Efficient and Thread-Safe Objects for Dynamically-Typed Languages

Benoit Daloze  Stefan Marr  Daniele Bonetta  Hanspeter Mössenböck

(42,-5)

shape
42
-5

(22,-33)

shape
22
-33

Shape
transitions
x: int@1
y: int@2

(42,-5,27)

shape
42
-5
27

z: int@3
We are in the multi-core era, but:

- Dynamic languages have poor support for parallel execution (e.g.: Ruby, Python, JavaScript, ...)
- Object models are not thread-safe or inefficient
- Allow adding or removing fields at run time
How is this executed?

@field

@field = value
How is this executed?

@field

@field = value

... when done concurrently on the same object?
A simple class

class Foo
  def a
    @a
  end
  def a=(v)
    @a = v
  end

  def b
    @b
  end
  def b=(v)
    @b = v
  end
end
obj = Foo.new

Thread.new {
  obj.a = "a"
}

Thread.new {
  obj.b = "b"

  obj.fields # => [:a, :b] OK
  obj.b      # => "b" OK
}
What could go wrong?

```ruby
obj = Foo.new

Thread.new {
  obj.a = "a"
}

Thread.new {
  obj.b = "b"

  obj.fields # => [:a] ??
  obj.b # => nil ??
}
```
What could go wrong?

```ruby
obj = Foo.new

Thread.new {
  obj.a = "a"
}

Thread.new {
  obj.b = "b"

  obj.fields # => [:b] OK
  obj.b      # => "a" ??
}
```
Objects Models

The Problems

One Solution

Performance
Based on **maps** from the SELF programming language

The Truffle Object Storage Model

An Object Storage Model for the Truffle Language Implementation Framework
The Truffle Object Storage Model

An Object Storage Model for the Truffle Language Implementation Framework
The Problems

Objects Models

The Problems

One Solution

Performance
The 3 Safety Problems

- Lost Field Definitions
- Out-Of-Thin-Air Values
- Lost Field Updates
Lost Field Definitions

T1

Find shape with a

obj.a = "a"

obj.shape = \{a\}

T2

Find shape with b

obj.b = "b"

obj.shape = \{b\}

obj.shape is \{a\}

obj.instance_variables => [:a]
Out-Of-Thin-Air Values

T1

obj.shape = {a@1}  
obj.storage[1] = "a"

T2

obj.shape = {b@1}  
obj.storage[1] = "b"  
obj.shape is {a@1}  
obj.storage is ["b"]

obj.a = "a"  
obj.b = "b"  
obj.a => "b"
Lost Field Updates

T1
- obj.a = 1
- obj.storage is [1]

T2
- copy(obj.storage, 2)

T1
- obj.a = 2
- obj.storage[1] = 2

T2
- new[2] = "b"
- obj.b = "b"

T3
- obj.a => 1
- obj.storage is [1, "b"]

T2
- obj.storage = new
Defining a new field

- Grow the object storage (allocate, copy, update pointer)
  ```python
  obj.storage = copy(obj.storage, size+1)
  ```
  and write the value:
  ```python
  obj.storage[size-1] = value
  ```

- Update the Shape pointer:
  ```python
  obj.shape = newShape
  ```

Two reference fields cannot be read and written atomically, unless using synchronization!
Can we just synchronize field updates?

![Graph showing median time per 10M writes (ms) when writing to a field and loop. The graph compares 'Unsafe' and 'Synchronized' methods. The 'Synchronized' method has a much higher median time of 290 ms compared to 30 ms for the 'Unsafe' method.]
One Solution

Objects Models

The Problems

One Solution

Performance
Local and Shared Objects

Thread 1
- Queue
- Message
- Hash
  - "foo"
  - "bar"

Thread 2
- Graph
  - Node
  - Node
Local and Shared Objects

Thread 1

Queue

Thread 2

Message

Hash

"foo"

"bar"

Node

Node

Node

Graph
Synchronize only on shared objects writes

Choices:
- Synchronize only on shared objects \textit{writes}
- Unsynchronized \textit{reads} on shared objects

Motivation:
- \textit{Reads} are more frequent than \textit{writes} on shared objects
- $28 \times$ more frequent in concurrent DaCapo benchmarks!

\textit{A Black-box Approach to Understanding Concurrency in DaCapo.}
\textit{T. Kalibera, M. Mole, R. Jones, and J. Vitek, 2012.}
One Solution: synchronize on shared objects

- **Lost Field Definitions and Updates**
  Synchronize writes, but only on *shared objects*
  *Local* objects need no synchronization

- **Out-Of-Thin-Air Values**
  Different storage locations for each field:
  A storage location of an object is only ever used for one field
Tracking the set of shared objects

- All globally-reachable objects are initially *shared*, transitively

- Write to shared object \( \implies \) share value, transitively

  ```ruby
  # Share 1 Array, 1 Object, 1 Hash and 1 String
  shared_obj.field = [Object.new, { "a" => 1 }]
  ```
Sharing: writing to a field of a shared object

```java
void share(DynamicObject object) {
    if (!isShared(obj.shape)) {
        object.shape = sharedShape(obj.shape);
        for (location : obj.getObjectLocations()) {
            share(location.get(obj)); // recursive call
        }
    }
}

void writeBarrier(DynamicObject sharedObject, Object value) {
    if (value instanceof DynamicObject) {
        share(value);
    }
    synchronized (sharedObject) {
        location.set(sharedObject, value);
    }
}
```
Sharing a Rectangle containing two Points

```ruby
shared_obj.field = Rectangle.new(
  Point.new(1, 2),
  Point.new(4, 3))
```
Optimized Sharing for a Rectangle and two Points

Object Graph

Rectangle

Point
1
2

Point
4
3

Truffle AST

obj.field = rectangle

Share Rectangle

Share Point

Share Point

void shareRectangle(DynamicObject rect) {
    if (rect.shape == localRectangleShape) {
        rect.shape = sharedRectangleShape;
    } else {
        /* Deoptimize */
    }
}

DynamicObject tl = rect.object1;
if (tl.shape == localPointShape) {
    tl.shape = sharedPointShape;
} else { /* Deoptimize */
}

DynamicObject br = rect.object2;
if (br.shape == localPointShape) {
    br.shape = sharedPointShape;
} else { /* Deoptimize */
}
Performance

Objects Models

The Problems

One Solution

Performance
Performance: Are we fast yet?

---

Cross-Language Compiler Benchmarking: Are We Fast Yet?
Impact on Sequential Performance

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

*All Shared* synchronizes on all object writes.

All object-related benchmarks from *Cross-Language Compiler Benchmarking: Are We Fast Yet?* S. Marr, B. Daloze, H. Mössenböck, 2016.
Performance for Parallel Actor Benchmarks

Benchmarks from
Conclusion

- Concurrently growing objects need synchronization to not lose updates or new fields

- Distinguish local/shared objects reduces overhead
  - Only synchronize on shared object writes
  - Needs a write barrier (can be specialized)

- Thread-safe objects in dynamic languages
  - Zero cost on sequential peak performance
  - Low overhead on parallel code
Efficient and Thread-Safe Objects for Dynamically-Typed Languages

Benoit Daloze  Stefan Marr  Daniele Bonetta  Hanspeter Mössenböck