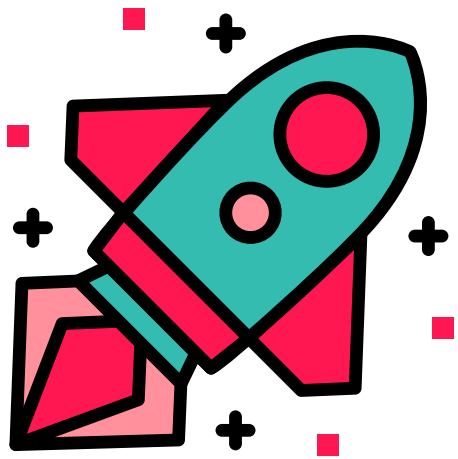
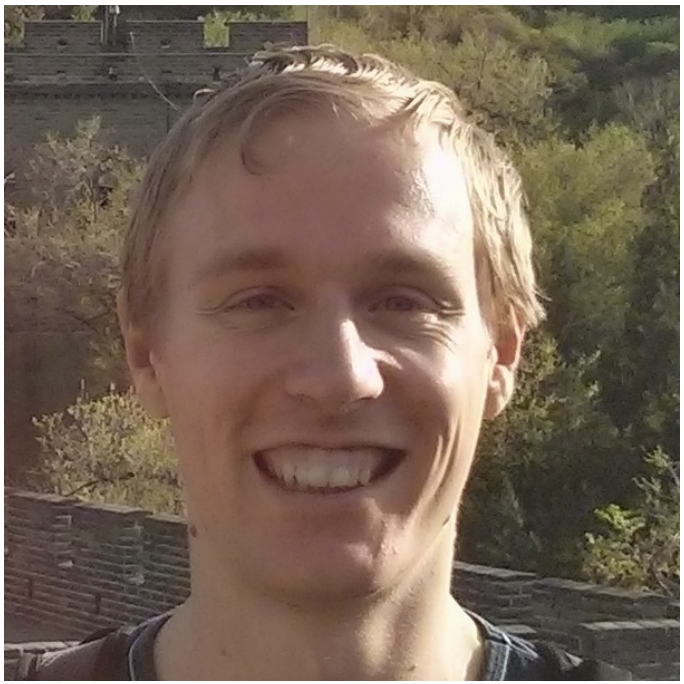
Parallel and Thread-Safe Ruby at High-Speed with TruffleRuby

Benoit Daloze
@eregontp



TRUFFLE
RUBY

Who am I?



Benoit Daloze

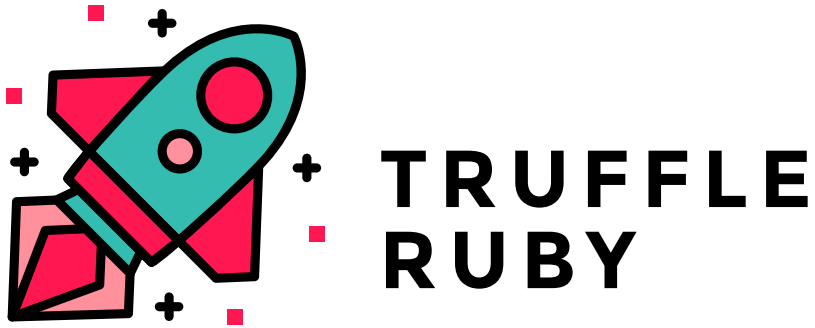
Twitter: @eregontp
GitHub: @eregon

- PhD student at Johannes Kepler University, Austria
- Research on concurrency with TruffleRuby
- Worked on TruffleRuby for 3+ years
- Maintainer of ruby/spec
- CRuby (MRI) committer

Agenda

1. Performance of TruffleRuby
2. Parallel and Thread-Safe Ruby

TruffleRuby



- A high-performance implementation of Ruby by Oracle Labs
- Uses the Graal Just-In-Time Compiler
- Targets full compatibility with CRuby, including C extensions
- `https://github.com/oracle/truffleruby`

Two Modes to Run TruffleRuby

On the Java Virtual Machine (JVM)

- Can interoperate with Java
- Great peak performance

On SubstrateVM, which AOT compiles TruffleRuby & Graal to produces a native executable (default mode)

- Fast startup, even faster than MRI 2.5.1! (25ms vs 44ms)
- Fast warmup (Graal & TruffleRuby interpreter precompiled)
- Lower footprint
- Great peak performance

Ruby 3x3 Project

Goal: CRuby 3.0 should be 3x faster than CRuby 2.0
$\Rightarrow$ with a just-in-time (JIT) compiler: MJIT

Ruby 3x3 Project

Goal: CRuby 3.0 should be 3x faster than CRuby 2.0
$\Rightarrow$ with a just-in-time (JIT) compiler: MJIT

- Do we need to wait for CRuby 3? ($\approx$ 2020)

Ruby 3x3 Project

Goal: CRuby 3.0 should be 3x faster than CRuby 2.0
$\Rightarrow$ with a just-in-time (JIT) compiler: MJIT

- Do we need to wait for CRuby 3? ($\approx$ 2020)
- Can Ruby be faster than 3x CRuby 2.0?

OptCarrot: Demonstration

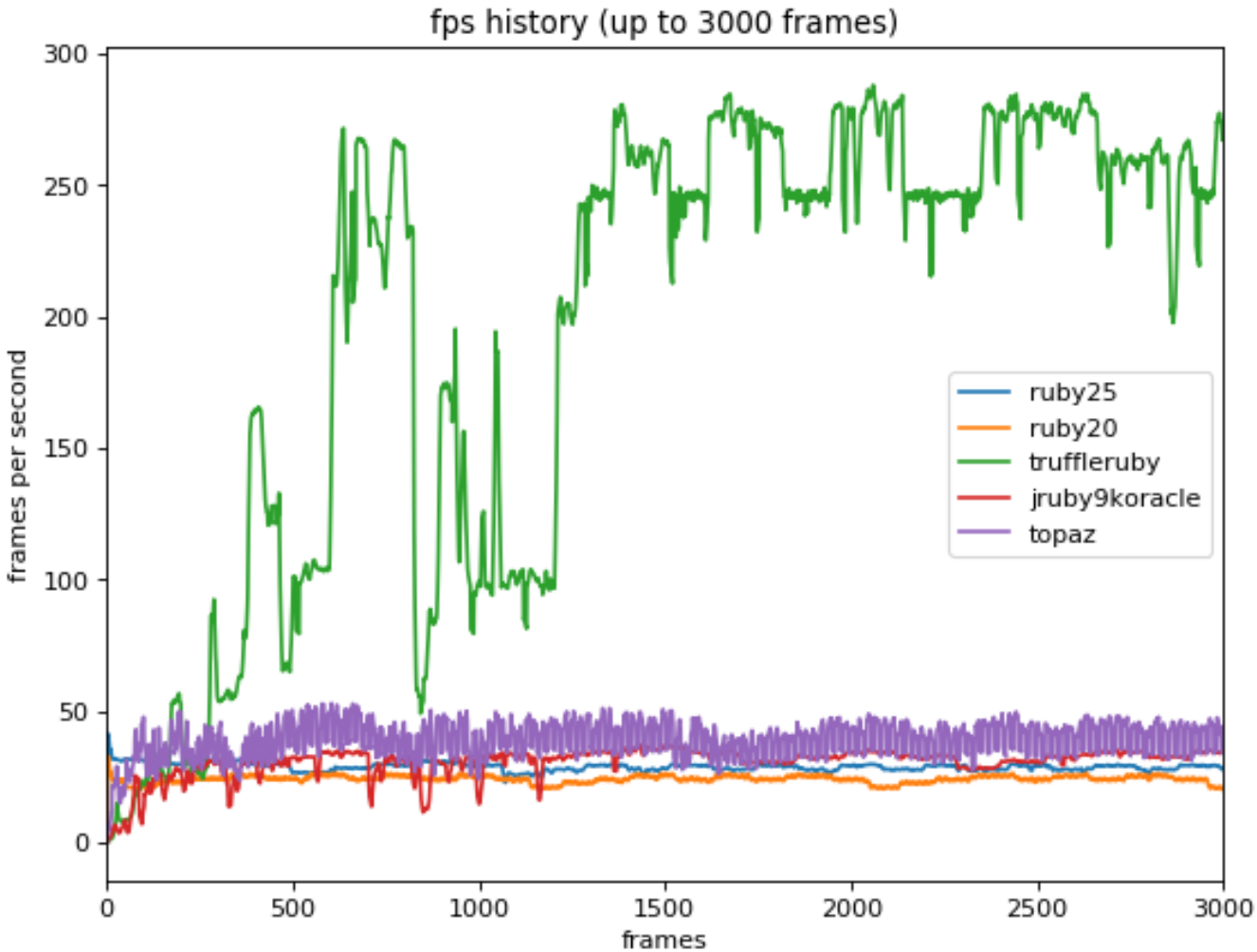
- The main CPU benchmark for Ruby 3x3
- A Nintendo Entertainment System emulator written in Ruby
- Created by @mame (Yusuke Endoh)

OptCarrot: Demonstration

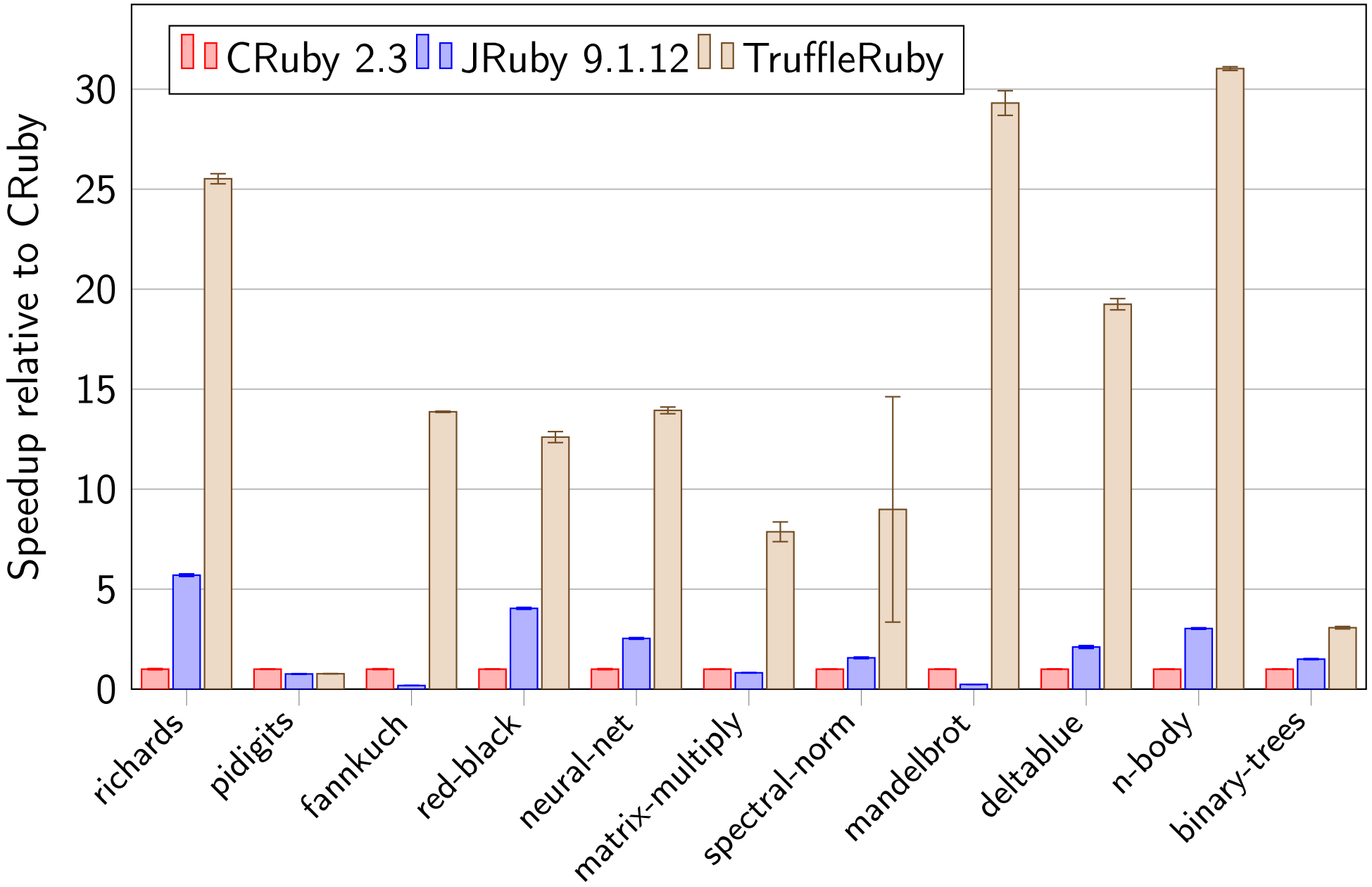
- The main CPU benchmark for Ruby 3x3
- A Nintendo Entertainment System emulator written in Ruby
- Created by @mame (Yusuke Endoh)

Demo!

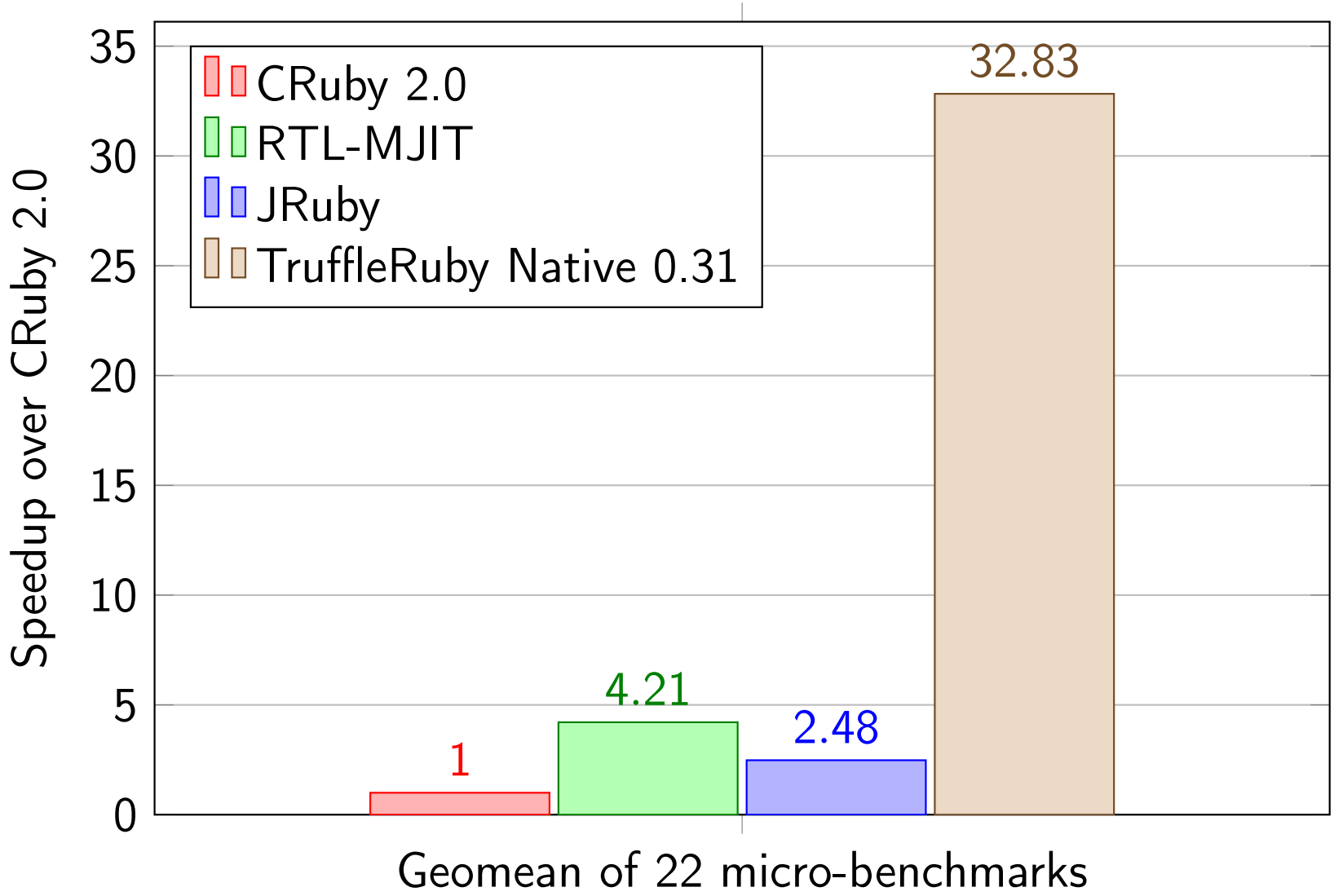
OptCarrot Warmup



Classic Benchmarks



MJIT Micro-Benchmarks



Other areas where TruffleRuby is very fast

9.4x faster than MRI 2.3 on rendering a small ERB template

- Thanks to Strings being represented as Ropes

- See Kevin Menard's talk at RubyKaigi 2016
  "A Tale of Two String Representations"
  `https://www.youtube.com/watch?v=UQnxukip368`

`eval()`, `Kernel#binding`, `Proc` & lambdas & blocks, ...

- But no time to detail in this talk

Rails benchmarks? Discourse?

- We are working on running Discourse
- Database migration and the asset pipeline work with a few patches
- Many dependencies (> 100)
- Many C extensions, which often need patches currently
  - Ongoing research & experiments to reduce needed patches
- We support openssl, psych, zlib, syslog, puma, sqlite3, unf, etc

How TruffleRuby achieves great performance

- Partial Evaluation
- The Graal JIT compiler

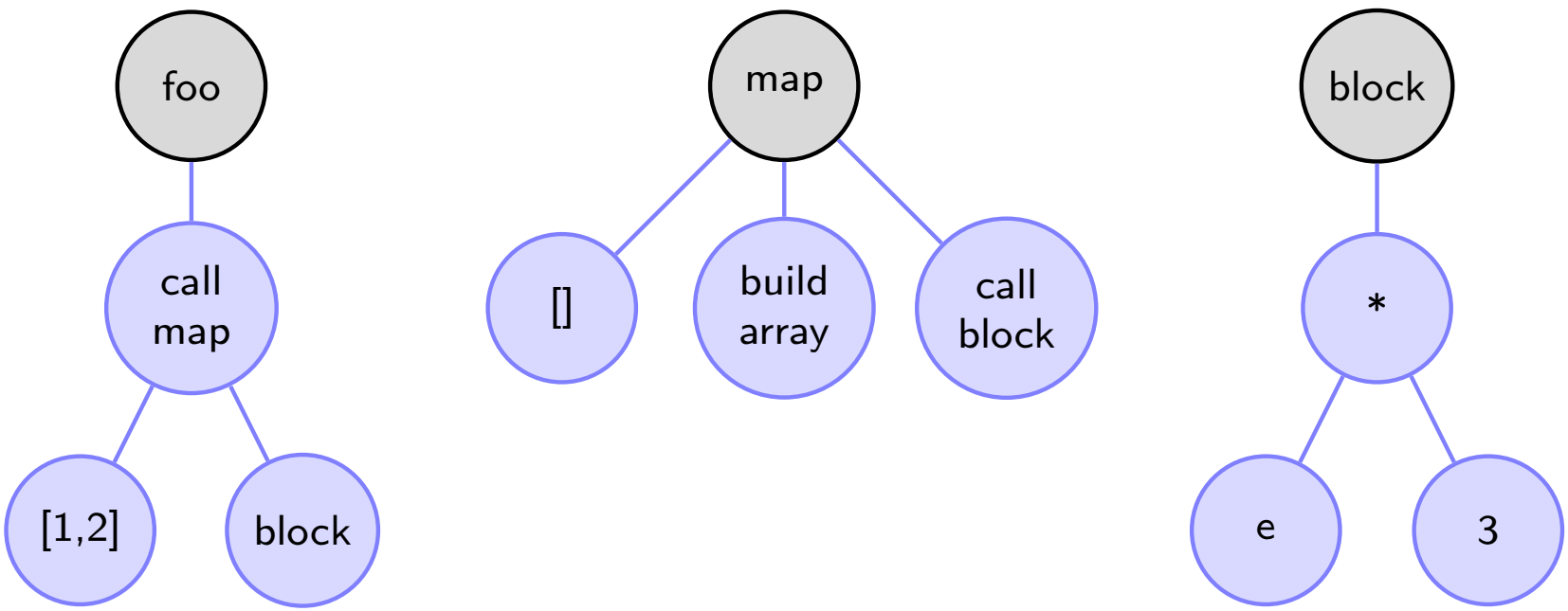
Implementation:

- core primitives (e.g., `Integer#+`) are written in Java
- the rest of the core library is written in Ruby

From Ruby code to Abstract Syntax Tree

```ruby
def foo
  [1, 2].map { |e| e * 3 }
end
```

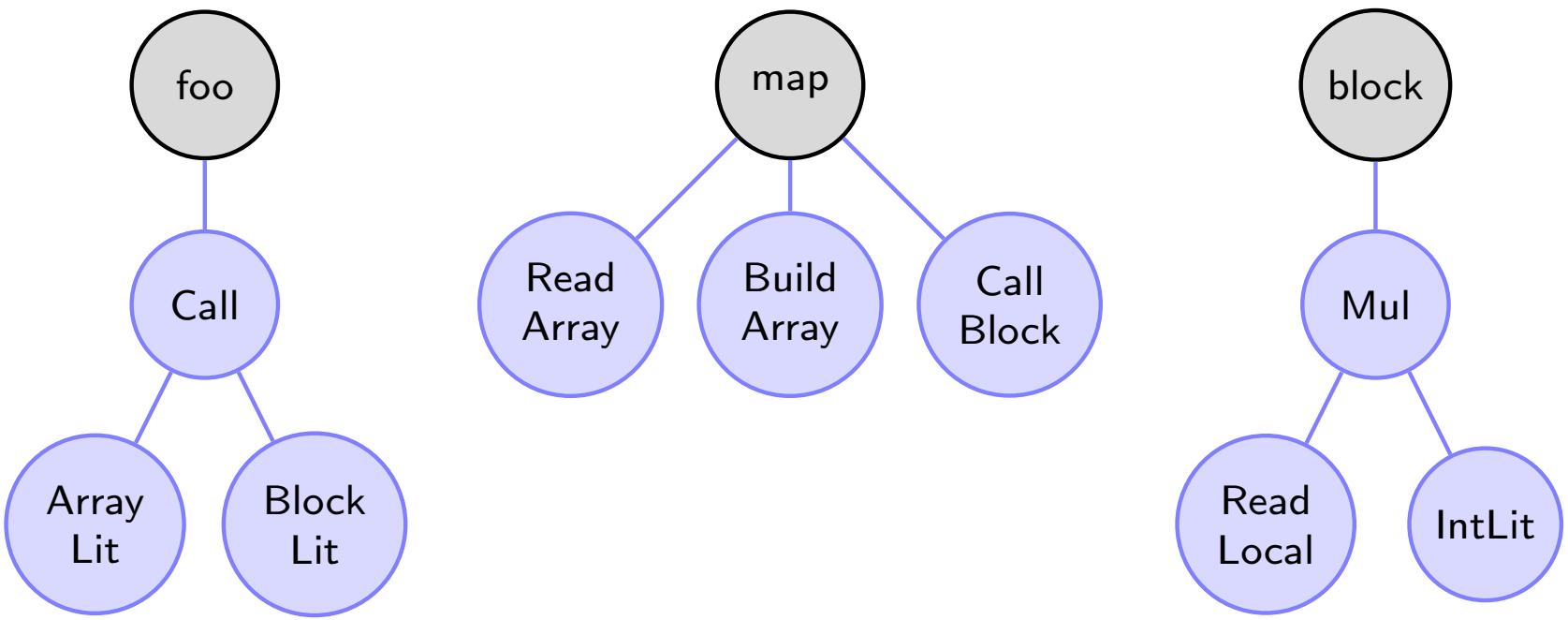
⇓ Parse



Truffle AST: a tree of Node objects with execute()

```
def foo
  [1, 2].map { |e| e * 3 }
end
```

⇓ Parse

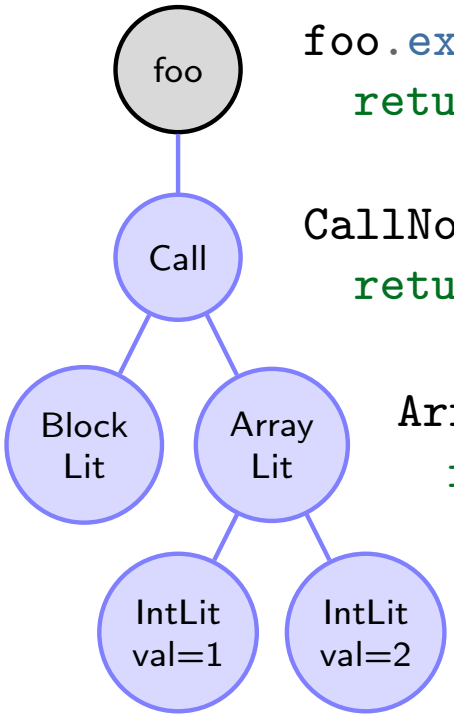


Partial Evaluation

PartialEvaluation(Truffle AST of a Ruby method) =
a CompilerGraph representing how to execute the Ruby method

- Start from the root Node execute() method and inline every Java method while performing constant folding

Partial Evaluation of foo()




```
foo.execute() =
  return child.execute()

CallNode.execute() =
  return callMethod(receiver.execute(), name, args.execute())

ArrayLitNode.execute() =
  return new RubyArray(values.execute())

IntLitNode.execute() =
  return val;
```

⇓ Partial Evaluation

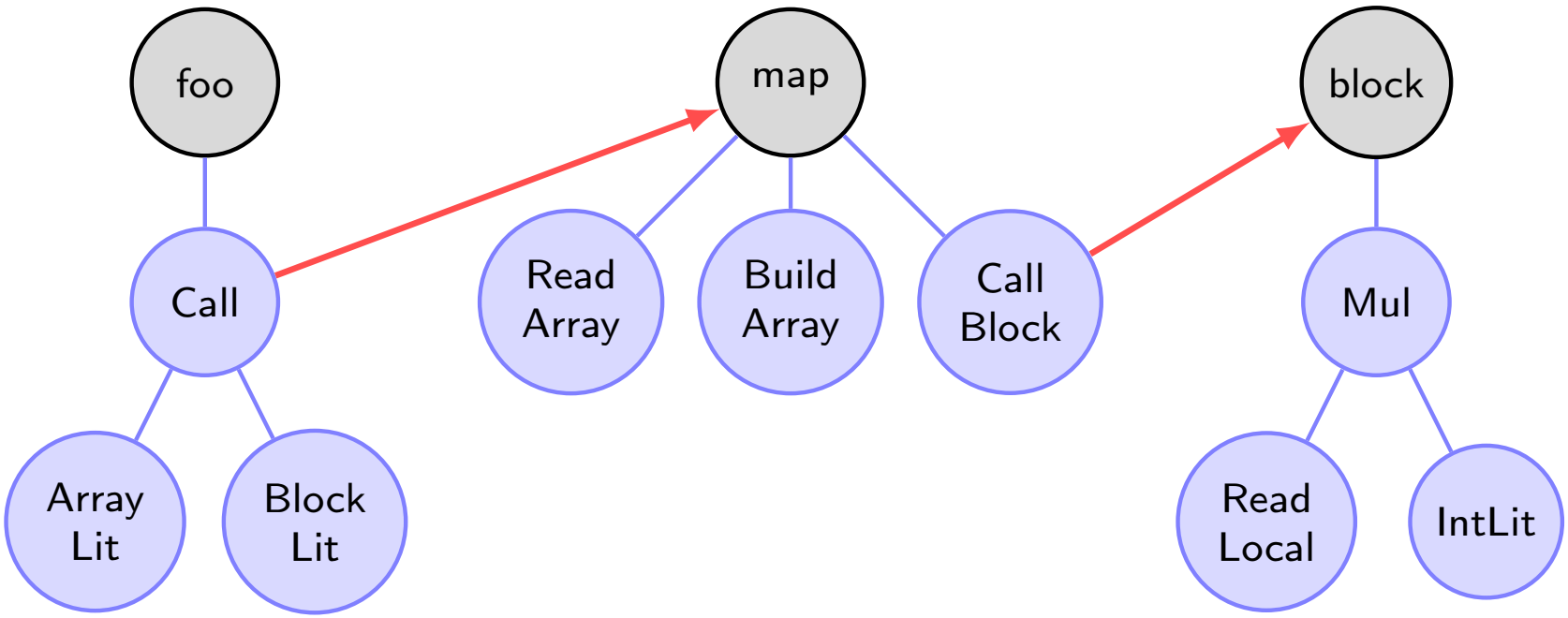
```
Object foo() {
  return callMethod(
    new RubyArray(new int[2] { 1, 2 }),
    "map",
    new RubyProc(...));
}
```

Partial Evaluation

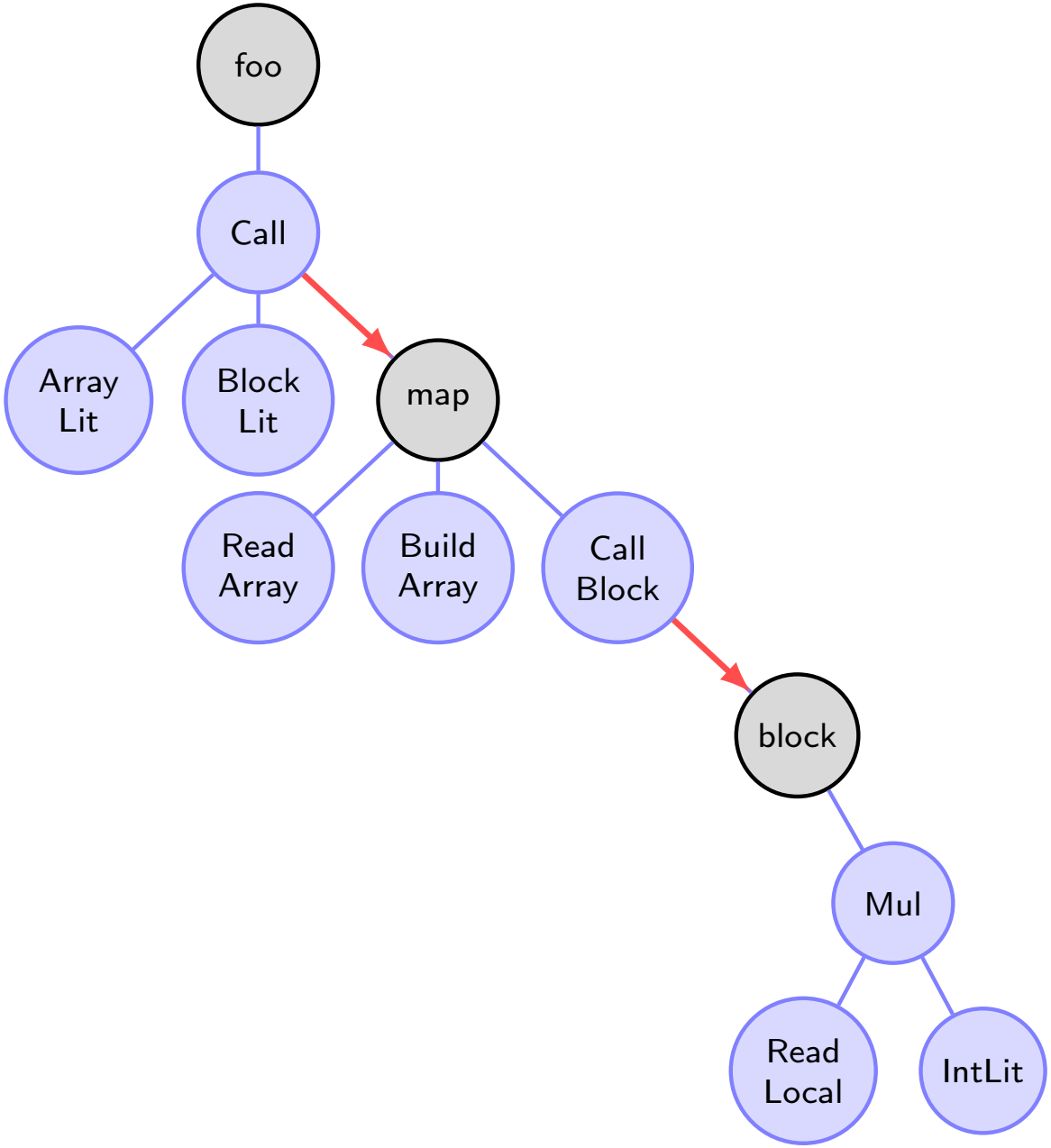
```
Object foo() {
  RubyArray array = new RubyArray(new int[2] { 1, 2 });
  RubyProc block = new RubyProc(this::block);
  return callMethod(array, "map", block);
}

RubyArray map(RubyArray array, RubyProc block) {
  RubyArray newArray = new RubyArray(new int[array.size]);
  for (int i = 0; i < array.size; i++) {
    newArray[i] = callBlock(block, array[i]);
  }
  return newArray;
}

Object block(Object e) {
  if (e instance Integer) {
    return Math.multiplyExact(e, 3);
  } else // deoptimize();
}
```

Inlining

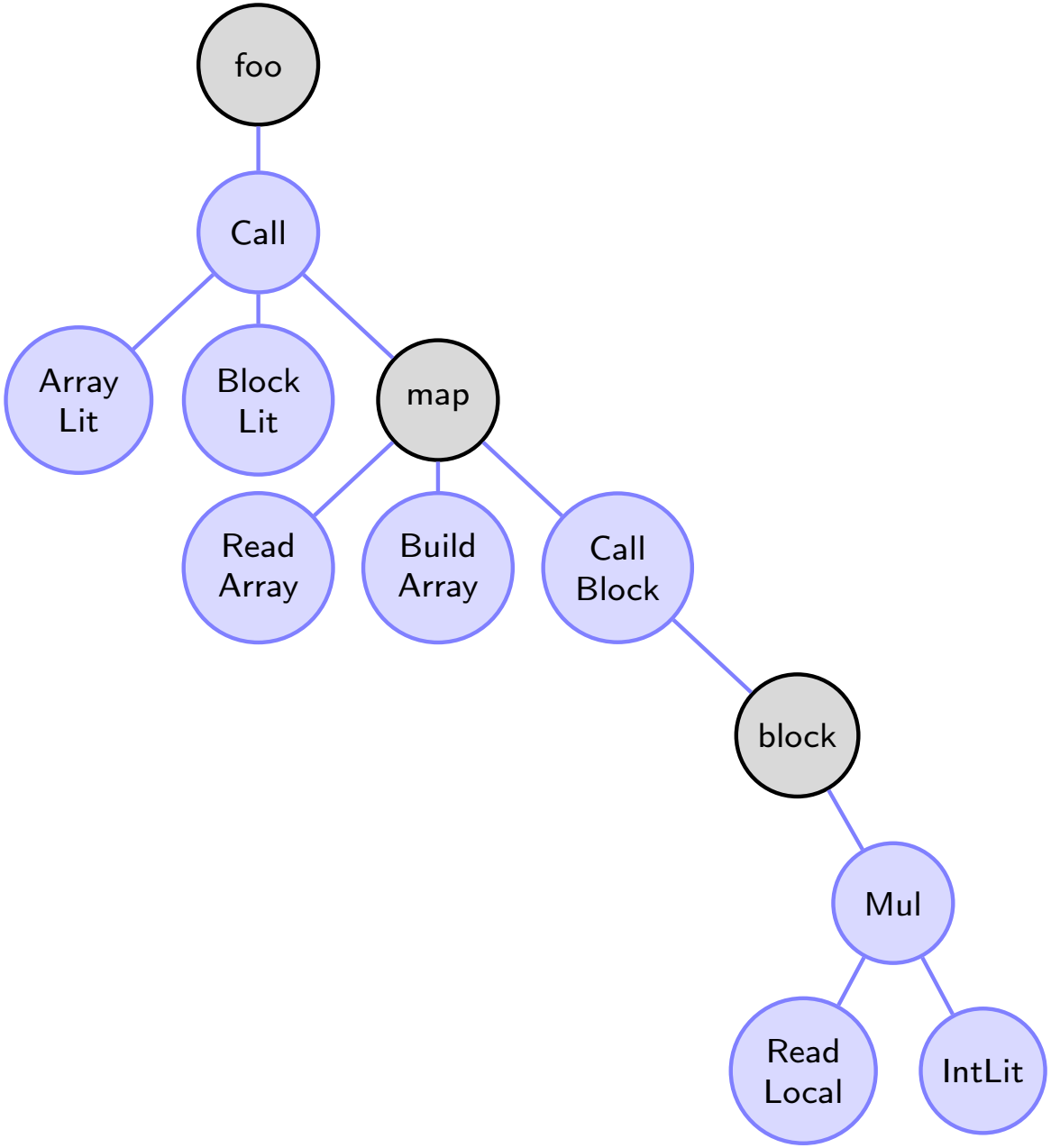
Inline caches in Call nodes cache which method or block was called



Inlining


foo
Call
Array Lit
Block Lit
map
Read Array
Build Array
Call Block
block
Mul
Read Local
IntLit

Inlining


foo
Call
Array Lit
Block Lit
map
Read Array
Build Array
Call Block
block
Mul
Read Local
IntLit

Inlining: inline map() and block()

```
Object foo() {
  RubyArray array = new RubyArray(new int[2] { 1, 2 });
  RubyArray newArray = new RubyArray(new int[array.size]);

  for (int i = 0; i < array.size; i++) {
    Object e = array[i];
    if (e instance Integer) {
      newArray[i] = Math.multiplyExact(e, 3);
    } else // deoptimize();
  }

  return newArray;
}
```

Escape Analysis: remove array

```
Object foo() {
  int[] arrayStorage = new int[2] { 1, 2 };
  RubyArray newArray = new RubyArray(new int[2]);

  for (int i = 0; i < 2; i++) {
    Object e = arrayStorage[i];
    if (e instance Integer) {
      newArray[i] = Math.multiplyExact(e, 3);
    } else // deoptimize();
  }

  return newArray;
}
```

Type propagation: arrayStorage is int[]

```
Object foo() {
  int[] arrayStorage = new int[2] { 1, 2 };
  RubyArray newArray = new RubyArray(new int[2]);

  for (int i = 0; i < 2; i++) {
    int e = arrayStorage[i];
    newArray[i] = Math.multiplyExact(e, 3);
  }

  return newArray;
}
```

Loop unrolling

```
Object foo() {
  int[] arrayStorage = new int[2] { 1, 2 };
  RubyArray newArray = new RubyArray(new int[2]);

  newArray[0] = Math.multiplyExact(arrayStorage[0], 3);
  newArray[1] = Math.multiplyExact(arrayStorage[1], 3);

  return newArray;
}
```

Escape Analysis: remove arrayStorage and replace usages

```
Object foo() {
  RubyArray newArray = new RubyArray(new int[2]);

  newArray[0] = Math.multiplyExact(1, 3);
  newArray[1] = Math.multiplyExact(2, 3);

  return newArray;
}
```

Constant Folding: multiplication of constants

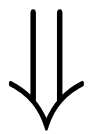
```
Object foo() {
  RubyArray newArray = new RubyArray(new int[2]);

  newArray[0] = 3;
  newArray[1] = 6;

  return newArray;
}
```

Escape analysis: reassociate reads/writes on same locations

```
Object foo() {
  return new RubyArray(new int[2] { 3, 6 });
}
```

TruffleRuby: Partial Evaluation + Graal Compilation

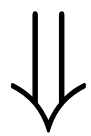
```ruby
def foo
  [1, 2].map { |e| e * 3 }
end
```



```
Object foo() {
  return new RubyArray(new int[2] { 3, 6 });
}
```

TruffleRuby: Partial Evaluation + Graal Compilation

```ruby
def foo
  [1, 2].map { |e| e * 3 }
end
```



```ruby
def foo
  [3, 6]
end
```

MJIT: The Method JIT for CRuby

The Ruby code is parsed to an AST, then transformed to bytecode

1. When a method is called many times, MJIT generates C code from Ruby bytecode
2. Call `gcc` or `clang` on the C code to create a shared library
3. Load the shared library and call the compiled C function

Current MJIT from CRuby trunk: generated C code

```ruby
def foo
  [1, 2].map { |e| e * 3 }
end
```

⇓ MJIT

```
VALUE foo() {
  VALUE values[] = { 1, 2 };
  VALUE array = rb_ary_new_from_values(2, values);
  return rb_funcall_with_block(array, "map", block_function);
}

VALUE block_function(VALUE e) {
  if (FIXNUM_P(e) && FIXNUM_P(3)) {
    return rb_fix_mul_fix(e, 3);
  } else (FLOAT_P(e) && FLOAT_P(3)) {
    ...
  } else goto deoptimize;
}
```

Clang/GCC: fold FIXNUM_P(3)

```
VALUE foo() {
  VALUE values[] = { 1, 2 };
  VALUE array = rb_ary_new_from_values(2, array_values);
  return rb_funcall_with_block(array, "map", block_function);
}

VALUE block_function(VALUE e) {
  if (FIXNUM_P(e)) {
    return rb_fix_mul_fix(e, 3);
  } else goto deoptimize;
}
```

Current MJIT: Cannot inline `rb_ary_map()`

`rb_ary_map()` is part of the Ruby binary

- MJIT knows nothing about it and cannot inline it
- foo() and the block cannot be optimized together

```
VALUE foo() {
  VALUE values[] = { 1, 2 };
  VALUE array = rb_ary_new_from_values(2, array_values);
  return rb_funcall_with_block(array, "map", block_function);
}

VALUE block_function(VALUE e) {
  if (FIXNUM_P(e)) {
    return rb_fix_mul_fix(e, 3);
  } else goto deoptimize;
}
```

Ruby Performance Summary

- The performance of Ruby can be significantly improved, as TruffleRuby shows
  - No need to rewrite applications in other languages for speed
- The JIT compiler needs access to the core library to be able to inline through it
- The JIT compiler needs to understand Ruby constructs
  - For example, C has no concept of Ruby object allocations, cannot easily understand it (it's just writes to the heap)

Parallel and Thread-Safe Ruby

The Problem

Dynamic languages have poor support for parallelism

The Problem

Dynamic languages have poor support for parallelism

Due to the implementations!

The Problem

Dynamic languages implementations have poor support for parallelism

Global Lock CRuby, CPython: cannot execute Ruby code in parallel in a single process. Multi processes waste memory and slow communication

The Problem

Dynamic languages implementations have poor support for parallelism

Global Lock CRuby, CPython: cannot execute Ruby code in parallel in a single process. Multi processes waste memory and slow communication

Unsafe JRuby, Rubinius: concurrent `Array#<<` raise exceptions!

The Problem

Dynamic languages implementations
have poor support for parallelism

Global Lock CRuby, CPython: cannot execute Ruby code in parallel in a single process. Multi processes waste memory and slow communication

Unsafe JRuby, Rubinius: concurrent `Array#<<` raise exceptions!

Share Nothing JavaScript, Erlang, Guilds: Cannot pass objects by reference between threads, need to deep copy.
Or, only pass deeply immutable data structures.

Guilds

- Stronger memory model, almost no shared memory and no low-level data races
- But objects & collections cannot be shared between multiple guilds, they have to be deep copied or transfer ownership
  - Unclear about scaling due to copy overhead. Shared mutable data (in their own Guild) can only be accessed sequentially.
- Existing libraries using Ruby Threads need some rewriting to use Guilds, it is a different programming model
- **Complementary**: some problems are easier to express with shared memory than an actor-like model and *vice versa*

Appending concurrently

```ruby
array = []

# Create 100 threads
100.times.map {
  Thread.new {
    # Append 1000 integers to the array
    1000.times { |i|
      array << i
    }
  }
}.each { |thread| thread.join }

puts array.size
```

Appending concurrently

CRuby, the reference implementation with a Global Lock:

```
ruby append.rb
100000
```

Appending concurrently

CRuby, the reference implementation with a Global Lock:

```
ruby append.rb
100000
```

JRuby, with concurrent threads:

```
jruby append.rb
64324
# or
ConcurrencyError: Detected invalid array contents due to
      unsynchronized modifications with concurrent users
     << at org/jruby/RubyArray.java:1256
```

Appending concurrently

CRuby, the reference implementation with a Global Lock:

```
ruby append.rb
100000
```

JRuby, with concurrent threads:

```
jruby append.rb
64324
# or
ConcurrencyError: Detected invalid array contents due to
      unsynchronized modifications with concurrent users
     << at org/jruby/RubyArray.java:1256
```

Rubinius, with concurrent threads:

```
rbx append.rb
Tuple::copy_from: index 2538 out of bounds for size 2398
                    (Rubinius::ObjectBoundsExceededError)
```

A Workaround: using a Mutex

```
array = [] # or use Concurrent::Array.new
mutex = Mutex.new

100.times.map {
  Thread.new {
    1000.times { |i|
      # Add user-level synchonization
      mutex.synchronize {
        array << i
      }
    }
  }
}.each { |thread| thread.join }

puts array.size
```

Problems of the Workarounds

- Easy to forget to wrap with `Mutex#synchronize` or to use `Concurrent::Array.new`
- If the synchronization is unnecessary, adds significant overhead
- Both workarounds prevent parallel access to the Array / Hash
- From the user point of view, Array and Hash are thread-safe on CRuby, thread-unsafe implementations are incompatible
  - Bugs were reported to Bundler and RubyGems because JRuby and Rubinius do not provide thread-safety for Array and Hash

A Hard Problem

How to make collections thread-safe,

and have no single-threaded overhead?

A Hard Problem

How to make collections thread-safe,

have no single-threaded overhead,

and support parallel access?

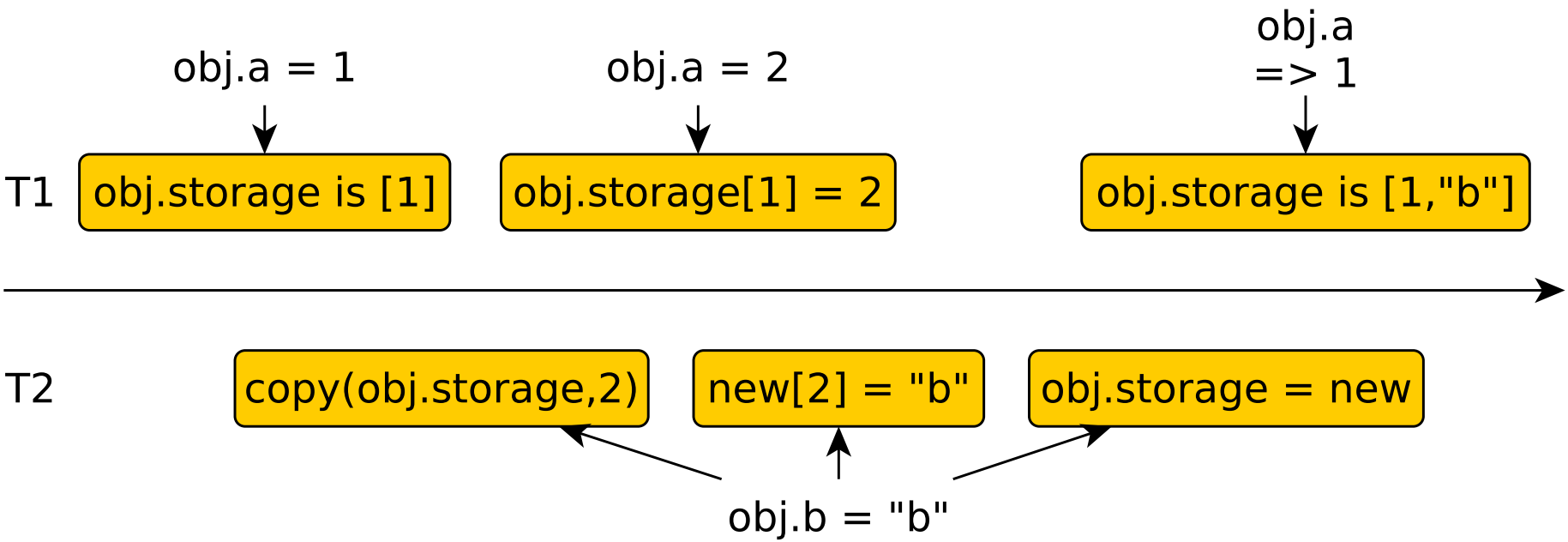
Similar Problem with Objects

Objects in dynamic languages support adding and removing fields at runtime

- The storage for fields in objects needs to grow dynamically
- Concurrent writes might be lost

Lost Object Field Updates

```
obj = Object.new
t1 = Thread.new { obj.a = 1; obj.a = 2 }
t2 = Thread.new { obj.b = "b" }
t1.join; t2.join
p obj.a # => 1 !!!
```

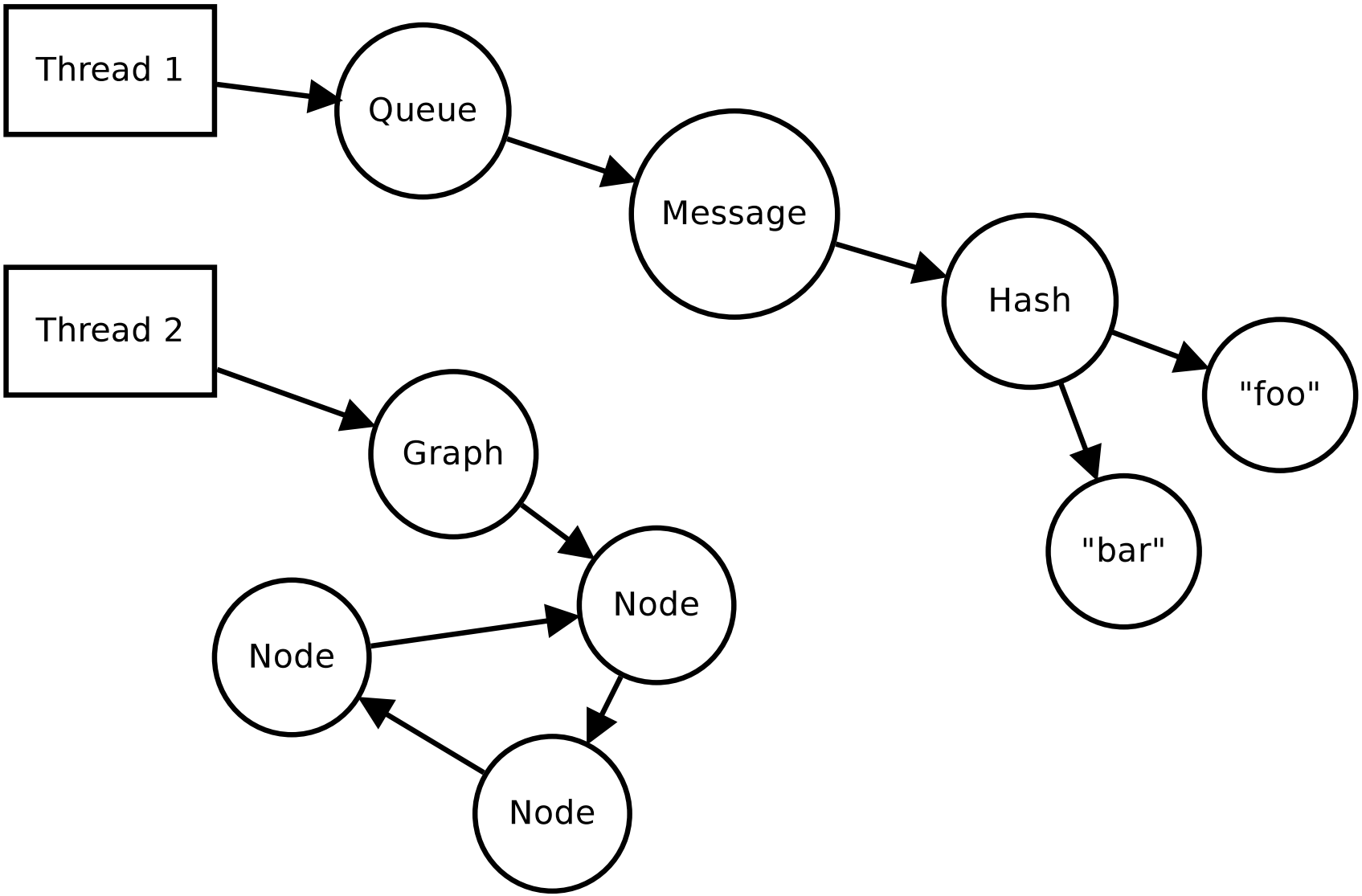


Idea: Distinguishing Local and Shared Objects

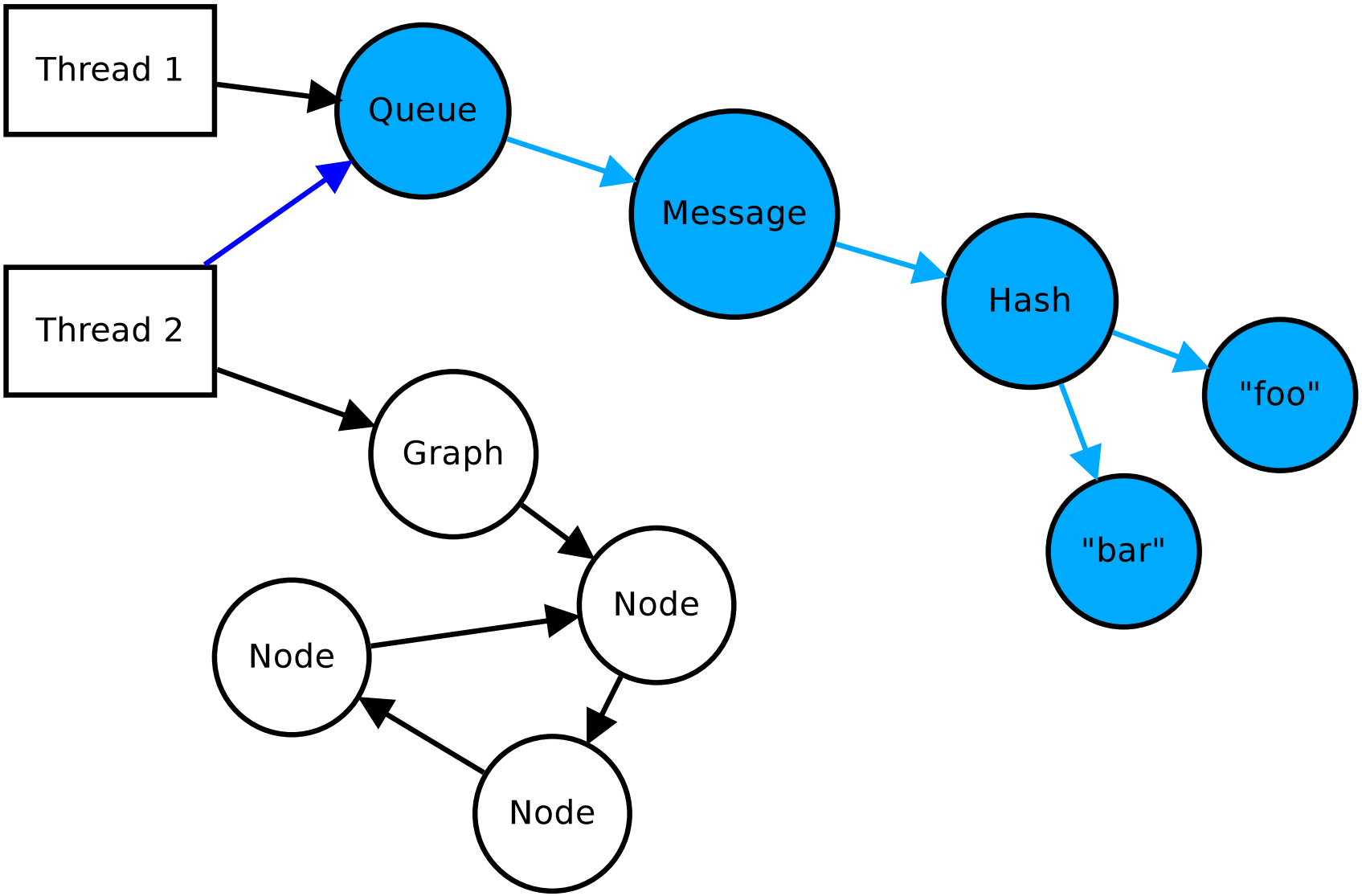
Idea: Only synchronize objects and collections which need it:

- Objects reachable by only 1 thread need no synchronization
- Objects reachable by multiple threads need synchronization

Local and Shared Objects: Reachability



Local and Shared Objects: Reachability



Write Barrier: Tracking the set of shared objects

- Write to shared object $\Longrightarrow$ share value, transitively

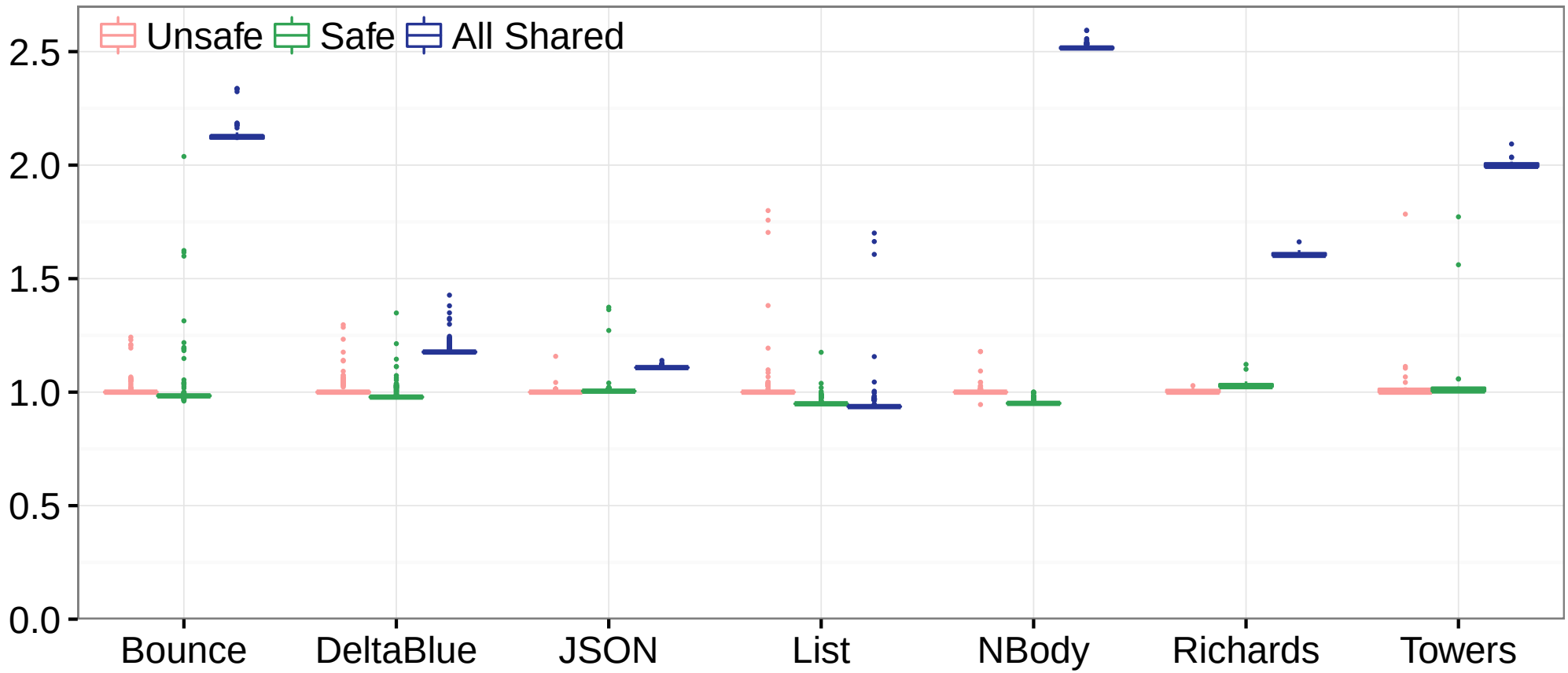
```
# Share 1 Hash, 1 String and 1 Object
shared_obj.field = { "a" => Object.new }
```

- Shared collections use a write barrier when adding elements

```
shared_array << Object.new # Share 1 Object
shared_hash["foo"] = "bar" # Share the key and value
```

Single-Threaded Performance for Objects

Peak performance, normalized to *Unsafe*, lower is better

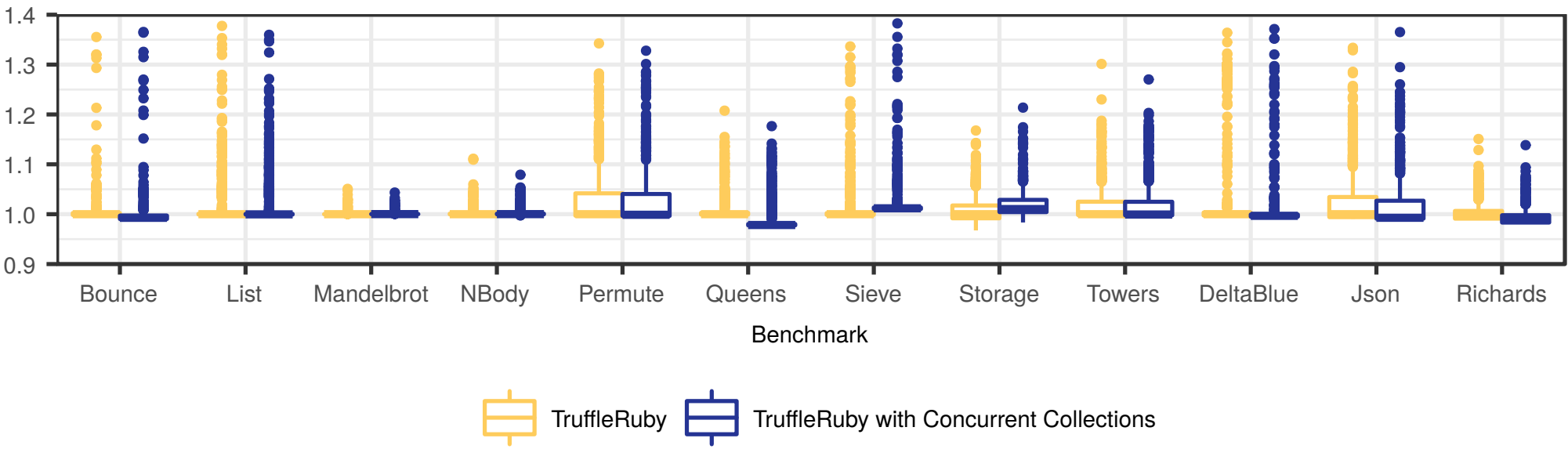


All Shared synchronizes on all object writes (similar to JRuby)

Benchmarks from *Cross-Language Compiler Benchmarking: Are We Fast Yet?*
S. Marr, B. Daloze, H. Mössenböck, 2016.

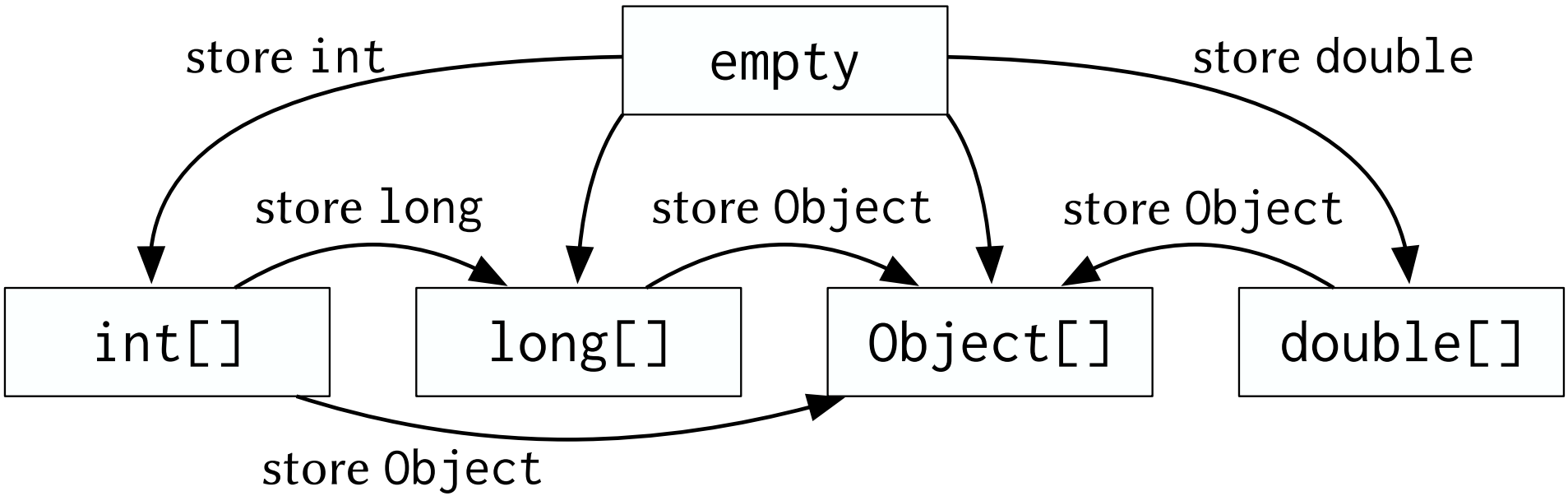
Single-Threaded Performance for Collections

Peak performance, normalized to *TruffleRuby*, lower is better



No difference because these benchmarks do not use shared collections.

Array Storage Strategies




```
array = [] # empty
array << 1 # int[]
array << "foo" # Object[]
```

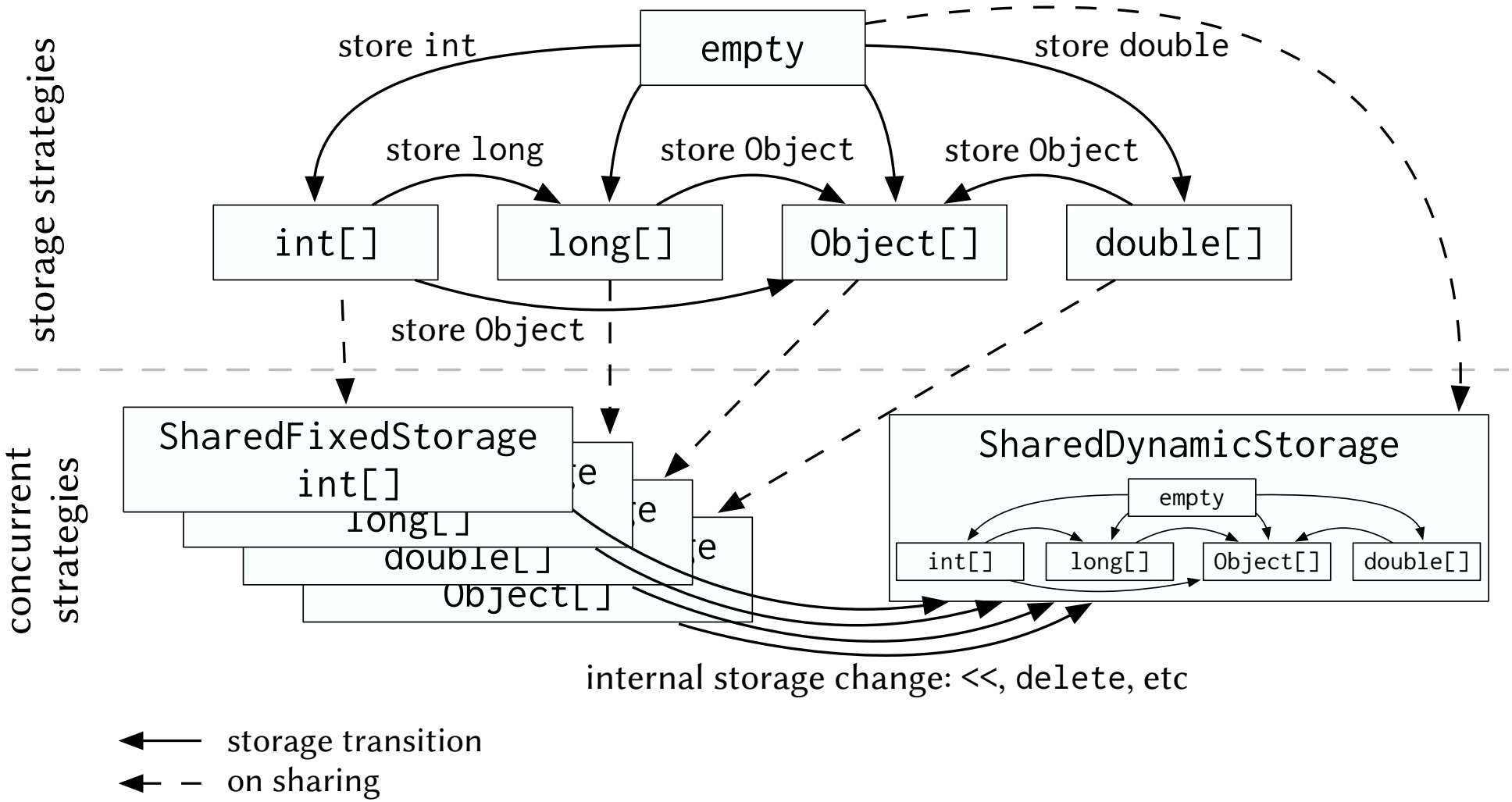
Storage Strategies for Collections in Dynamically Typed Languages
C.F. Bolz, L. Diekmann & L. Tratt, OOPSLA 2013.

Goals for Shared Arrays

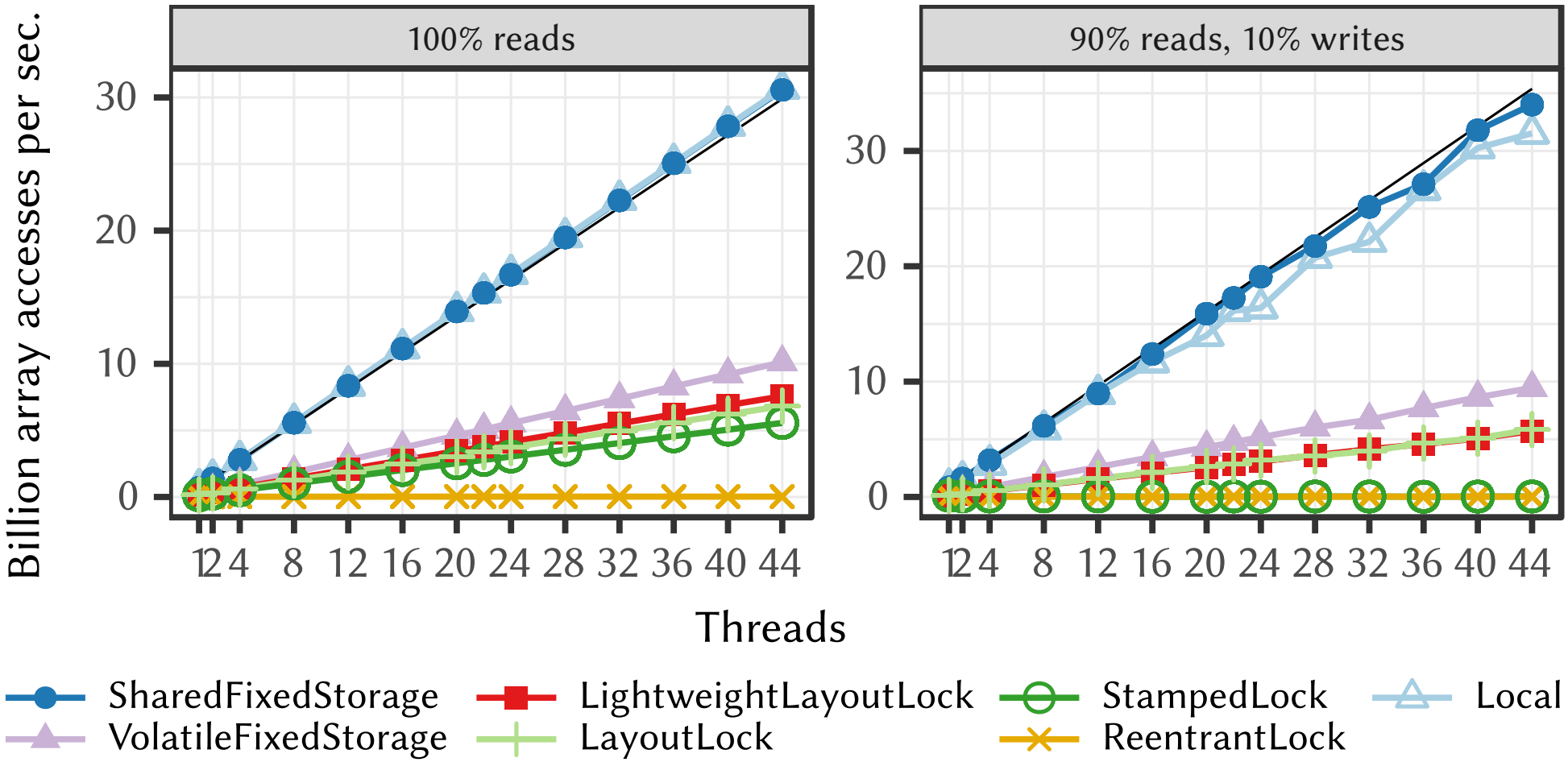
Goals:

- All Array operations supported and thread-safe
- Preserve the compact representation of storage strategies
- Enable parallel reads and writes to different parts of the Array, as they are frequent in many parallel workloads

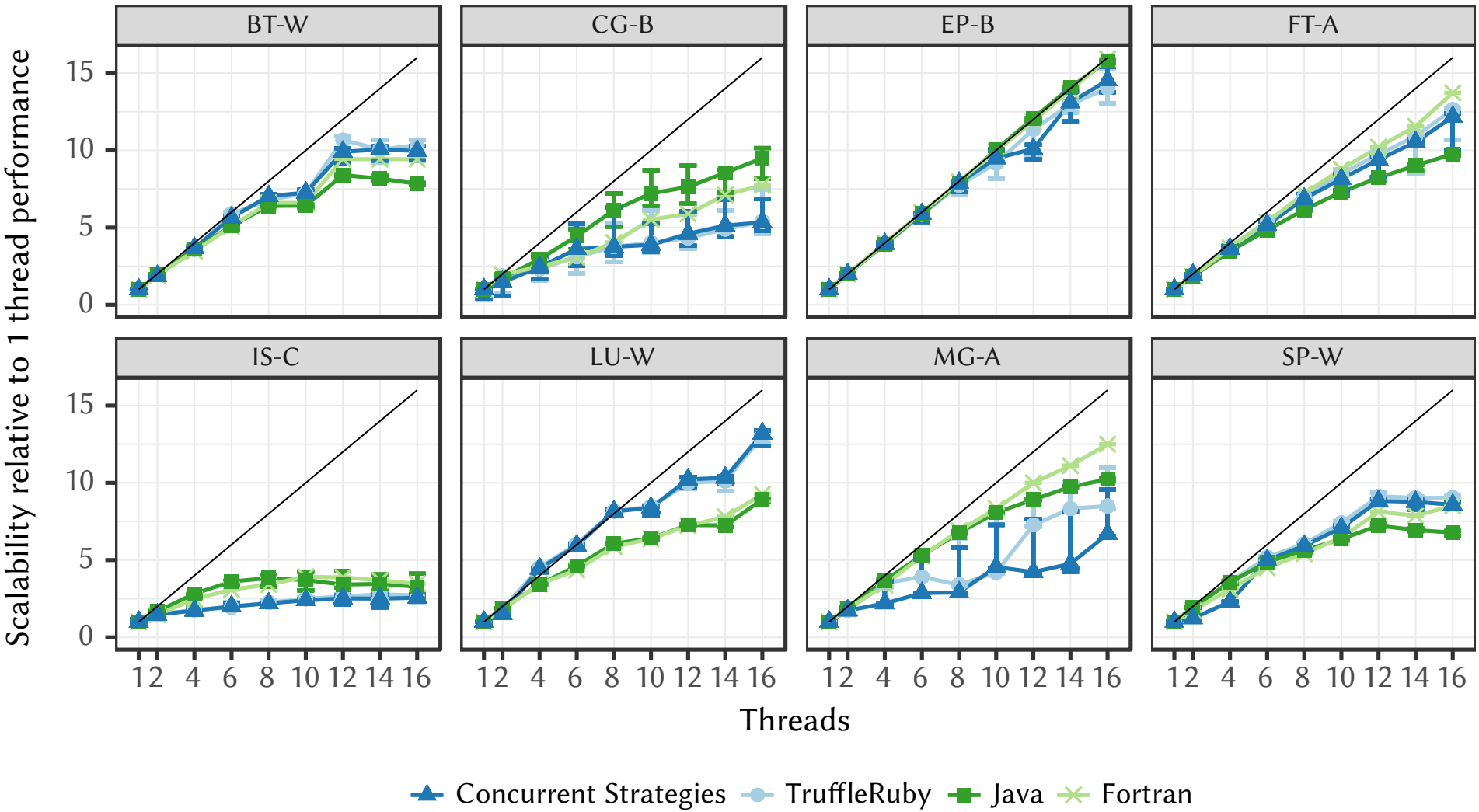
Concurrent Array Strategies



Scalability of Array Reads and Writes



NASA Parallel Benchmarks



TruffleRuby Conclusion

- TruffleRuby runs *unmodified* Ruby code faster, as fast as the best dynamic languages implementations like V8 for JavaScript
- TruffleRuby can run Ruby code in parallel safely: "GIL bugfix"
  - Safe Array/Hash not in `master` yet (ETA: this summer)
- TruffleRuby can run existing Ruby code using Threads in parallel, no need to change to a different programming model

Conclusion

- The performance of Ruby can be significantly improved, as TruffleRuby shows
  - No need to rewrite applications in other languages for speed
- We can have parallelism and thread-safety for objects and collections in Ruby, with no single-threaded overhead
- We can execute Ruby code in parallel, with the most important thread-safety guarantees and scale to many cores

Trying TruffleRuby

Soon:

```
$ ruby-install truffleruby
```

```
$ rbenv install truffleruby
```

```
$ rvm install truffleruby
```

GraalVM 1.0

- GraalVM 1.0RC1 was released (17 April 2018)

  `http://www.graalvm.org/`

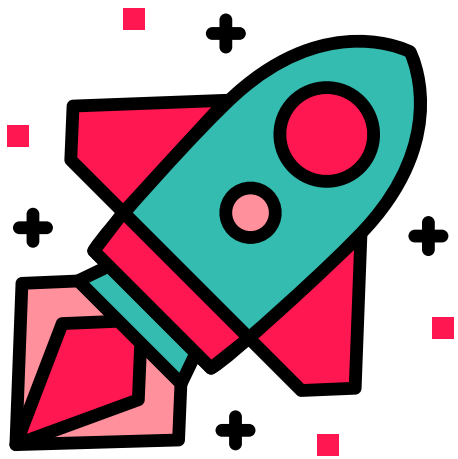
- Open-Source Community Edition and Enterprise Edition

- TruffleRuby, JavaScript and Node.js, R, Python and LLVM bitcode (C, C++, Rust) in a single VM!

- All these languages can interoperate easily and efficiently

Parallel and Thread-Safe Ruby at High-Speed with TruffleRuby

Benoit Daloze
@eregontp



TRUFFLE
RUBY