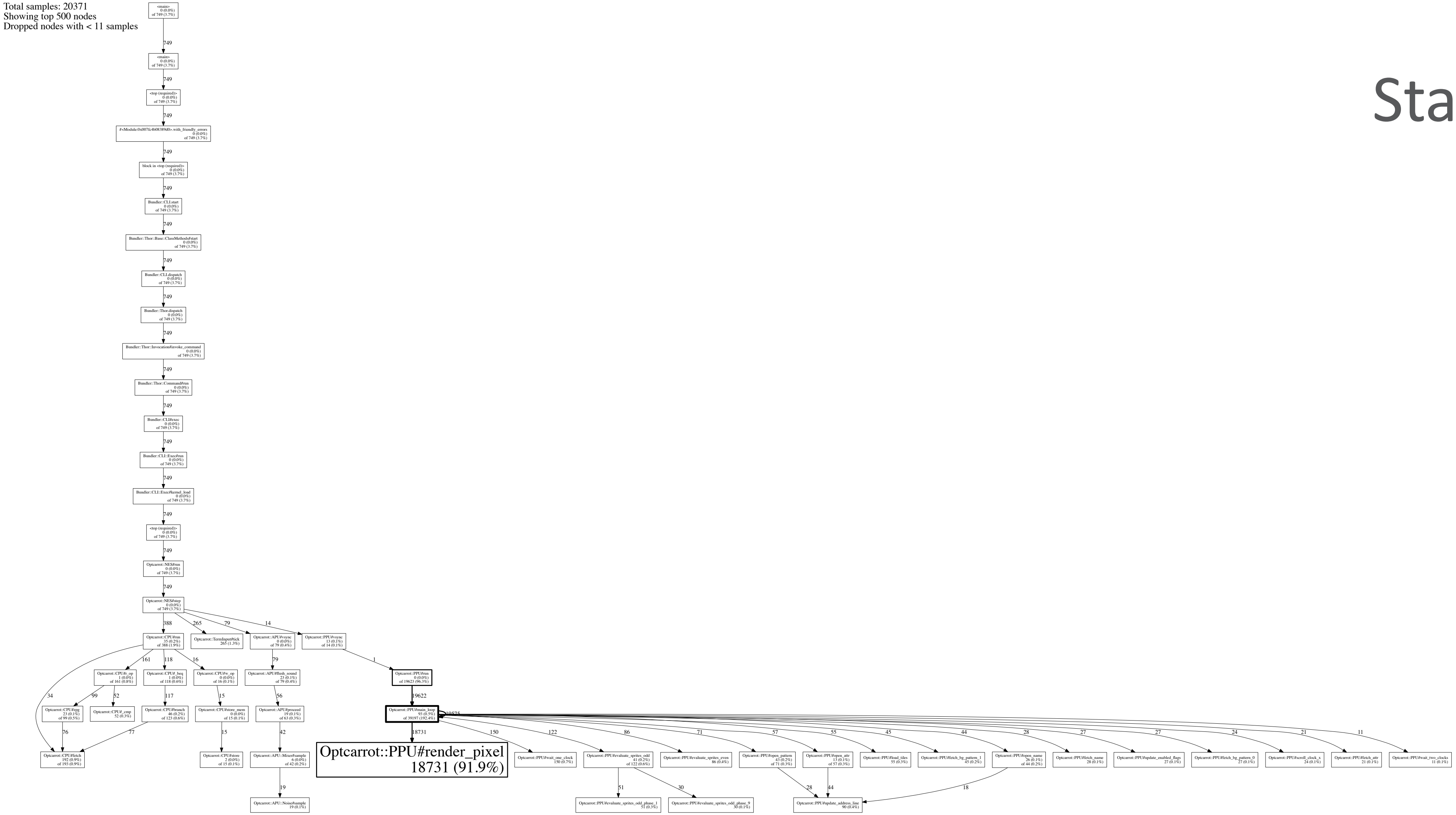
9.5 times faster

ORACLE

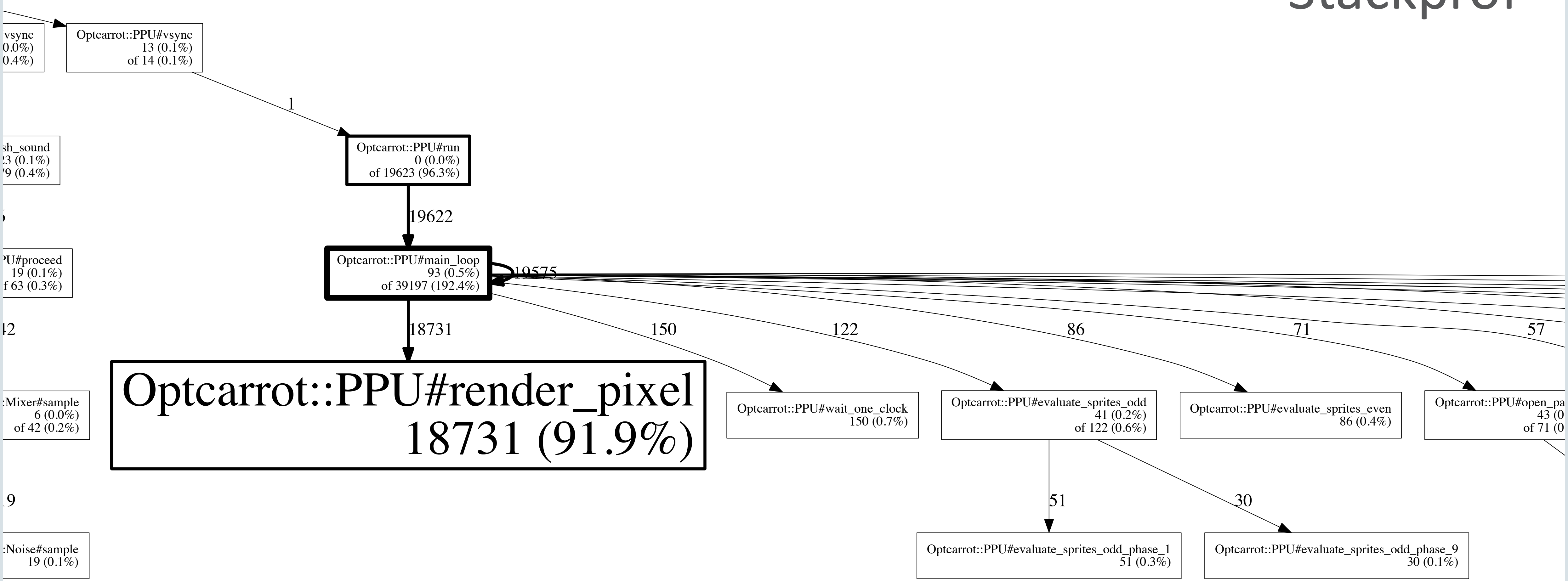
Optcarrot

Closer look

Stackprof

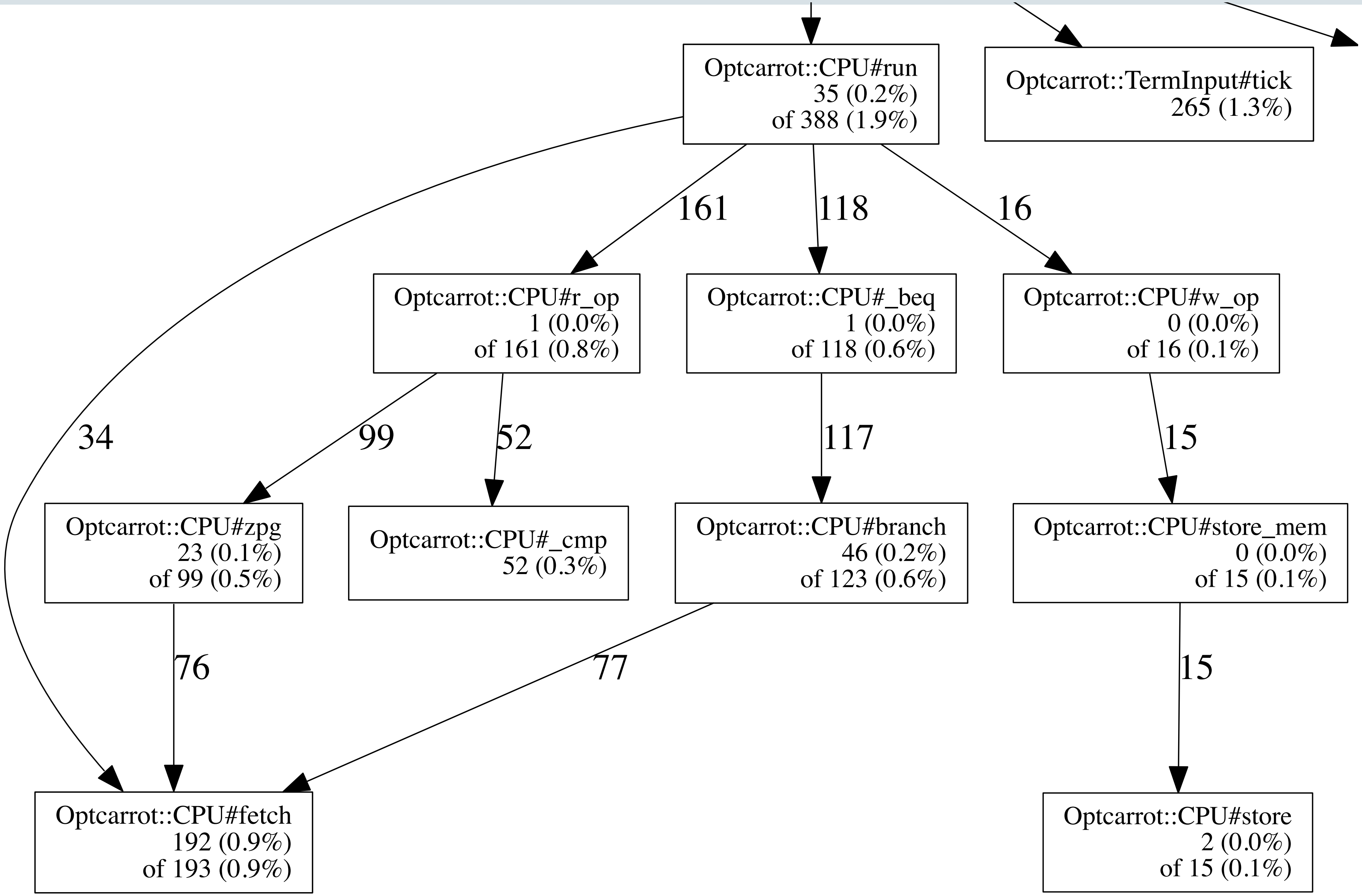

Total samples: 20371
Showing top 500 nodes
Dropped nodes with < 11 samples
Optcarrot::PPU#render_pixel
18731 (91.9%)

Stackprof



vsync
0.0%)
0.4%)
Optcarrot::PPU#vsync
13 (0.1%)
of 14 (0.1%)
1
sh_sound
3 (0.1%)
9 (0.4%)
Optcarrot::PPU#run
0 (0.0%)
of 19623 (96.3%)
19622
U#proceed
19 (0.1%)
f 63 (0.3%)
Optcarrot::PPU#main_loop
93 (0.5%)
of 39197 (192.4%)
19575
18731
150
122
86
71
57
Mixer#sample
6 (0.0%)
of 42 (0.2%)
Optcarrot::PPU#render_pixel
18731 (91.9%)
Optcarrot::PPU#wait_one_clock
150 (0.7%)
Optcarrot::PPU#evaluate_sprites_odd
41 (0.2%)
of 122 (0.6%)
Optcarrot::PPU#evaluate_sprites_even
86 (0.4%)
Optcarrot::PPU#open_pa
43 (0
of 71 (0
51
30
Noise#sample
19 (0.1%)
Optcarrot::PPU#evaluate_sprites_odd_phase_1
51 (0.3%)
Optcarrot::PPU#evaluate_sprites_odd_phase_9
30 (0.1%)

Stackprof



Optcarrot::CPU#run
35 (0.2%)
of 388 (1.9%)
Optcarrot::TermInput#tick
265 (1.3%)
161
118
16
34
Optcarrot::CPU#r_op
1 (0.0%)
of 161 (0.8%)
Optcarrot::CPU#_beq
1 (0.0%)
of 118 (0.6%)
Optcarrot::CPU#w_op
0 (0.0%)
of 16 (0.1%)
99
52
117
15
Optcarrot::CPU#zpg
23 (0.1%)
of 99 (0.5%)
Optcarrot::CPU#_cmp
52 (0.3%)
Optcarrot::CPU#branch
46 (0.2%)
of 123 (0.6%)
Optcarrot::CPU#store_mem
0 (0.0%)
of 15 (0.1%)
76
77
15
Optcarrot::CPU#fetch
192 (0.9%)
of 193 (0.9%)
Optcarrot::CPU#store
2 (0.0%)
of 15 (0.1%)

PPU – Source code

```
def render_pixel
 if @any_show
  pixel = @bg_enabled ? @bg_pixels[@hclk % 8] : 0
  if @sp_active && (sprite = @sp_map[@hclk])
   if pixel % 4 == 0
    pixel = sprite[2]
   else
    @sp_zero_hit = true if sprite[1] && @hclk != 255
    pixel = sprite[2] unless sprite[0]
   end
  end
 else
  pixel = @scroll_addr_5_14 & 0x3f00 == 0x3f00 ? @scroll_addr_0_4 : 0
  @bg_pixels[@hclk % 8] = 0
 end
 @output_pixels << @output_color[pixel]
end
```

- Instance variable reads and writes
- Fixnum operations
- Array
  - Access
  - Append

ORACLE

CPU – Source code

```
op_r :_ldx, :imm

def r_op(instr, mode)
  send(mode, true, false)
  send(instr)
end

def imm(_read, _write)
  @data = fetch(@_pc)
  @_pc += 1
  @clk += CLK_2
end

def _ldx
  @_p_nz = @_x = @data
end
```

- Instance variable reads and writes
- Integer operations
- Method calls
- Dynamic method calls
  - #send

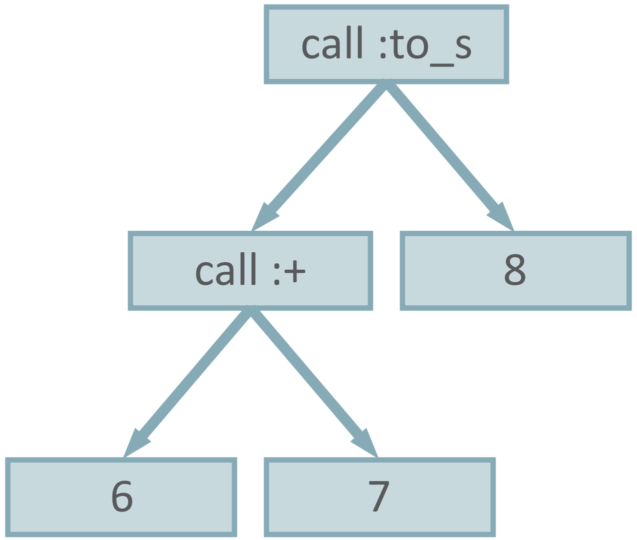
ORACLE

TruffleRuby

How does it work?

AST – Abstract Syntax Tree

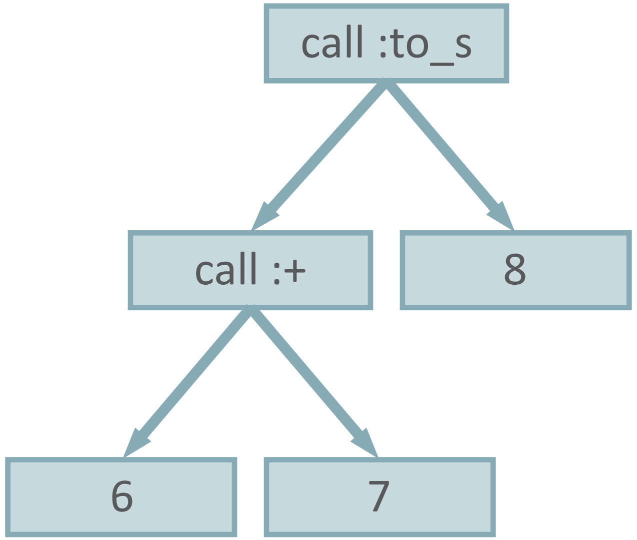
```
def a_method
  (6 + 7).to_s(8)
end
```



AST Interpreter

- Each node is an object with execute method

```
class LiteralNode
  def initialize(value)
    @value = value
  end

  def execute
    @value
  end
end
```



AST Interpreter

```
class MethodCallNode
  def initialize(name, receiver, arguments)
    @name, @receiver, @arguments = name, receiver, arguments
  end

  def execute
    receiver = @receiver.execute
    method = lookup_method receiver, @name
    method.call receiver, *@arguments.map(&:execute)
  end
end
```

- Greatly simplified

Self-optimizing AST Interpreter

```ruby
class UninitialisedMethodCallNode
  def initialize(name, receiver, arguments)
    @name, @receiver, @arguments = name, receiver, arguments; end
  def execute
    method = lookup_method @receiver.execute, @name
    self.replace(CachedMethodCallNode.new method, @receiver, @arguments).execute; end
end

class CachedMethodCallNode
  def initialize(method, receiver, arguments)
    @method, @receiver, @arguments = method, receiver, arguments; end
  def execute
    @method.call @receiver.execute, *@arguments.map(&:execute); end
end
```

- Node replacement
- Monomorphic cache
  - Method lookup is expensive
- Greatly simplified

Partial evaluation

- Eliminates overhead of nodes
- Evaluates constants
  - Final fields
  - Compilation final fields
- Result is a compilation unit for Compiler

ORACLE

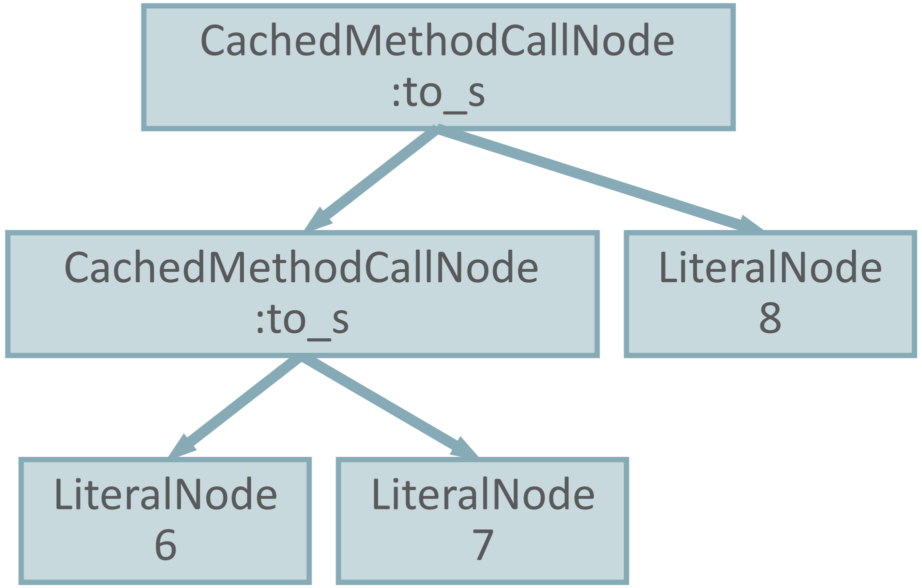
Partial evaluation

```
class CachedMethodCallNode
  attr_final :method, :receiver, :arguments
  def initialize(method, receiver, arguments)
    @method, @receiver, @arguments = method, receiver, arguments; end
  def execute
    @method.call @receiver.execute, *@arguments.map(&:execute); end
end

class LiteralNode
  attr_final :value
  def initialize(value)
    @value = value; end
  def execute
    @value; end
end
```

- Final fields
  - Let's assume Ruby has them for the example

ORACLE

Partial evaluation



.[] represents existing values

```
@method.call @receiver.execute, *@arguments.map(&:execute)
@method.call @receiver.execute, *[LiteralNode[8]].map(&:execute)
@method.call @receiver.execute, LiteralNode[8].execute
@method.call @receiver.execute, 8
@method.call receiver = @receiver.execute, 8

receiver = @receiver.execute
receiver = @method.call @receiver.execute,
                 *@arguments.map(&:execute)
receiver = @method.call @receiver.execute, 7
receiver = @method.call LiteralNode[6].execute, 7
receiver = @method.call 6, 7
receiver = direct_call(Method[Integer, :+], 6, 7)

@method.call receiver = direct_call(Method[Integer, :+], 6, 7), 8
@method.call direct_call(Method[Integer, :+], 6, 7), 8
direct_call(Method[Integer, :to_s],
        direct_call(Method[Integer, :+], 6, 7),
        8)
```

Compilation

```
direct_call(
        Method[Integer, :to_s],
        direct_call(Method[Integer, :+], 6, 7),
        8)
```

- Compiled by Graal compiler
- Next time the compiled code is called instead of interpreting the AST
  - No longer calls the 5 execute methods on the nodes
  - Any overhead of the self-optimizing AST interpreter is eliminated

TruffleRuby example

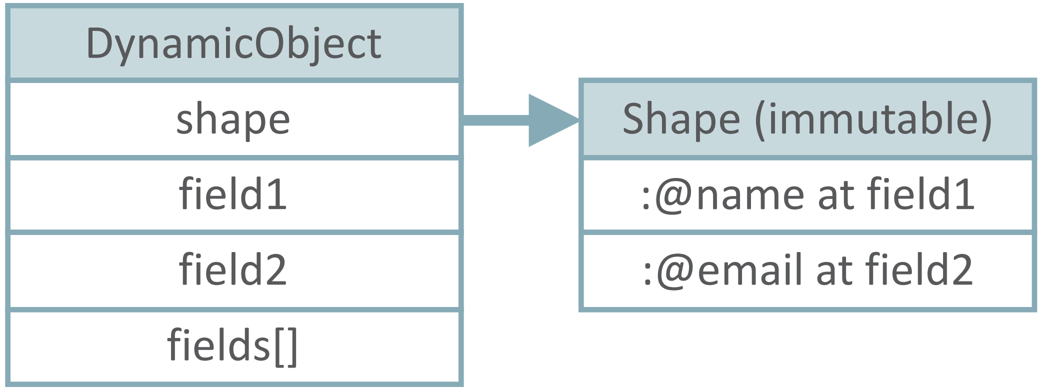
```
@CoreMethod(names = "+", required = 1)
public abstract static class AddNode extends CoreMethodArrayArgumentsNode {
  public abstract Object executeBuiltin(VirtualFrame frame, Object... arguments);

  @Specialization(rewriteOn = ArithmeticException.class)
  public int add(int a, int b) { return Math.addExact(a, b); }

  @Specialization
  public long addWithOverflow(int a, int b) { return (long) a + (long) b; }

  @Specialization(rewriteOn = ArithmeticException.class)
  public long add(long a, long b) { return Math.addExact(a, b); }

  @Specialization
  public Object addWithOverflow(long a, long b) {
    return fixnumOrBignum(BigInteger.valueOf(a).add(BigInteger.valueOf(b)));
  }
  // ...
```

- Fixnum#+
- DSL
- Annotation processor
  - Generates node for each specialization
  - Creates polymorphic chain for more specializations
- Type specialization

Instance variable access

Instance variable access

- Ruby objects are DynamicObjects
  - Or a primitive type like int, long, float if possible
- Values of instance variables are stored in DynamicObject
- Shape defines mapping between instance variable names and fields



Instance variable access – Implementation

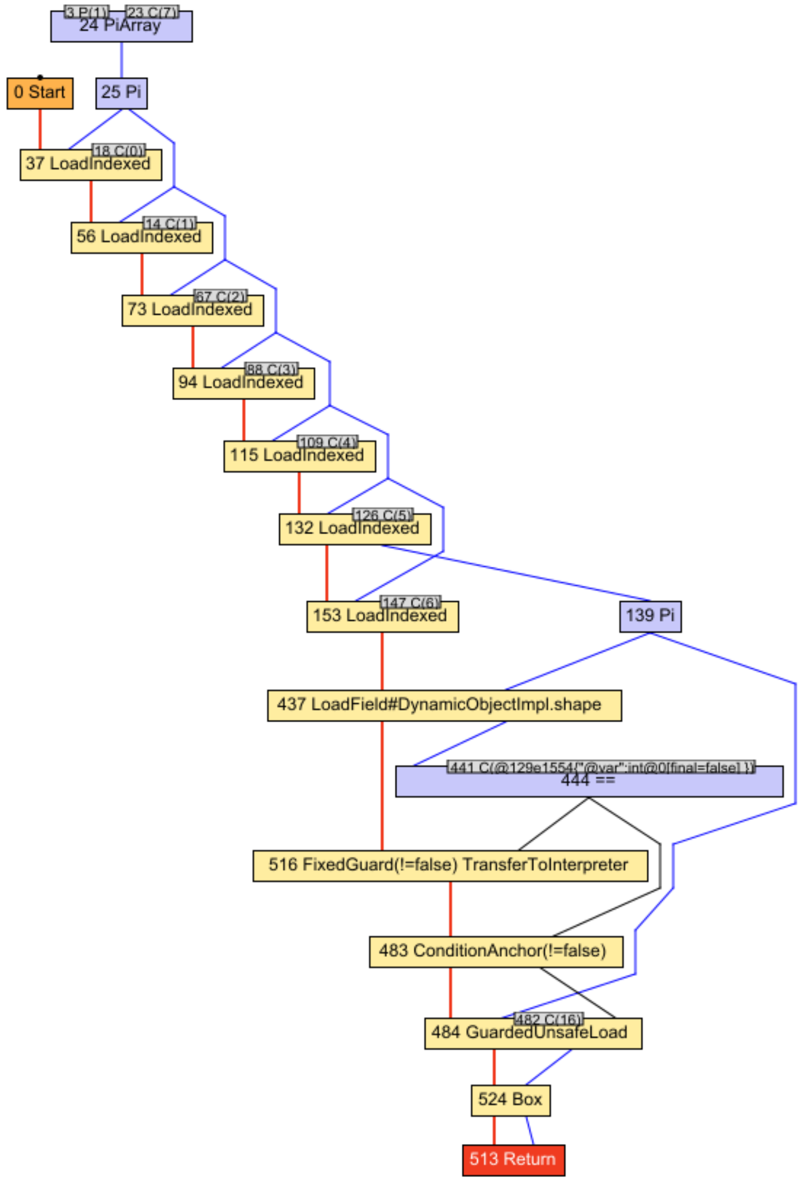
```
public abstract class ReadObjectFieldNode extends RubyBaseNode {
  private final Object defaultValue;
  protected final Object name;

  public ReadObjectFieldNode(Object name, Object defaultValue) {
    this.name = name;
    this.defaultValue = defaultValue;
  }
  @Specialization(guards = "receiver.getShape() == cachedShape", limit = "getCacheLimit()")
  protected Object readObjectFieldCached(DynamicObject receiver,
          @Cached("receiver.getShape()") Shape cachedShape,
          @Cached("cachedShape.getProperty(name)") Property cachedProperty) {
    if (cachedProperty != null) {
      return cachedProperty.get(receiver, cachedShape);
    } else {
      return defaultValue;
    }
  }
```

ORACLE

Instance variable read – IGV

def ***read;*** **@var; end**

- After specialization it only:
  - Reads arguments
    - Including self object
  - Reads the shape of self
  - Ensures it's a correct shape against a constant
  - Read the value of instance variable from a constant offset in self object



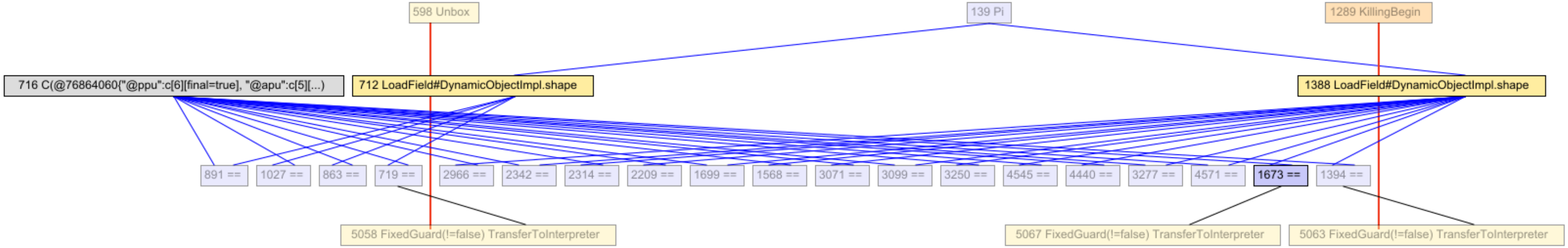
Instance variable access – IGV

```
# zero-page addressing
def zpg(read, write)
 @addr = fetch(@_pc)
 @_pc += 1
 @clk += CLK_3
 if read
  @data = @ram[@addr]
  @clk += CLK_2 if write
 end
end
```

- Many accesses to instance variables
- Checks are merged
  - Only access to a constant offset remains

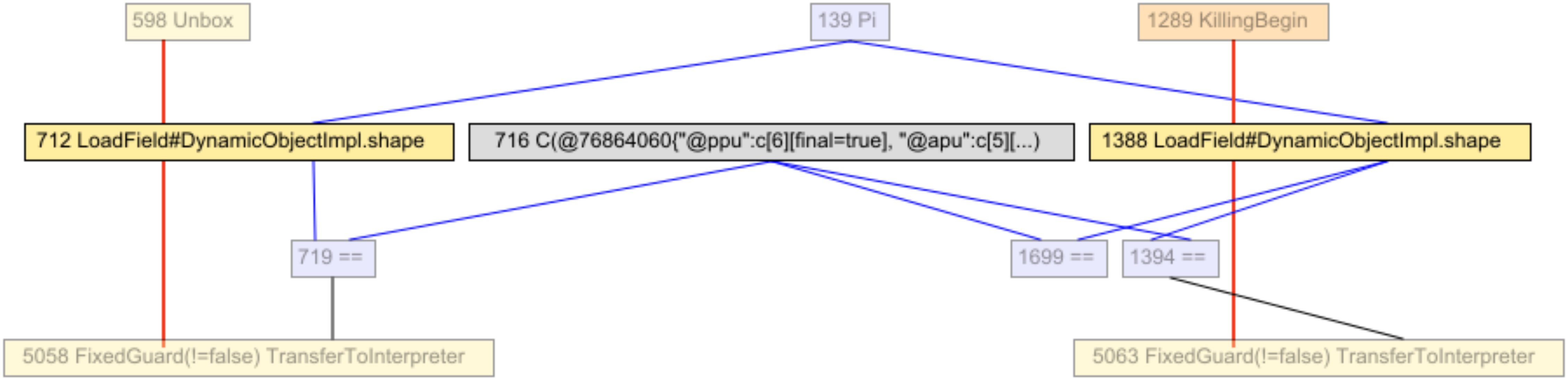
ORACLE

Instance variable access – IGV



ORACLE

Instance variable access – IGV



Splitting

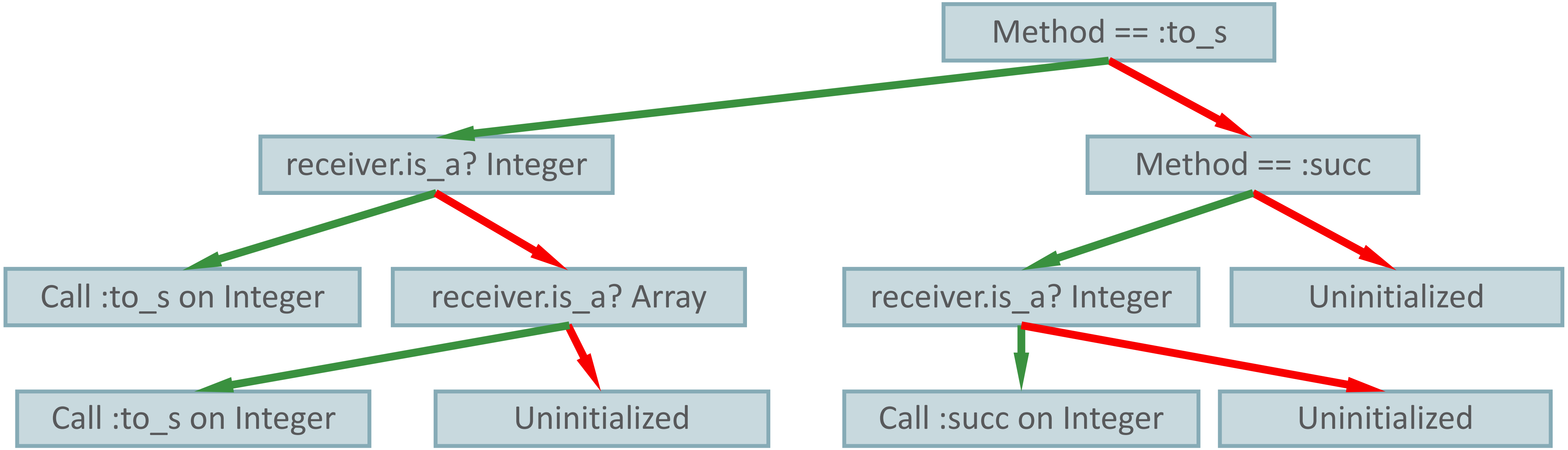
CPU – Source code

```
loop { send(*DISPATCH[@opcode]) }
# send :op_r, :_ldx, :imm

def r_op(instr, mode)
  send(mode, true, false)
  send(instr)
end

def imm(_read, _write)
  @data = fetch(@_pc)
  @_pc += 1
  @clk += CLK_2
end

def _ldx
  @_p_nz = @_x = @data
end
```

- Send – dynamic method call
- Optimized with 2-dimensional polymorphic inline cache
  - Caches already called methods by name and receiver

ORACLE

Two dimensional polymorphic inline cache



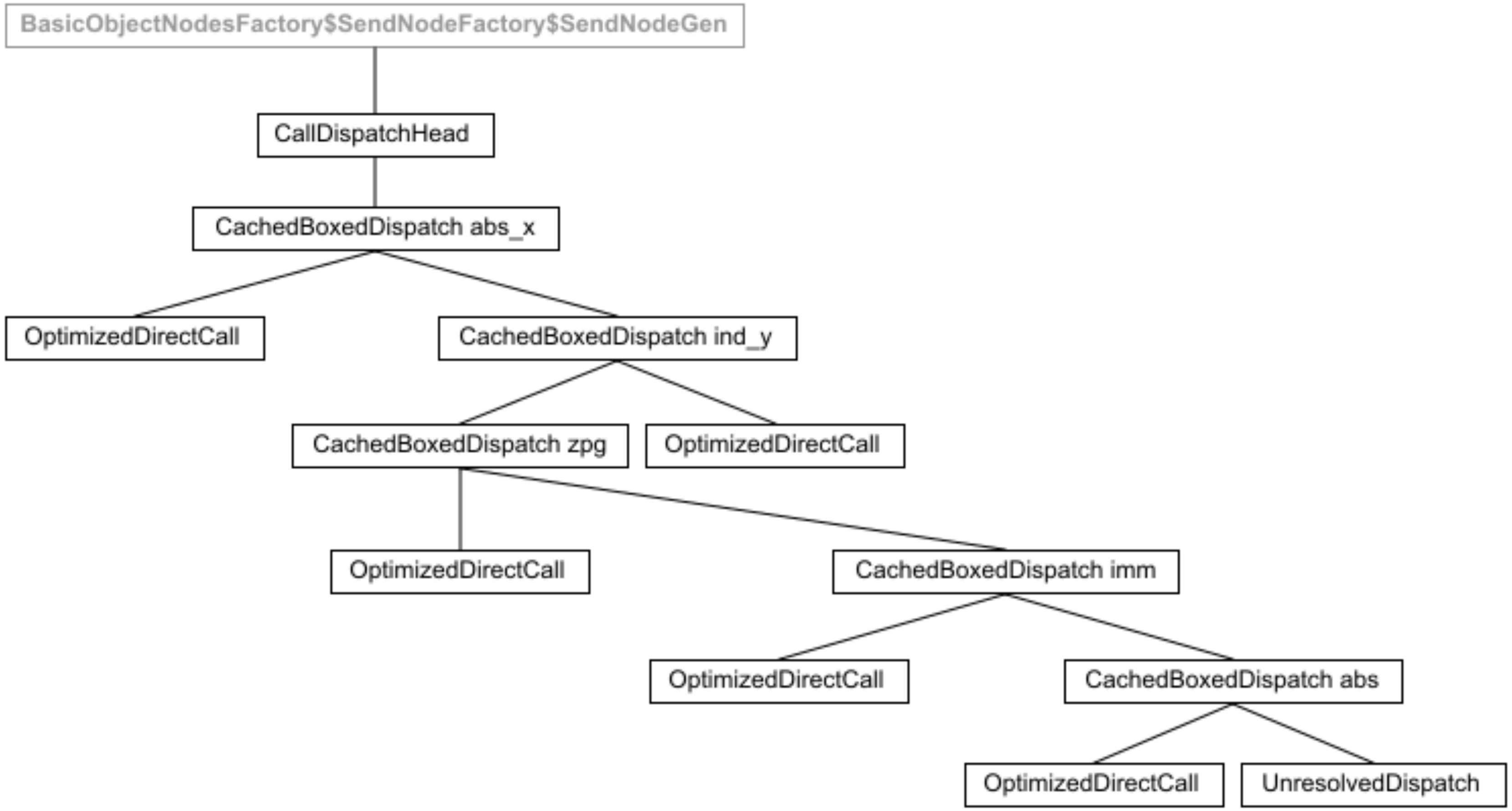
Why split?

```
def r_op(instr, mode)
  send(mode, true, false)
  send(instr)
end

def imm(_read, _write)
  @data = fetch(@_pc)
  @_pc += 1
  @clk += CLK_2
end

def _ldx
  @_p_nz = @_x = @data
end
```

- There are 2 sends in r_op
- Each has to have its own cache to specialize effectively
  - Modes: #abs, #imm, #zpg, #ind_y, #abs_x
  - Instructions: #_ida, #_and, #_cmp, #_ldx, #_ldy, #_adc, #_ora
- Only finds the method in the cache
  - Avoids expensive method lookup in Ruby modules/classes

ORACLE

The cache representation in the IGV



ORACLE

Splitting – summary

- Applied to all methods
- Important for core Ruby methods
  - #each, #step, #==, #to_s
- Avoids megamorphic nodes
- Truffle framework does it automatically

ORACLE

Inlining

Why inlining?

- Splitting ensures methods are specialized in their calling context
- They are optimized independently
  - Same guards cannot be merged across methods
  - Same code cannot be eliminated across methods
- Compiler cannot see into called methods
  - Cannot move things above or bellow method invocations

Why inlining?

- Methods and blocks are inlined into their callers
  - The already specialized and split AST replaces the direct method call
- Creates big chunk of code – compilation unit
  - Can be analyzed and optimized by compiler together
  - More optimizations can be applied
    - Guards merged
    - Repeated and dead code eliminated
- Truffle framework does it automatically

Examples

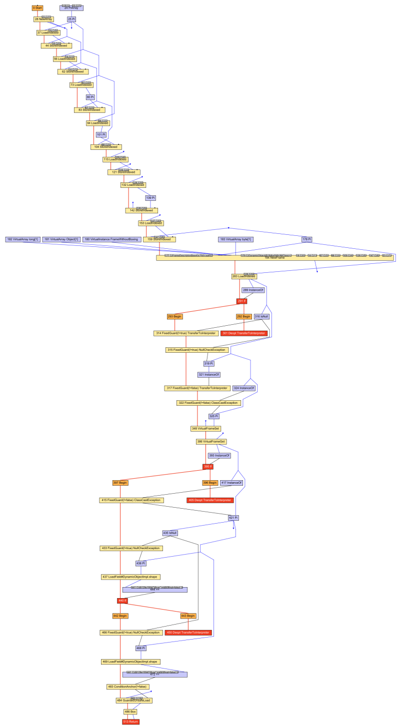
- A Fixnum #+ method is turned into just 2 instructions %addi, %jo after inlining in the calling method
- Ruby blocks
  - Is a very important abstraction in Ruby
  - Are used abundantly
  - Can be fully inlined in TruffleRuby

Block inlining

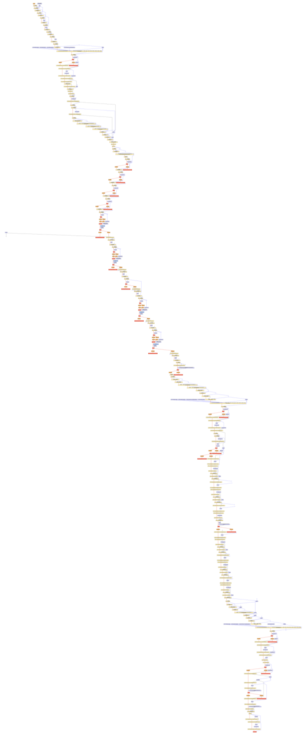
```ruby
def read
  @var
end
```

```ruby
def block
  -> { @var }.call
end
```

Block inlining

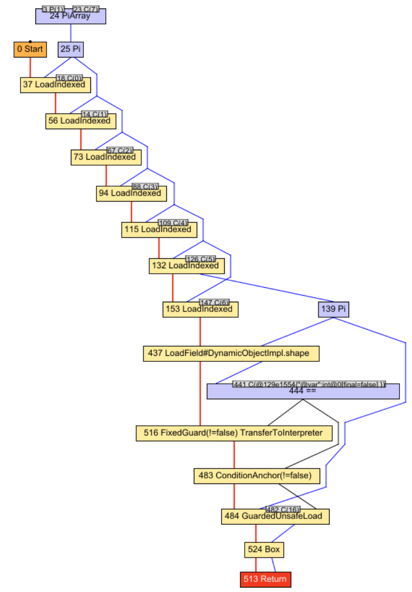


#read method before optimizations

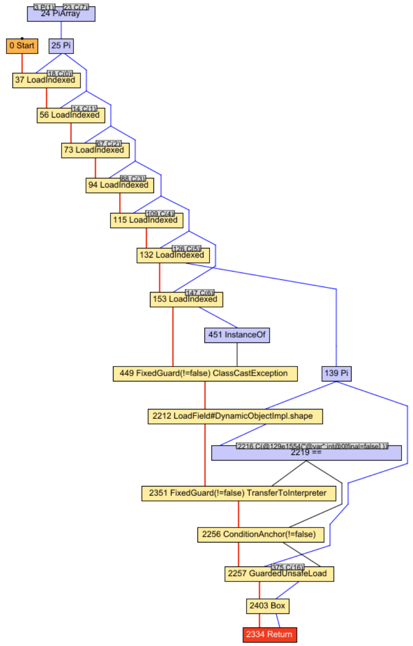


#block method before optimizations

Block inlining



#read method after optimizations



#block method after optimizations

Conclusion

What makes TruffleRuby run Optcarrot 9 times faster than MRI?

- Combination of several optimizations
  - Quick instance variable access
  - Truffle
    - Splitting
    - Inlining
    - Partial evaluation
  - Graal
    - High quality compiler

ORACLE

Acknowledgements

Benoit Daloze
Brandon Fish
Petr Chalupa
Duncan MacGregor
Kevin Menard
Chris Seaton
Jruby & Rubinius Contributors

Oracle
Danilo Ansaloni
Stefan Anzinger
Cosmin Basca
Daniele Bonetta
Matthias Brantner
Petr Chalupa
Jürgen Christ
Laurent Daynès
Gilles Duboscq
Martin Entlicher
Bastian Hossbach
Christian Humer
Mick Jordan
Vojin Jovanovic

Oracle (continued)
Peter Kessler
David Leopoldseder
Kevin Menard
Jakub Podlešák
Aleksandar Prokopec
Tom Rodriguez
Roland Schatz
Chris Seaton
Doug Simon
Štěpán Šindelář
Zbyněk Šlajchrt
Lukas Stadler
Codrut Stancu
Jan Štola
Jaroslav Tulach
Michael Van De Vanter
Adam Welc
Christian Wimmer
Christian Wirth
Paul Wögerer
Mario Wolczko
Andreas Wöß
Thomas Würthinger

Oracle Interns
Brian Belleville
Miguel Garcia
Shams Imam
Alexey Karyakin
Stephen Kell
Andreas Kunft
Volker Lanting
Gero Leinemann
Julian Lettner
Joe Nash
David Piorkowski
Gregor Richards
Robert Seilbeck
Rifat Shariyar

Alumni
Erik Eckstein
Michael Haupt
Christos Kotselidis
Hyunjin Lee
David Leibs
Chris Thalinger
Till Westmann

JKU Linz
Prof. Hanspeter Mössenböck
Benoit Daloze
Josef Eisl
Thomas Feichtinger
Matthias Grimmer
Christian Häubl
Josef Haider
Christian Huber
Stefan Marr
Manuel Rigger
Stefan Rumzucker
Bernhard Urban

University of Edinburgh
Christophe Dubach
Juan José Fumero Alfonso
Ranjeet Singh
Toomas Remmelg

LaBRI
Floréal Morandat

University of California, Irvine
Prof. Michael Franz
Gulfem Savrun Yeniceri
Wei Zhang

Purdue University
Prof. Jan Vitek
Tomas Kalibera
Petr Maj
Lei Zhao

T. U. Dortmund
Prof. Peter Marwedel
Helena Kotthaus
Ingo Korb

University of California, Davis
Prof. Duncan Temple Lang
Nicholas Ulle

University of Lugano, Switzerland
Prof. Walter Binder
Sun Haiyang
Yudi Zheng

ORACLE

Safe Harbor Statement

The preceding is intended to provide some insight into a line of research in Oracle Labs. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. Oracle reserves the right to alter its development plans and practices at any time, and the development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.

Integrated Cloud

Applications & Platform Services

ORACLE®