Exploring JRuby, Truffle and Graal

Virtual Machines and Execution Environments, WS2014/15

Jan Graichen, Fabio Niephaus, Matthias Springer, Malte Swart

Hasso Plattner Institute, Software Architecture Group

February 5, 2015

HPI

Overview

HPI

Recap: What is Truffle & Graal?

- Truffle and Graal is a tool chain to build fast VMs easily
  - Similar to RPython
- Truffle is an AST interpreter framework
- Graal is modified JVM
  - Comes with an aggressive JIT compiler written in Java
  - Profiles code and detects hot methods
  - Truffle can use these information for making assumptions
  - Compiles specified code segment into machine code
- Truffle uses node replacements for specific optimizations (like type specific actions)

Overview

Demo

```ruby
def multiply(a, b)
  a * b
end

100_000.times.each do |times|
    start = Time.now
    (1..1_000_000).each do |i|
        multiply(1, 2)
    end
    end_time = Time.now
    puts "Time elapsed #{(end_time - start)*1000} ms"
end
```

HPI

Example Runtimes

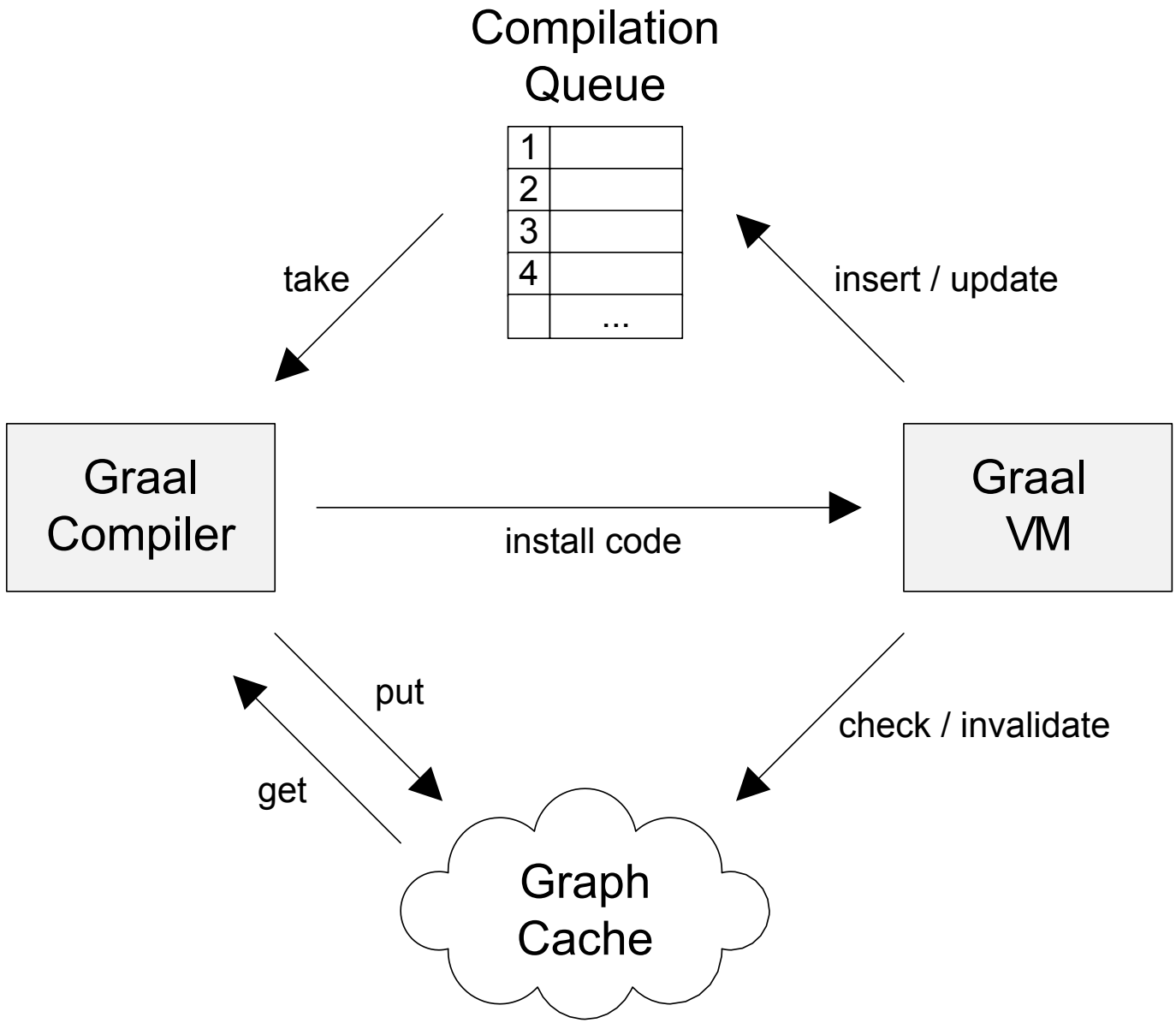
Empirical Figures

1. **MRI:** 175ms
2. **JRuby:** 80ms
3. **JRuby + Truffle:** 720ms
4. **JRuby + Graal:** 180ms and then 70ms
5. **JRuby + Truffle + Graal:** 1.5ms

Warm-Up Time

Truffle and Graal end with a very low execution time per iteration, but has large boot up time
→ Only faster if there is a large number of iterations/long overall execution time

Graal VM - System Architecture

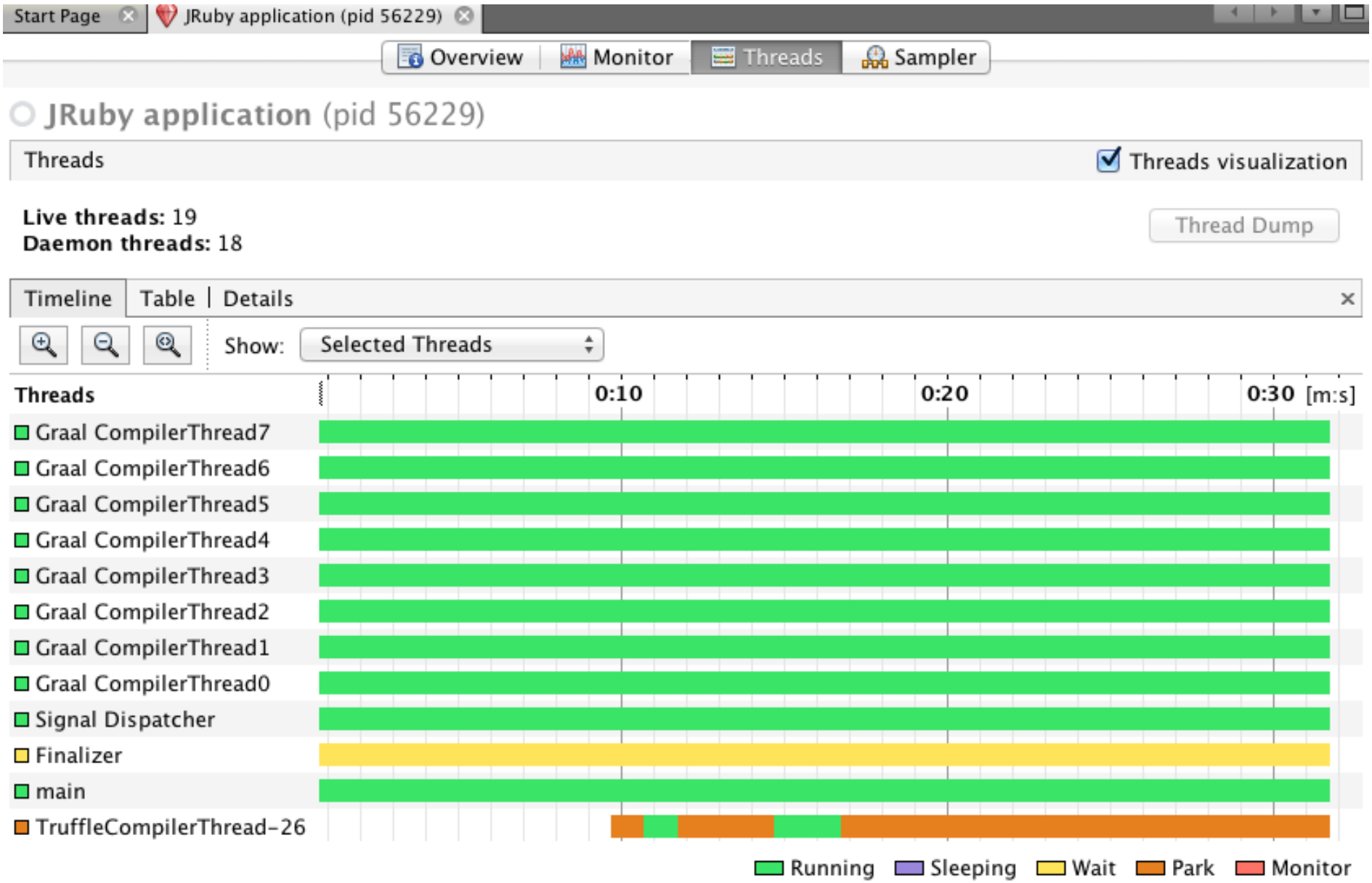


HPI

Handout only: Graal VM - Details

1. Graal VM detects *hot* methods
2. Graal VM adds these methods to compilation queue
3. Compiler threads compile methods with highest priorities
4. Machine code is installed into runtime's cache

JRuby, Truffle and Graal: Overview of Threads

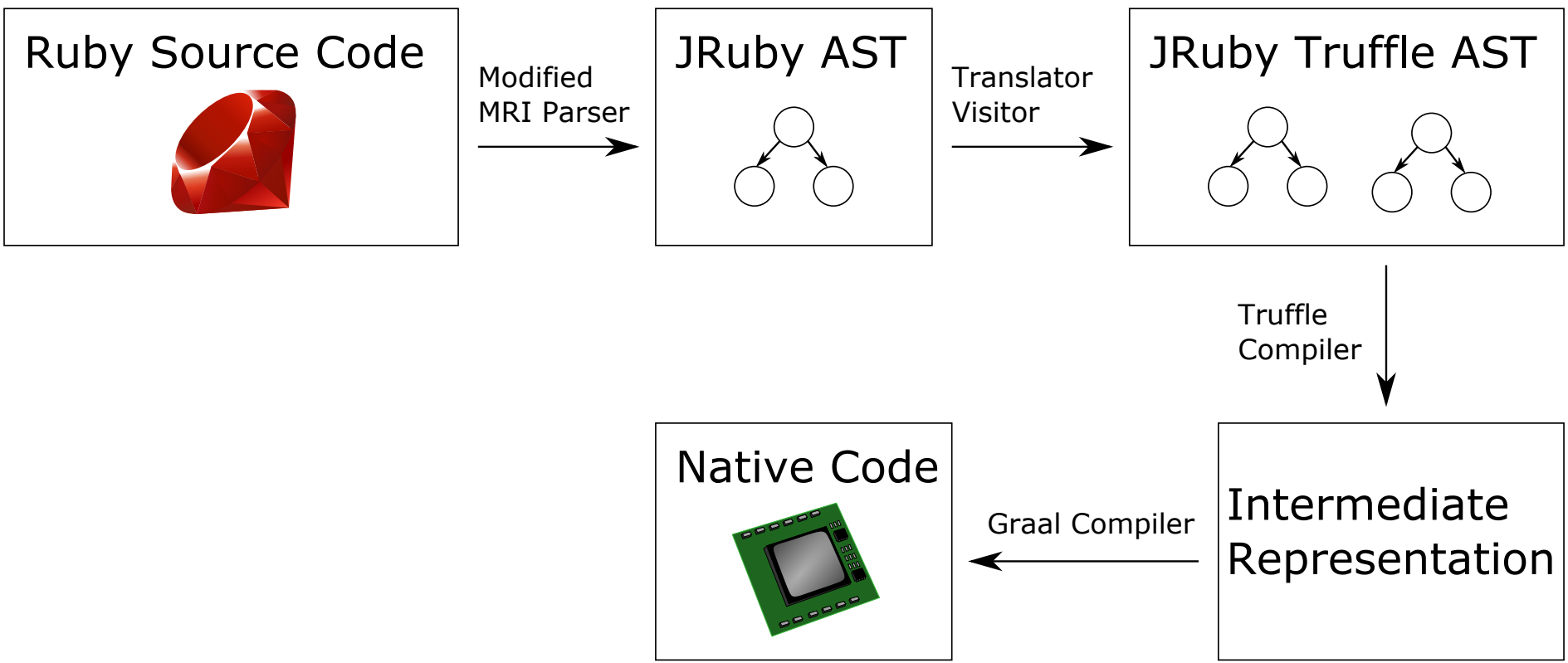


Overview

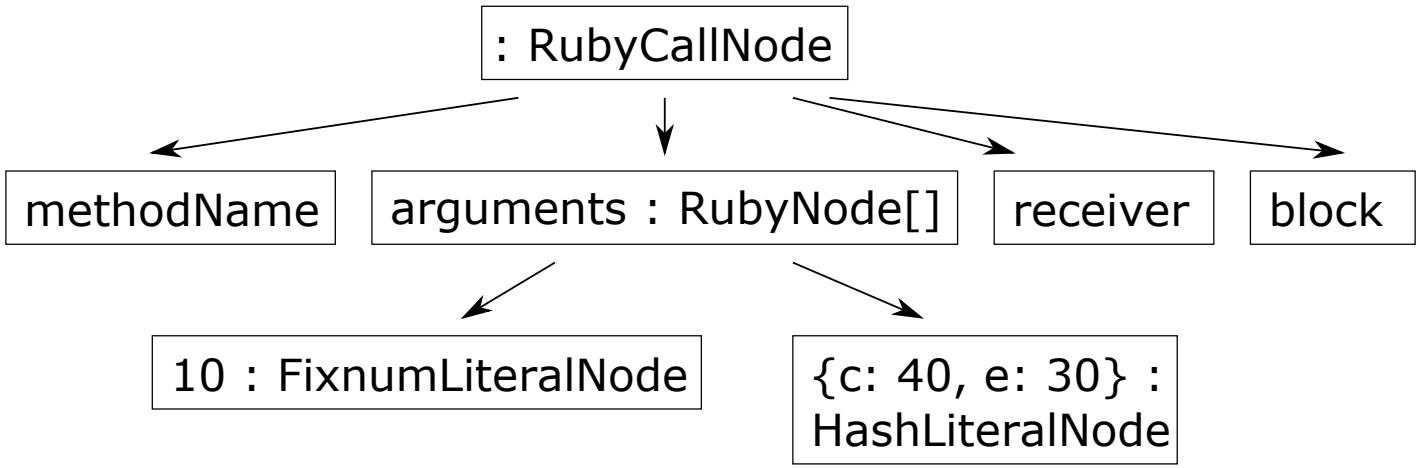
Ways to use Truffle within an existing AST Interpreter

Convert to Truffle: Translate all AST nodes to Truffle nodes

Add-On Truffle: Add an additional set of AST nodes



Method Call Nodes in (J)Ruby



- `RubyCallNode` contains:
  - Receiver object
  - Method name (fix)
  - List of argument AST nodes
  - Block AST node
- Dynamic call → Dynamic dispatch is run on every execution

HPI

Method Callee Node in (J)Ruby

1. `RubyRootNode`
2. `Catch*Nodes` (`CatchNextNode`, `CatchRetryAsErrorNode`, `CatchReturnNode` ...)
3. `SequenceNode`
   - 3.1 `CheckArityNode`
   - 3.2 `WriteLocalVariableNode` for argument 1
   - 3.3 `WriteLocalVariableNode` for argument 2
   - 3.4 `WriteLocalVariableNode` for kwargument e
   - 3.5 `WriteLocalVariableNode` for kwargument c
   - 3.6 Statement sequence itself (wrapped in `TracingNodes`, with `CyclicAssumptionS`)

Nice: Every argument has a node to create its default argument, maybe a node that throws every time a exception

HPI

Overview

Task: Keyword Arguments in Ruby 2.x

- Shortcut to call method with dictionary as last argument:

```
method(10, e: 30, c: 40)
method(10, {:e => 30, :c => 40})
```

- Starting with Ruby 2.0, Ruby can process this dictionary automatically (so called keyword arguments):

```
def method(a, b=3, e:, c:30)
end
```

HPI

Performance Bottlenecks

- `Hash` object creation: object is created, passed as argument, then destructed again
- Inefficient code paths (e.g., multiple scans of `Hash` object)
- Code involving `Hash` objects is harder to optimize than code involving primitive objects (Graal optimizations)
- Keyword argument nodes are not optimized by Truffle (Java `equals`, Truffle boundary for `Hash` iterator)
- Execution remains in interpreter modus

Goal: Pass keyword arguments as normal arguments

HPI

Optimizations

1. Optimize implementations (efficient hash operations)
2. Store kwargs within normal arguments array, separated by marker
3. Cache kwargs mapping within dispatch chain

→ We will now look into optimization #3

HPI

Handout only: Fully Optimized Keyword Arguments

Callee's Point of View

- `VirtualFrame` contains `arguments` array.
- Array contains `Marker` object, generated by `MarkerNode` as last element, if call is optimized.
- `CachedBoxedDispatchNode` is always optimized if keyword arguments are present (rewriting of `argumentNodes` array).
- `ReadKeywordArgumentNode` has offset (from right side) into `arguments` array as instance variable.
- `ReadKeywordArgumentNode` accesses `arguments` array at offset if call is optimized, otherwise expects a `RubyHash` (old behavior).
- `CachedBoxedDispatchNode` might generate an additional `RubyHash` if rest keyword arguments are present.

Fully Optimized Keyword Arguments

Example

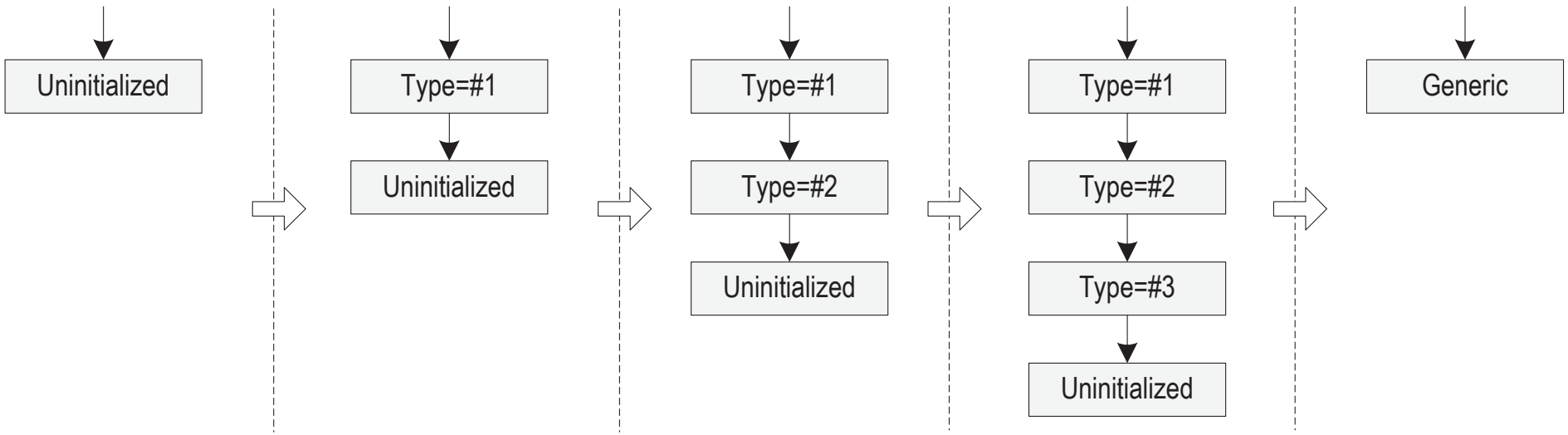
```
class Cls1
    def method(a:, **kwargs)
    end
end

class Cls2
    def method(a:, b:)
    end
end

[Cls1.new, Cls2.new].each do |obj|
   obj.method(a: 1, b: 2)
end
```

Recap: Type Decision Chains

Source: "Self-Optimizing AST Interpreters"

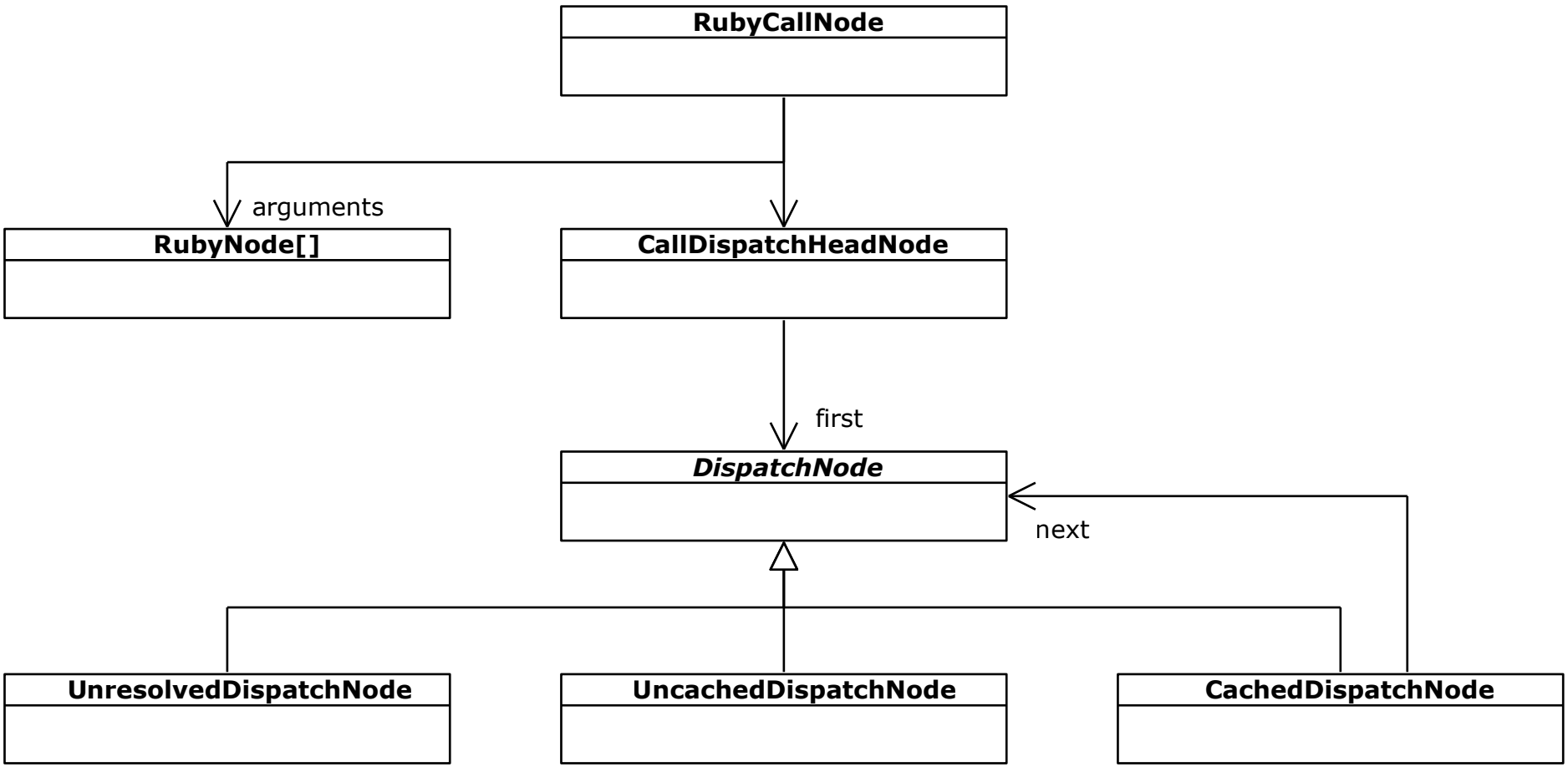


Fully Optimized Keyword Arguments

Problems

- Nodes are specific with regard to user-defined Ruby classes (cannot use Truffle DSL)
- Truffle DSL supports only specialization for language types
- Type of receiver is not known before dispatching the call

Guest Language PIC in JRuby

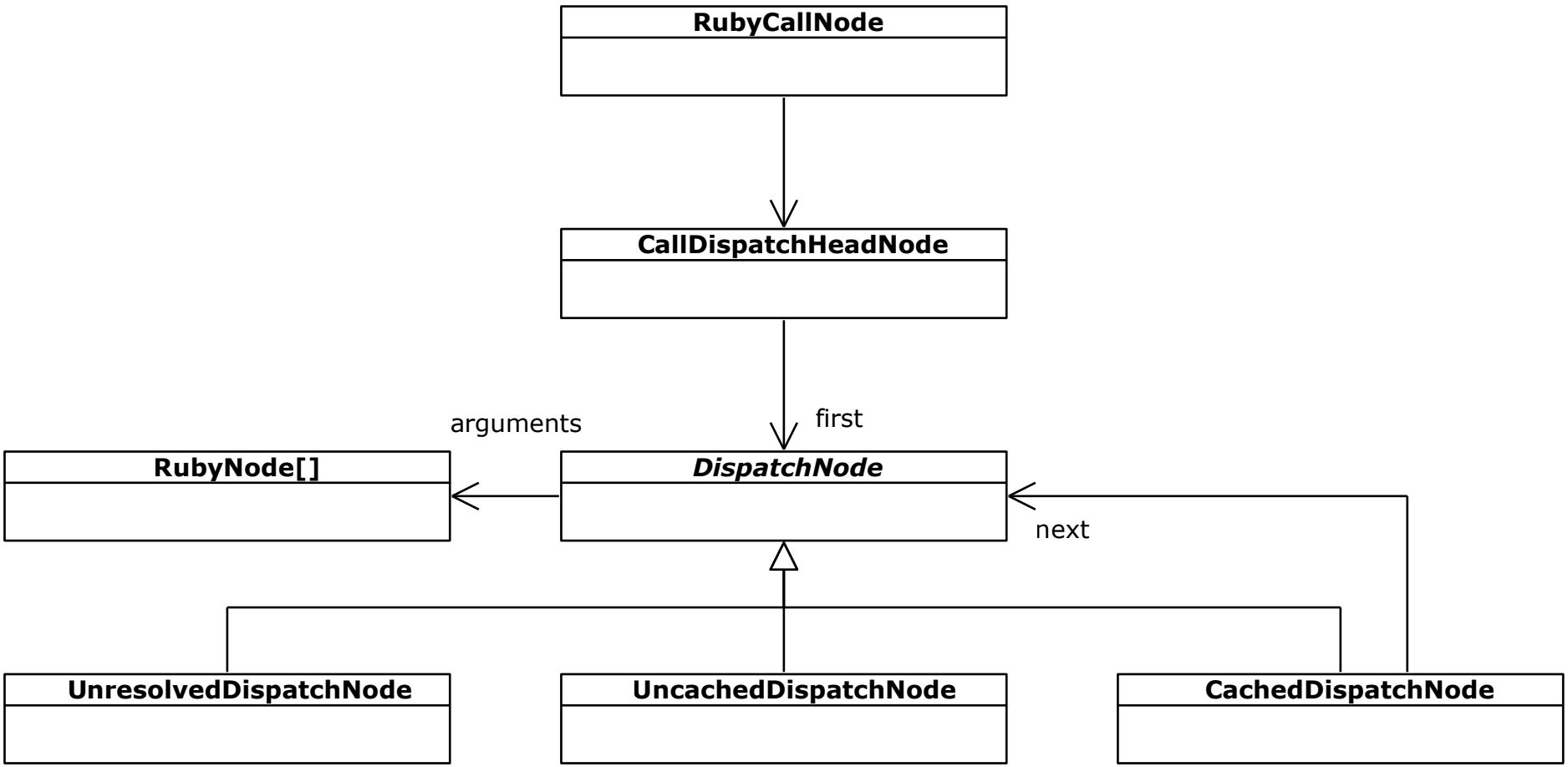


HPI

Handout only: Guest Language PIC in JRuby

- `UnresolvedDispatchNode`: corresponds to Truffle's *unspecified node*
- `UncachedDispatchNode`: corresponds to Truffle's *generic node*
- `CachedDispatchNode`: corresponds to Truffle's *specialized nodes*
- Node rewriting similar to Truffle but without Truffle

Argument Passing in *DispatchNode*



HPI

Handout only: Argument Passing in `DispatchNode`

- Unmodified arguments array (possible with `HashLiteralNode`) is stored in `UnresolvedDispatchNode`
- `CachedDispatchNode` contains keyword arguments mentioned in signature in array, and other keyword arguments in `HashLiteralNode`
- `ReadKeywordArgumentNode` checks if method dispatch is optimized (marker present in arguments array) and reads keyword arguments from arguments array, otherwise extracts them from `Hash` (same as before)

HPI

Evaluation

Results

Keyword arguments are as fast as position arguments
(for specific but common cases)

HPI

Handout only: Evaluation

→ Keyword arguments are as fast as position arguments

- Optimization affects only arguments passed in keyword argument syntax in method calls
- Optimization does not affect keyword arguments passed as an already existing `Hash`

HPI

Overview

Truffle Summary

- Specific Java code cannot be translated by Graal (or it is disallowed)
- Large AST interpreters can still get unclear/distracting, knowledge is the composition of nodes, not the nodes itself
- Truffle DSL is not enough for efficient implementation of complex languages
- It is still needed to write efficient code and node implementations

Truffle and RPython - A Very Subjective Comparison

RPython

- Lightweight stack
- A little bit easier to get to work - mostly getting the correct libs in the Python path
- Difficult to debug in depth what is happening at execution

Truffle

- Heavy stack (Java, mostly multiple JDK and often maven . . .)
- If you get it working, you have the full power of (debugging) Java, even Graal itself

References

- L. Stadler, G. Duboscq, H. Mössenböck, T. Wurthinger, **Compilation Queuing and Graph Caching for Dynamic Compilers**, http://lafo.ssw.uni-linz.ac.at/papers/2012_VMIL_Graal.pdf
- T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, M. Wolczko. **One VM to Rule Them All**, 2013, http://lafo.ssw.uni-linz.ac.at/papers/2013_Onward_OneVMToRuleThemAll.pdf
- T. Würthinger, A. Woß, L. Stadler, G. Duboscq, D. Simon, C. Wimmer. **Self-Optimizing AST Interpreters**, 2012, http://lafo.ssw.uni-linz.ac.at/papers/2012_DLS_SelfOptimizingASTInterpreters.pdf
- Graal (http://hg.openjdk.java.net/graal/graal)
- JRuby (https://github.com/jruby/jruby)
- JRuby Developers (especially Chris Seaton)
- JRuby Benchmarks (https://github.com/jruby/bench9000)