Who reordered my code?!

Petr Chalupa
Principal Member of Technical Staff
Oracle Labs
September 08, 2016

JRuby+Truffle
Concurrent Ruby

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.

Outline

1. When you can see reordering?
2. What does it do?
3. Embrace or reject?
4. How to deal with reordering?
5. Does it have a practical use?

Ruby's new goals

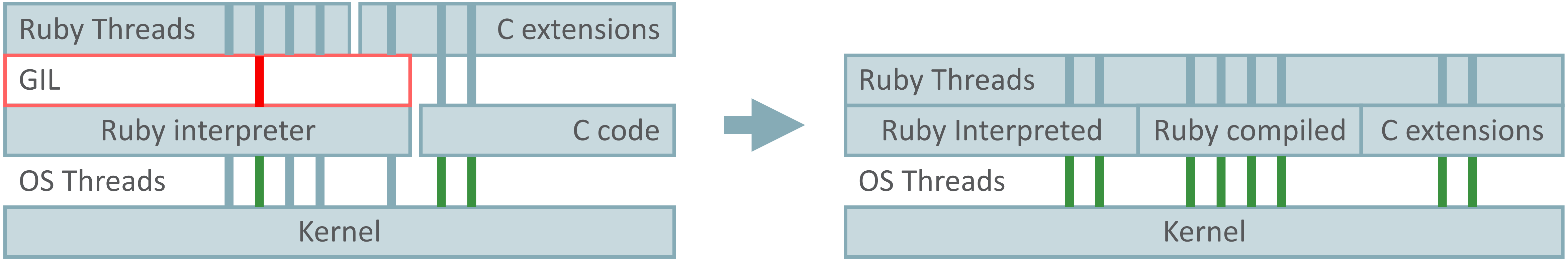
Performance

- CRuby 3x3 (Heroku, Appfolio)
- Ruby OMR preview – OMR, J9 (IBM)
- JRuby – invokedynamic, new IR (Red Hat)
- JRuby+Truffle – Truffle, Graal (Oracle)

ORACLE

Parallelism

- Almost every computer has more than one core
- Parallel computation has to be supported to utilize all cores
- JRuby and JRuby+Truffle support parallel execution
- Maybe GIL will be removed in Ruby 3?



Concurrent library

- Ideas considered for Ruby 3: actors, isolation, channels, streams, …
  - Easy to use high-level concurrency abstraction
- Unanswered questions:
  - How to write fast concurrent data-structures?
  - How to write more concurrent abstractions?

ORACLE

Reordering

When we can see it?

- Fast Ruby implementation
- Parallel execution

Fast Ruby implementation

For Ruby language to be fast an implementation with **speculatively optimizing dynamic compilation** and **parallel** execution is needed.

- **Speculative**: can speculate on following propositions
  - Method body is invariable
  - Constant's value is invariable
  - Type speculation
  - …

```ruby
def foo(a, b)
  COUNT * (a + b)
end

foo(1, 2)
```

Fast Ruby implementation

For Ruby language to be fast an implementation with **speculatively optimizing dynamic compilation** and **parallel** execution is needed.

- **Optimizing**: does all the clever optimizations as e.g. gcc
  - In-lining
  - Splitting
  - Constant folding
  - Value numbering
  - Hoisting
  - …

ORACLE

Fast Ruby implementation

For Ruby language to be fast an implementation with **speculatively optimizing dynamic compilation** and **parallel** execution is needed.

- **Dynamic:**
  - Just-in-time compilation of hot methods
  - Also deoptimize when speculatively taken assumptions fail
- **Parallel:**
  - Ruby code runs in parallel

ORACLE

Fast Ruby implementation

- JRuby+Truffle is such an implementation
  - **Truffle:** self optimizing AST interpreter
  - **Graal:** compiler written in Java

ORACLE

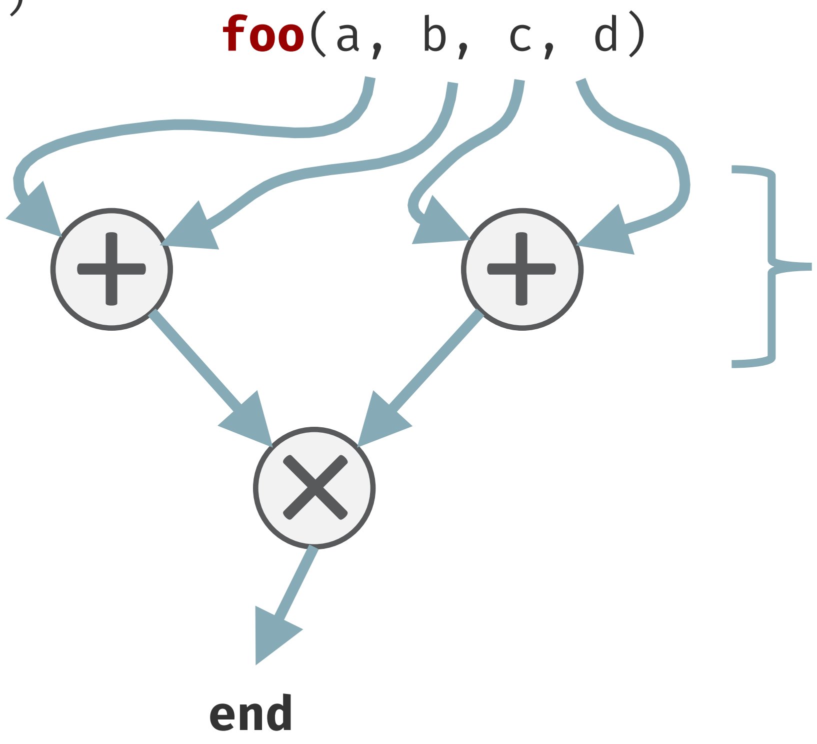
Sources of reordering

Compiler reorders code

- Optimizes by transforming the code
- Is allowed to do for us any optimization if the transformation cannot be observed on the same thread
  - The code has the same result
  - Assumes only one thread

Seemingly sequential Ruby code

```
def foo(a, b, c, d)
  x = a + b
  y = c + d
  x + y
end
```



These two operations can happen in either order

Why? Because they are independent operations – there are no dependencies between the two.

Expanded to a parallel graph in the compiler

Seemingly sequential Ruby code

(pseudo assembly)

```
add a b %r1
add c d %r2
mul %r1 %r2 %r3
ret %r3
```

```
add c d %r1
add a b %r2
mul %r1 %r2 %r3
ret %r3
```

Generated machine code can use either order of operations

Why? Because they are independent operations – there are no dependencies between the two.

ORACLE

Seemingly sequential Ruby code

```
add a b %r1        add c d %r2

      mul %r1 %r2 %r3

      ret %r3
```

These two operations can happen in your processor in either order

Why? Because they are independent instructions – there are no dependencies between the two.

Even if our compiler didn't reorder, the processor could do it anyway!

ORACLE

Example

```
class Future
  def initialize; @value = nil; end
  def fulfill(v); end
  def value; end
end
```

Thread 2

```
def value
  Thread.pass until @value
  @value
end
```

Transformed into

```
def value
  value = @value
  Thread.pass until value
  @value
end
```

Thread 1

```
def fulfill(result)
  @value = result
end
```

Order

```
2: value = @value   # nil
2: Thread.pass until value # nil
1: @value = result # :result
```

If `value` *is called before* `fulfill` *it will* ***block*** *indefinitely.*

Cache reordering effects

- Dekker's algorithm
- Compiler without reordering
- Old processor executing in program order
  - No out-of-order execution
- Coherent cache with just a write buffer

Cache reordering effects

```
flag1 = flag2 = false
```

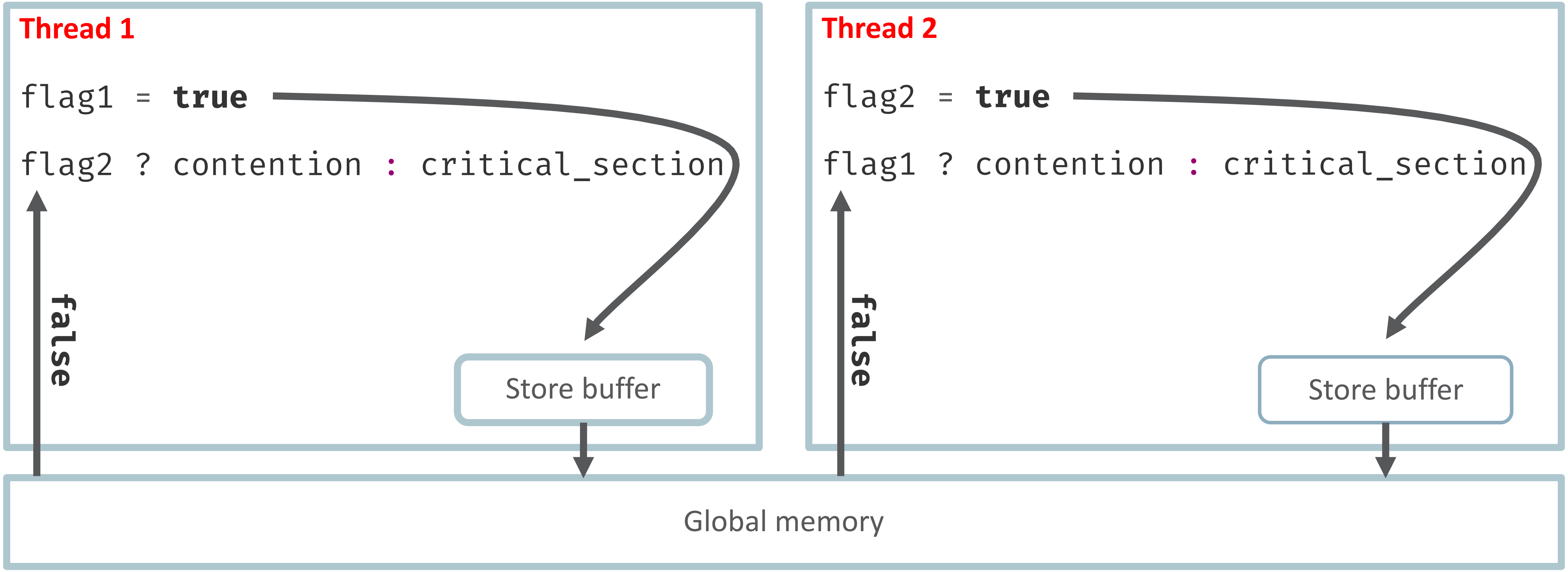
Thread 1

```
flag1 = true
flag2 ? contention : critical_section
```

Thread 2

```
flag2 = true
flag1 ? contention : critical_section
```

Cache reordering effects



Processor reordering effects

- Decker's algorithm
- Compiler without reordering
- Out-of-order processor
- No cache

Processor reordering effects

```
flag1 = flag2 = false
```

Thread 1

```
flag1 = true
flag2 ? contention : critical_section
```

Thread 2

```
flag2 = true
flag1 ? contention : critical_section
```

Processor reordering effects

```
flag1 = flag2 = false
```

Thread 1

```
r1    = flag2 # read
flag1 = true  # write
r1 ? contention : critical_section
```

Thread 2

```
r1    = flag1 # read
flag2 = true  # write
r1 ? contention : critical_section
```

- Store reordered with load
- StoreLoad reordering is allowed on x86

ORACLE

Live example

- Decker's algorithm on JRuby+Truffle
  - Without compiler
  - With Graal enabled

Who reordered my code?!

- It might have been:
  - Compiler
  - Cache
  - Processor
- We do not care who it was though, only the actual execution matters
- The reordered code runs faster while the transformation cannot be observed on a single thread

ORACLE

Do we want reordering?

- **Yes**
  - Even the very basic code transformations would be forbidden without it
  - It would require memory barriers around every read and write
- We want to let the compiler, cache, processor
  - keep working for us,
  - run our code **faster** then we wrote it,
  - minimize waiting for memory

ORACLE

Relaxed memory order

```ruby
class Variable
  def initialize
    @mutex, @updates, @seen_up_to = Mutex.new, [], {}
  end

  def write(value)
    @mutex.synchronize do
      @updates << value
      @seen_up_to[Thread.current] = @updates.size - 1
    end
    value
  end

  def read
    @mutex.synchronize do
      seen = @seen_up_to[Thread.current] || 0
      new_seen = (seen...@updates).to_a.sample # already seen or newer
      @seen_up_to[Thread.current] = new_seen
      return @updates[new_seen]
    end
  end
end
```

Updates	Seen by
-	Thread 1
0	
1	Thread 2, Thread 3
42	Thread 4
54	

ORACLE

Relaxed memory order

- Each thread sees different values
- Variables are completely independent
- Only the order of the values is shared
- Not every value has to be seen by a given thread
- No way to tell if a thread got the latest value
- Corresponds to relaxed order of atomics variables in C++

Taming reordering

Sequential consistency

“The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.” — Leslie Lamport 1979

- Allows to reason about the program as if it is executed interleaved on one thread even though it's executed in parallel on many threads
- Cannot be done for all variables
- Better to apply to just shared variables

ORACLE

Sequential consistency

Thread 1

```
line :a
line :b
```

Thread 2

```
line 1
line 2
```

Allowed orders

```
line :a         line :a         line :a             line 1          line 1          line 1
line :b            line 1          line 1           line 2      line :a         line :a
   line 1          line 2       line :b         line :a            line 2       line :b
   line 2       line :b            line 2       line :b         line :b            line 2
```


Sequential consistency

Can `:a` and `:b` be both printed?

```
a = b = false
```

Thread 1

```
a = true
```

Thread 2

```
b = true
```

Thread 3

```
if a && !b
  puts :a
end
```

Thread 4

```
if a && !b
  puts :a
end
```

Assuming `a && !b` the order has to be

```
a = true
a && !b # => true
# puts :a
b = true
# puts :a
```

- Impossible to insert `b && !a` to a place where it would be true
- The reasoning is just mirrored for puts `:b`

Memory model

- It's difficult to define
  - We'll focus only on implications
- Defines shared variables
- Allows optimizations while keeping sequential consistency
- Contract: the program is sequentially consistent if there are no data races
- Answers which values can a particular read return in a program

ORACLE

Shared variables

- Called volatile in Java and atomic in C++
- We have to tell the compiler which variables are shared
  - It has to assume that they may be accessed at any time from other threads
  - Reads and writes of shared variables cannot be reordered
- Reads and writes are atomic
- To conform with sequential consistency, intuitively:
  - Release: When written, it has to be made visible immediately to all other threads
  - Acquire: When read, it reads the latest value
- Provides safe publication
  - Release and acquire has useful effect on non-shared variables

ORACLE

Shared variables

```ruby
a = 0
shared = false
```

Thread 1

```ruby
a = 42 # cannot be moved down
shared = true # release
```

Thread 2

```ruby
if r1 = shared # acquire
  r2 = a # cannot be moved up
end
[r1, r2] # => [true, 42], [false, nil]
```

Possible orders

```ruby
r1 = shared # false
# no `r2 = a`
a = 42
shared = true
```

```ruby
a = 42
r1 = shared # false
# no `r2 = a`
shared = true
```

```ruby
a = 42
shared = true
r1 = shared # true
r2 = a
```

Example – fixed

```
class Future
  shared :@value
  def initialize; @value = nil; end
  def fulfill(v); end
  def value; end
end
```

Thread 1

```
def value
  Thread.pass until @value
  @value
end
```

Transformed into

```
def value
  value = @value
  Thread.pass until @value
  @value
end
```

Thread 2

```
def fulfill(value)
  @value = value
end
```

`@value` cannot be reordered, has to actually **read** the value each time.

Building with memory model

Counter

- A counter:
  - `.new(value = 0)`
  - `#add(increment = 1)`
  - `#value`
- Staring by using what is currently available Mutex

Counter

```ruby
class MutexCounter
  def initialize(value = 0)
    @mutex = Mutex.new
    @mutex.synchronize { @value = value }
  end

  def add(increment = 1)
    @mutex.synchronize do
      @value += increment
    end
  end

  def value
    @mutex.synchronize { @value }
  end
end
```

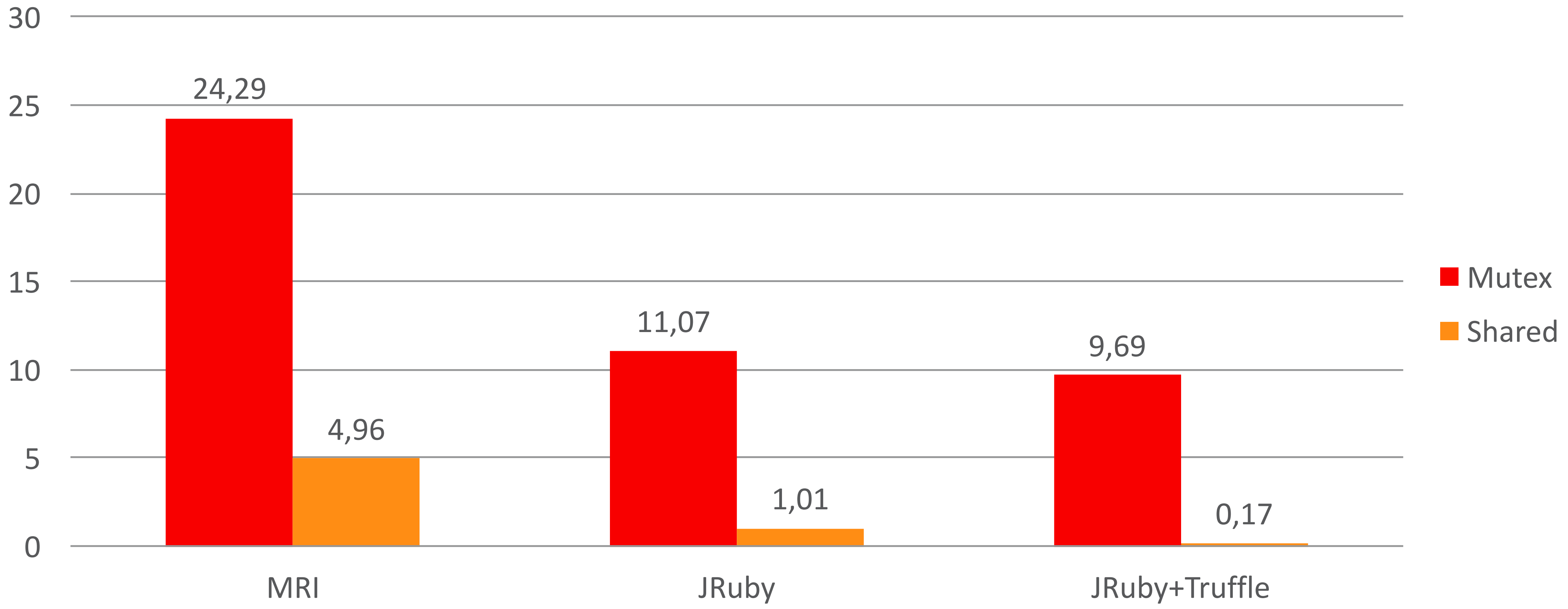
Counter

```ruby
class SharedCounter
  def initialize(value = 0)
    @mutex = Mutex.new
    @value = AtomicReference.new value
  end

  def add(increment = 1)
    @mutex.synchronize do
      @value.set @value.get + increment
    end
  end

  def value
    @value.get
  end
end
```

Benchmark – `value` improvement



ORACLE

Compare-and-set operations

- Atomic operation on a shared variable

```
compare_and_set expected, new_value # => true || false

attr_atomic :value # shared variable
self.value = 1
```

Thread 1

```
while true
  current = value
  new_value = current + 1
  break if compare_and_set_value(
      current, new_value)
end
```

Thread 2

```
while true
  current = value
  new_value = current * 2
  break if compare_and_set_value(
      current, new_value)
end
```

ORACLE

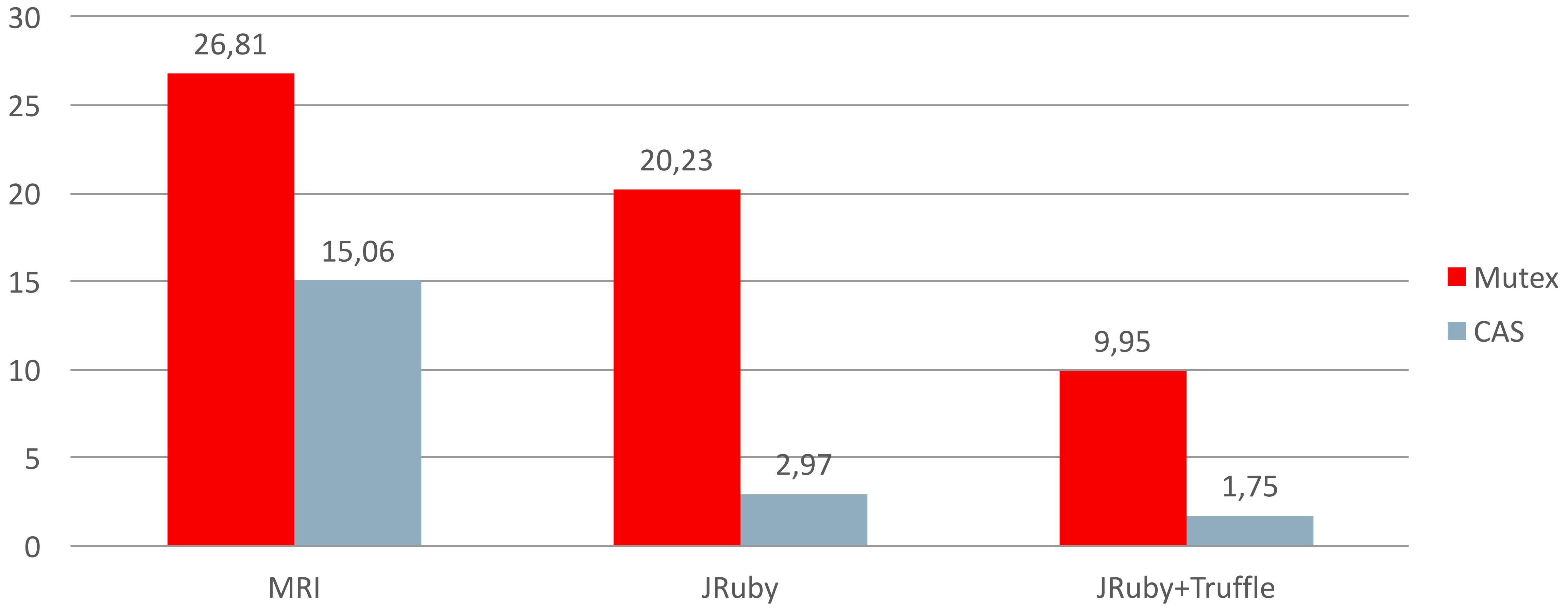
Counter

```ruby
class CasCounter
  def initialize(value = 0)
    @value = AtomicReference.new value
  end

  def add(increment = 1)
    while true
      current = @value.get
      new_value = current + increment
      break if @value.compare_and_set(current, new_value)
    end
  end

  def value
    @value.get
  end
end
```

Benchmark – add improvement



Conclusions

- Fast Ruby implementation
- Parallel execution
- Shared memory

→ Reordering + Memory model

- Shared variables
- Sequential consistency

Fast concurrent data structures and concurrency abstractions built directly in Ruby

It is not for every day coding. Look for abstractions in gems like concurrent-ruby first.

ORACLE

Acknowledgements

Oracle
Danilo Ansaloni
Stefan Anzinger
Cosmin Basca
Daniele Bonetta
Matthias Brantner
Petr Chalupa
Jürgen Christ
Laurent Daynès
Gilles Duboscq
Martin Entlicher
Brandon Fish
Bastian Hossbach
Christian Humer
Mick Jordan
Vojin Jovanovic
Peter Kessler
David Leopoldseder
Kevin Menard
Jakub Podlešák
Aleksandar Prokopec
Tom Rodriguez

Oracle (continued)
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