Thread-Safe and Efficient Data Representations in Dynamically-Typed Languages

Benoit Daloze

Supervisor	Prof. Hanspeter Mössenböck
Second Examiner	Dr. Jeremy Singer
Third Examiner	Prof. Gabriele Anderst-Kotsis
Präses	Prof. Armin Biere

JOHANNES KEPLER
UNIVERSITÄT LINZ

15 November 2019

Motivation: Multi-Core Processors

- Single-core performance no longer improves as it used to.
- The main way to achieve higher CPU performance on a single machine is to use multi-core processors.

The Multi-Core Era

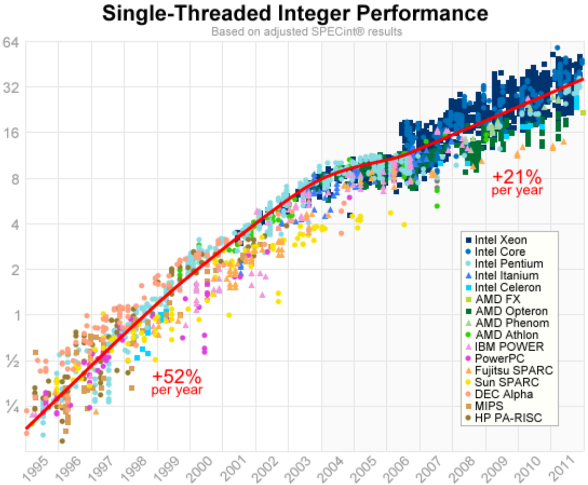


Figure: SPECint® results over the years. Source:

https://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance/

The Multi-Core Era

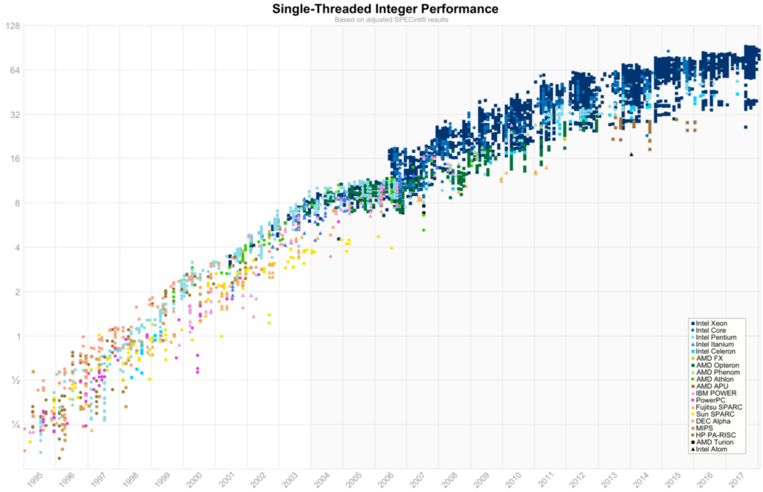


Figure: SPECint® results until April 2018. Source: `https://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance/#IDComment1061418665`

Problem Statement

We are in the multi-core era, but:

- Dynamically-typed languages have poor support for parallel execution (e.g.: Ruby, Python, JavaScript, . . .)
- State-of-the-art implementations either sequentialize execution or lack basic thread-safety, exposing low-level data races of the VM to the user
- The biggest reason (my interpretation) is their representation of objects and built-in collections are not thread-safe

Does performance matter in these languages?

- Many server applications written in Ruby, Python, JavaScript, etc
- They are run in production, and the cost is linked to how many resources are used (servers, memory, processing power)

- Global Lock: CRuby, CPython, PyPy, V8, ...
  $\Rightarrow$ No shared-memory parallelism in a single process

Dynamic Language Implementations

- Global Lock: CRuby, CPython, PyPy, V8, ...
  $\Rightarrow$ No shared-memory parallelism in a single process

- Fine-grained synchronization: Jython, PyPy-STM, Ruby-HTM
  $\Rightarrow$ Significant overhead on single-threaded performance
  $\Rightarrow$ Sequentialize all writes to the same object/collection

Dynamic Language Implementations

- Global Lock: CRuby, CPython, PyPy, V8, ...
  $\Rightarrow$ No shared-memory parallelism in a single process

- Fine-grained synchronization: Jython, PyPy-STM, Ruby-HTM
  $\Rightarrow$ Significant overhead on single-threaded performance
  $\Rightarrow$ Sequentialize all writes to the same object/collection

- Unsafe: JRuby, Rubinius, Nashorn
  $\Rightarrow$ Break basic thread-safety like reading/writing to an object or operations on built-in arrays/dictionaries

```
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, on the JVM with parallel threads:

```
jruby append.rb
64324
```

Appending concurrently

CRuby, the reference implementation with a Global Lock:

```
ruby append.rb
100000
```

JRuby, on the JVM with parallel threads:

```
jruby append.rb
64324

# or
ConcurrencyError: Detected invalid array contents due to
      unsynchronized modifications with concurrent users
     << at org/jruby/RubyArray.java:1256
  block at append.rb:8
```

A workaround

```
array = []
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
```

Summary

State-of-the-art implementations either

- sequentialize important part of the execution or
- violate basic thread-safety guarantees

We need thread-safe and efficient data representations which:

- provide thread-safety guarantees
- have no overhead on single-threaded execution
- enable parallel workloads to scale

Challenges

- How to provide thread-safety guarantees with no single-threaded overhead?
- How to provide efficent and thread-safe objects to which fields can be added or removed dynamically?
- How to provide efficent, thread-safe and scalable versatile collections?

Scientific Contributions

- A method to automatically detect which objects and collections to synchronize based on reachability, with the only overhead being a write barrier on shared objects and collections
- A thread-safe object model for dynamic languages, synchronizing only on shared objects writes
- A method to gradually synchronize built-in collections in dynamic languages, achieving scalable and thread-safe access to these collections
- Guest-Language Safepoints, a synchronization mechanism that enables interrupting any thread to execute arbitrary code.

Supporting Publications

- A Powerful Synchronization Mechanism
  Techniques and Applications for Guest-Language Safepoints. B. Daloze, C. Seaton, D. Bonetta, H. Mössenböck, ICOOOLPS 2015.

- A Thread-Safe Object Model
  Efficient and Thread-Safe Objects for Dynamically-Typed Languages. B. Daloze, S. Marr, D. Bonetta, H. Mössenböck, OOPSLA 2016.

- Thread-Safe and Scalable Built-in Collections
  Parallelization of Dynamic Languages: Synchronizing Built-in Collections. B. Daloze, A. Tal, S. Marr, H. Mössenböck, E. Petrank, OOPSLA 2018.

Additional Publications

- *Cross-Language Compiler Benchmarking: Are We Fast Yet?*
  S. Marr, B. Daloze, H. Mössenböck, DLS 2016.
- *Few Versatile vs. Many Specialized Collections: How to Design a Collection Library for Exploratory Programming?*
  S. Marr, B. Daloze, PX/18.
- *Specializing Ropes for Ruby*
  K. Menard, C. Seaton, B. Daloze, ManLang'18.

Outline

Single-Threaded Performance and Thread-Safe Objects

Thread-Safe and Scalable Collections

Conclusion

Outline

Single-Threaded Performance and Thread-Safe Objects

Thread-Safe and Scalable Collections

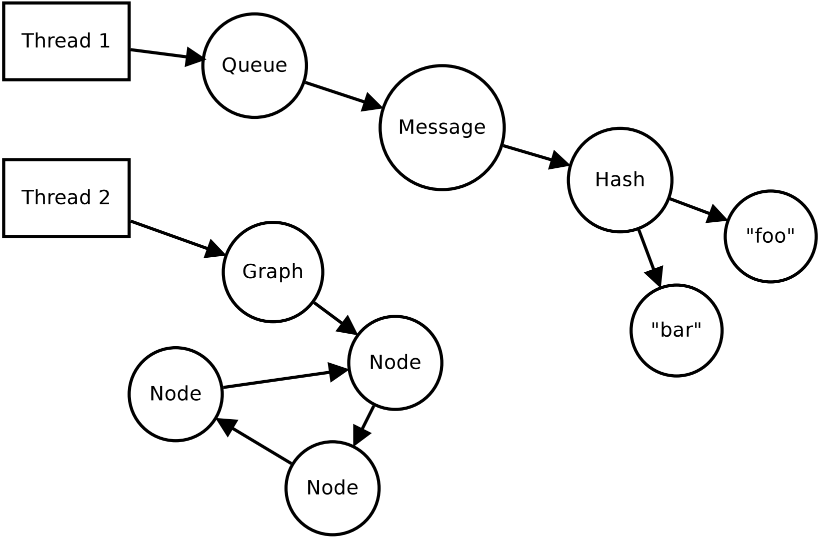
Conclusion

Idea: Distinguishing Local and Shared

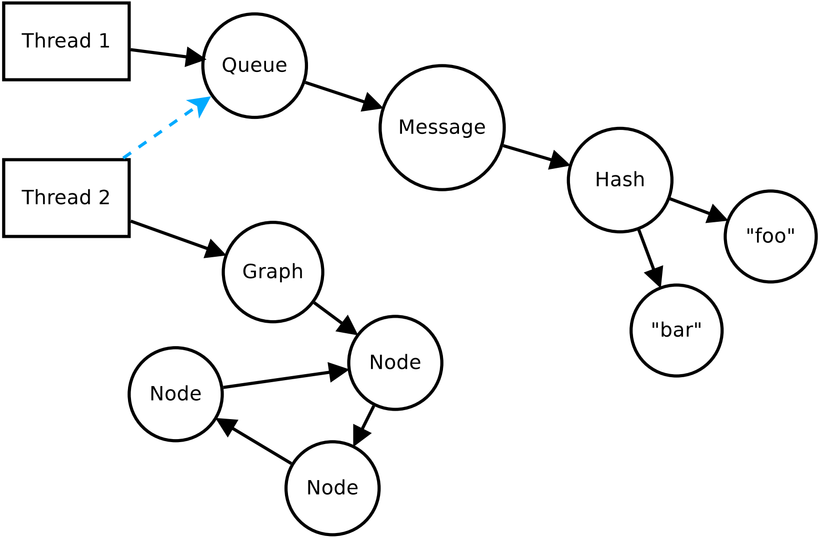
Idea: Only synchronize objects which need it:

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

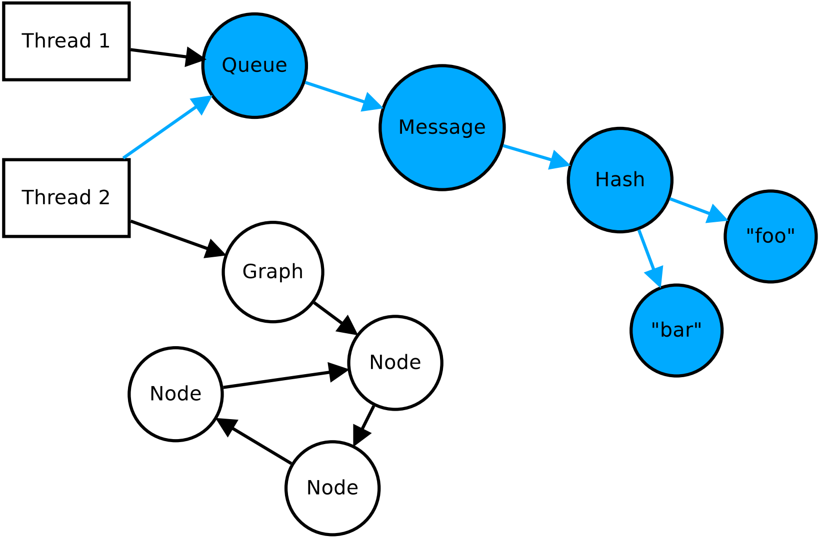
Local and Shared Objects: Reachability



Local and Shared Objects: Reachability



Local and Shared Objects: Reachability



Tracking the set of shared objects

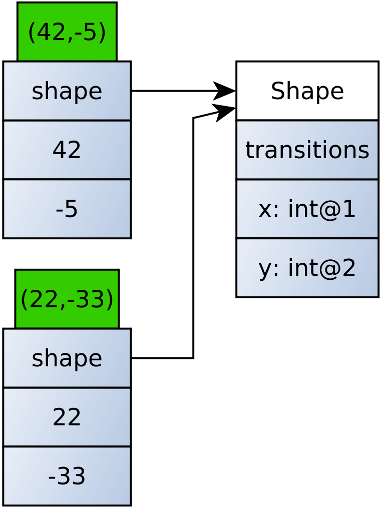
- All globally-reachable objects are *shared* when a second thread is created
- Write to shared object $\Longrightarrow$ share value, transitively

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

No Overhead?

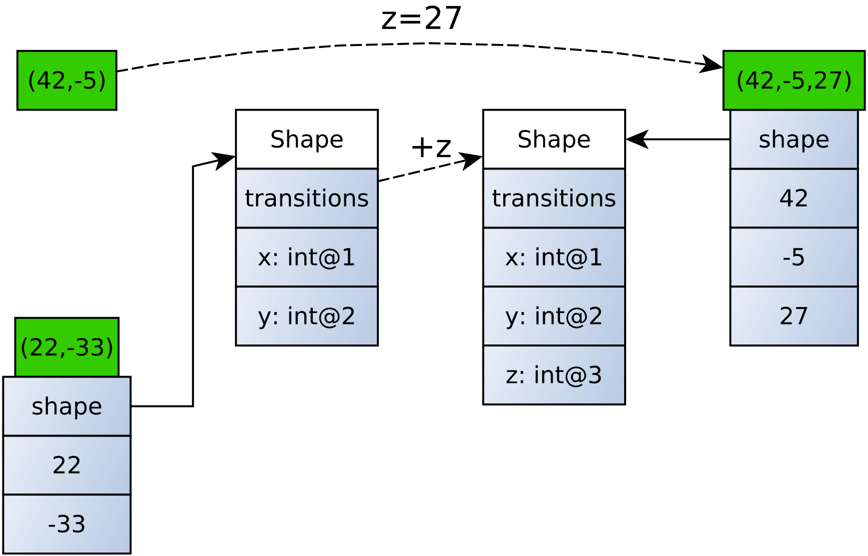
Pseudo code to write a value to a field of an object (`obj.x = 42`):

```
void write(Object object, String field, Object value) {
  if (isShared(object)) {
    // use synchronization
  } else {
    // direct access
  }
}
```



An Object Storage Model for the Truffle Language Implementation Framework
A. Wöß, C. Wirth, D. Bonetta, C. Seaton, C. Humer & H. Mössenböck, 2014.

The Truffle Object Storage Model



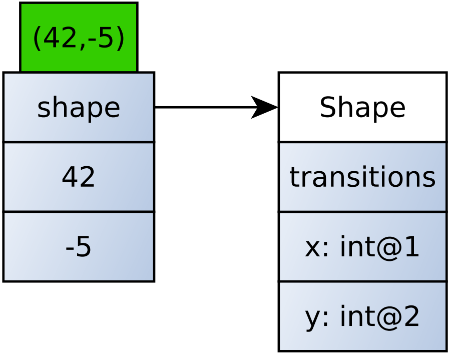
An Object Storage Model for the Truffle Language Implementation Framework
A. Wöß, C. Wirth, D. Bonetta, C. Seaton, C. Humer & H. Mössenböck, 2014.

Updating an existing field with Shapes

Updating the value of an existing field of an object (`obj.x = 42`):

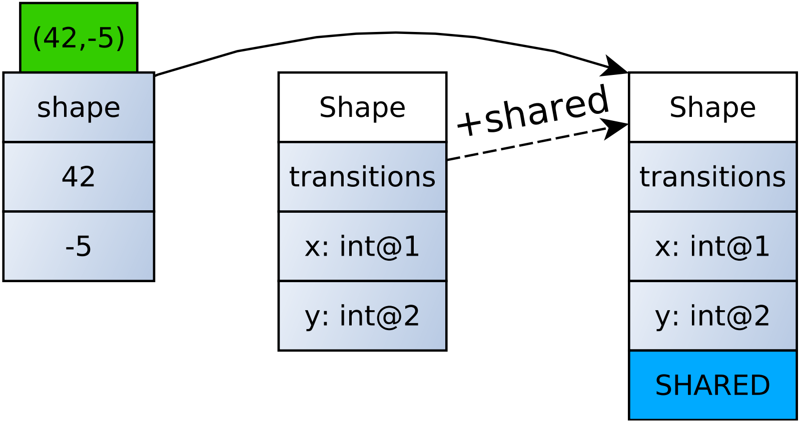
```
void write(Object object, String field, Object value) {
  if (object.shape == CACHED_SHAPE) {
    object[CACHED_OFFSET] = value;
  } else {
    transferToInterpreterAndInvalidate(); // deoptimize
    CACHED_SHAPE = object.shape;
    CACHED_OFFSET = CACHED_SHAPE.getOffset(field);
    write(object, field, value);
  }
}
```

Tracking Sharing in the Shape



Efficient and Thread-Safe Objects for Dynamically-Typed Languages
B. Daloze, S. Marr, D. Bonetta, H. Mössenböck, OOPSLA 2016.

Tracking Sharing in the Shape



Shapes are checked for every field access and method call
$\Rightarrow$ No cost to know if an object is shared

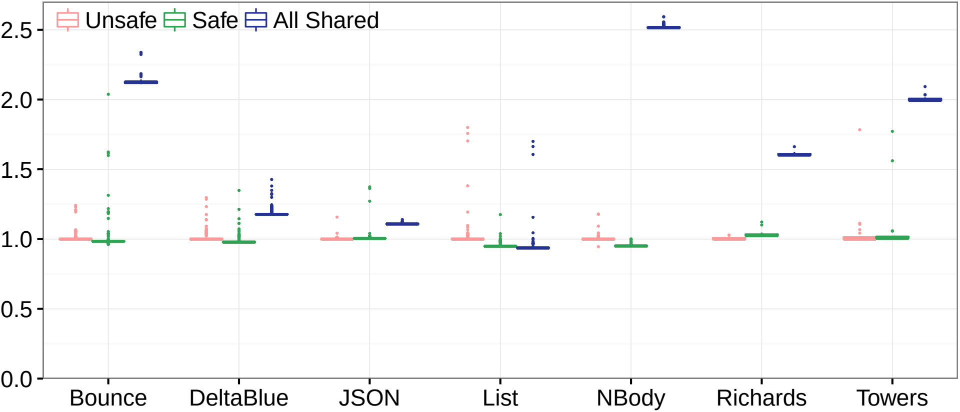
Updating an existing field with Shapes

Updating the value of an existing field of an object (`obj.x = 42`):

```
void write(Object object, String field, Object value) {
  if (object.shape == CACHED_SHAPE) {
    if (SHARED_SHAPE) {
      // use synchronization
    } else {
      object[CACHED_OFFSET] = value;
    }
  } else {
    transferToInterpreterAndInvalidate(); // deoptimize
    CACHED_SHAPE = object.shape;
    CACHED_OFFSET = CACHED_SHAPE.getOffset(field);
    SHARED_SHAPE = CACHED_SHAPE.isSharedShape();
  }
}
```

Single-Threaded Performance for Objects

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



All Shared synchronizes on all object field writes.

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

Extending Sharing to Collections

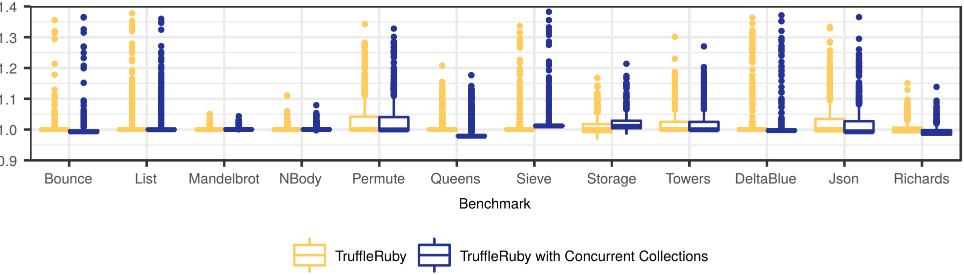
- We need to synchronize on collections too, e.g., to avoid append races
- Collections are objects, they can track sharing in the Shape too
- Shared collections use a write barrier when adding an element to the collection

```ruby
shared_array[3] = Object.new
shared_hash["foo"] = "bar"
```

- Collections can change their representation when shared

Single-Threaded Performance for Collections

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



No difference because these benchmarks do not use shared collections.

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

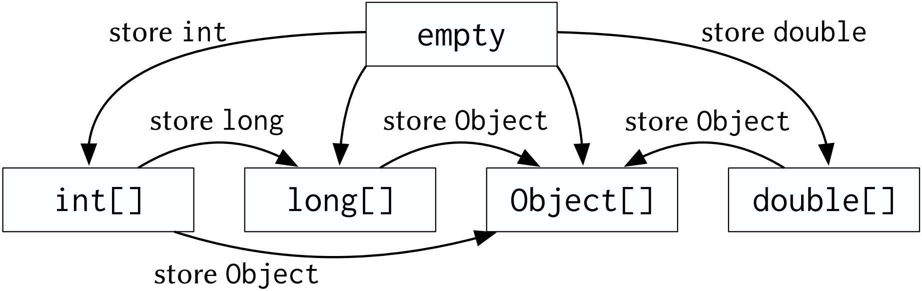
Focus on Built-in Collections

- Built-in collections are the most used (array/list, map, set)
- Built-in collections are thread-safe with a global lock
  $\Rightarrow$ we want to preserve this thread-safety
- User-defined collections are unknown to language implementations, so they cannot guarantee thread-safety

- Array (a stack, a queue, a deque, set-like operations)
- Hash (compare keys by `#hash + #eql?` or by identity, maintains insertion order)

That's all!

Array storage strategies



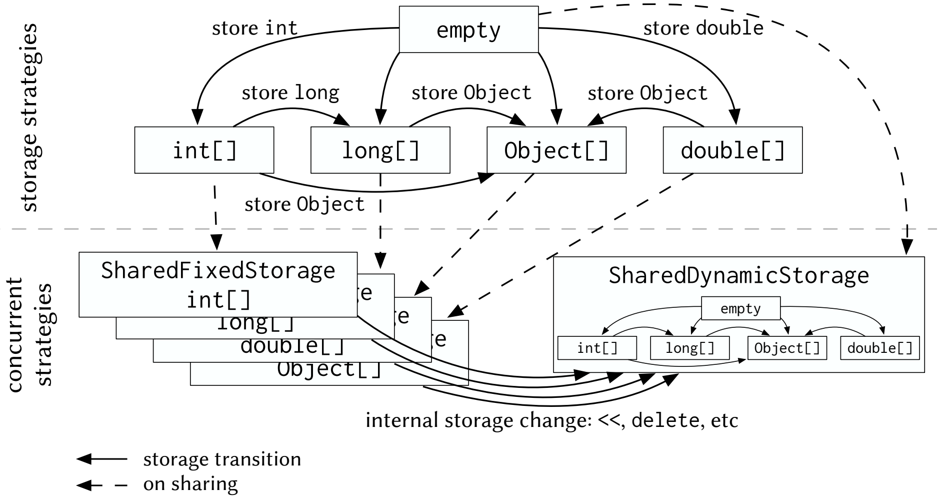

```
class RubyArray {
  // null, int[], long[],
  // double[] or Object[]
  Object storage;
}
```

```
                   storage:
array = []         # empty
array << 1         # int[]
array << 2**42     # long[]
array << "foo"     # Object[]
```

Storage Strategies for Collections in Dynamically Typed Languages

C.F. Bolz, L. Diekmann & L. Tratt, OOPSLA 2013.

Concurrent Array Strategies



SharedFixedStorage

- Assumes the storage (e.g. `int[16]`) does not need to change
  $\Rightarrow$ Array size and type of the elements fits the storage
- If so, the Array can be accessed without any synchronization, in parallel and without any overhead (except the write barrier)

Migrating to SharedDynamicStorage

- What if we need to change the storage?

```
$array = [1, 2, 3] # SharedFixedStorage(int[3])
# Migrate to SharedDynamicStorage
$array[1] = Object.new
$array << 4
$array.delete_at(1)
```

- We use a Guest-Language Safepoint to migrate to SharedDynamicStorage. Once all threads reach the safepoint, we change the strategy to SharedDynamicStorage.

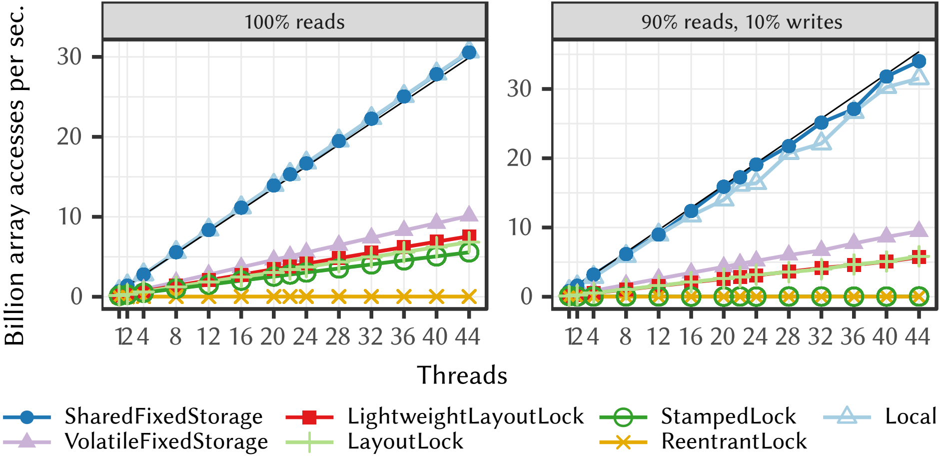
Techniques and Applications for Guest-Language Safepoints
B. Daloze, C. Seaton, D. Bonetta, H. Mössenböck, ICOOOLPS 2015.

- SharedDynamicStorage uses a lock to synchronize operations
- To keep scalability when writing on different parts of the Array, an exclusive lock or a read-write lock is not enough
- We use a Layout Lock

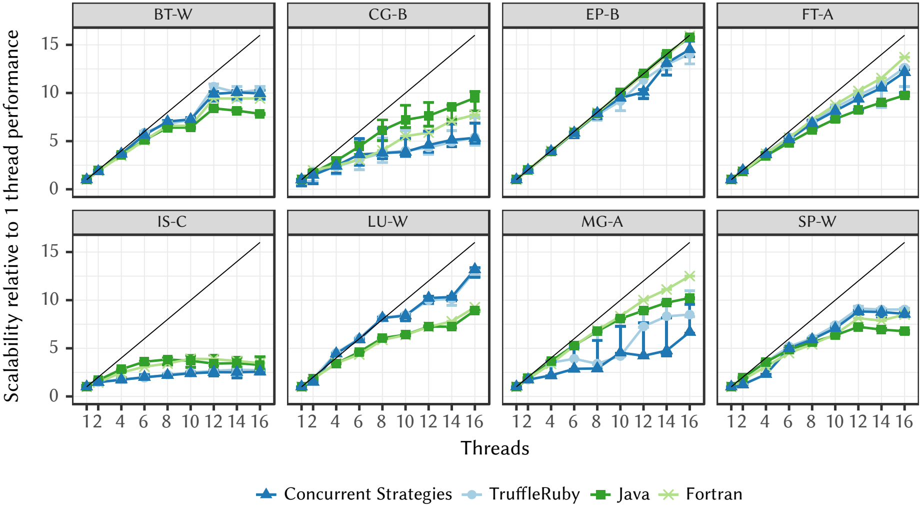
Layout Lock

- 3 access modes: reads, writes and layout changes (storage changes, such as `int[]` to `Object[]`)
- Enables parallel reads and writes
- Layout changes execute exclusively and block reads and writes

Scalability of Array Reads and Writes



NAS Parallel Benchmarks



Single-Threaded Performance and Thread-Safe Objects

Thread-Safe and Scalable Collections

Conclusion

Challenges and Solutions

- How to provide thread-safety guarantees with no single-threaded overhead?
  - By tracking reachability of objects and collections, and only synchronize on shared objects and collections
- How to provide efficent and thread-safe objects to which fields can be added or removed dynamically?
  - By extending Self maps with an extra "shared" field and only synchronizing for shared object writes
- How to provide efficent, thread-safe and scalable versatile collections?
  - By using reachability, a new lock, and dynamically changing the representation based on which operations are used.

Limitations and Future Work

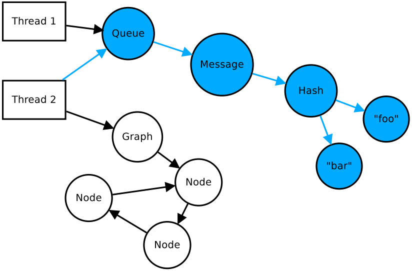
- Shared object field writes are serialized on a given object
- Guest-Language Safepoints currently deoptimize and cause recompilation
- We evaluated for dynamic languages, but some ideas apply to statically-typed languages too
- Explore if other thread-safety guarantees provided by the GIL are useful for users

Conclusion

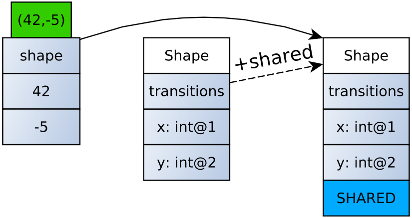
- Synchronizing dynamically based on reachability is a good way to avoid overhead on single-threaded performance
- Objects and built-in collections in dynamic languages can be made thread-safe, efficient and scalable
- We enable parallel programming for dynamic languages, using the existing objects and built-in collections

Summary

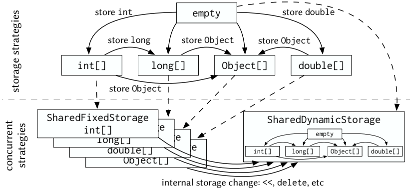
Tracking reachability



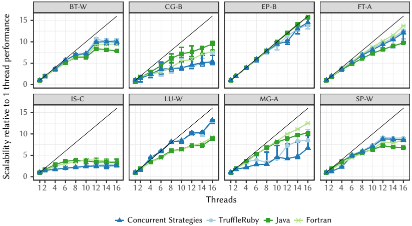
Thread-Safe Efficient Object Model



Thread-Safe Scalable Collections



Scalable Dynamic Languages



Thread-Safe and Efficient Data Representations in Dynamically-Typed Languages

Benoit Daloze

Supervisor	Prof. Hanspeter Mössenböck
Second Examiner	Dr. Jeremy Singer
Third Examiner	Prof. Gabriele Anderst-Kotsis
Präses	Prof. Armin Biere

JOHANNES KEPLER
UNIVERSITÄT LINZ

15 November 2019

Scientific Contributions Summary

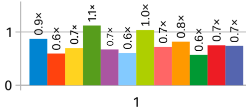
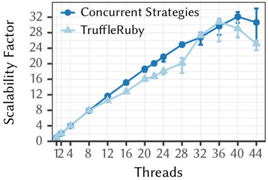
- Guest-Language Safepoints
- A general approach for efficient synchronization based on reachability
- A thread-safe and efficient object model for dynamic languages
- A thread-safe and scalable design for collections in dynamic languages

Together, they enable shared-memory dynamically-typed languages to run in parallel with thread-safe and efficient data representations

Scientific Contributions

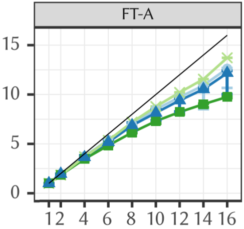
- A method to automatically detect which objects and collections to synchronize based on reachability, with the only overhead being a write barrier on shared objects and collections
- A thread-safe object model for dynamic languages, synchronizing only on shared objects writes
- A method to gradually synchronize built-in collections in dynamic languages, achieving scalable and thread-safe access to these collections
- Guest-Language Safepoints, a synchronization mechanism that enables interrupting any thread to execute arbitrary code.

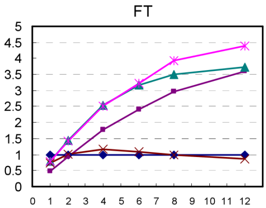
Comparison to PyPy-STM (Meier et al., 2018)

	This thesis	PyPy-STM
Thread-safety guarantees	Thread-safe objects and collections	Sequential consistency
Sequential performance	Identical for 1 thread benchs. 5% geom.avg. on multithreaded benchs	30% slower than GIL (geom.avg. of 10 benchs) 
Parallel speedup	8x faster on 8 threads (20x faster on 22 threads) for PyPy-STM mandelbrot 	1.5x - 6.9x (2.46x geom. avg.) faster on 8 threads than GIL 1 thread

Comparison to Ruby-HTM (Odaira et al., 2014)

	This thesis	Ruby-HTM
Thread-safety guarantees	Thread-safe objects and collections	Sequential consistency
Sequential performance	Identical for 1 thread benchs. 5% geom.avg. on multithreaded benchs	$\geq$ 25% slower than GIL on each NPB benchmark
Max parallel speedup on NPB FT benchmark	9x faster on 12 threads	4.5x faster on 12 threads than GIL 1 thread





Thread Safety Requirements (1-4)

Example	GIL	Goal	Unsafe
Initial: `array = [0, 0]` `array[0] = 1` | `array[1] = 2` Result: `print array`	`[1, 2]`	`[1, 2]`	`[1, 2]`
Initial: `array = [0, 0]` `array[0] = "s"` | `array[1] = 2` Result: `print array`	`["s", 2]`	`["s", 2]`	`["s", 2]` `["s", 0]`

Initial: `array = []`
`array << 1` | `array << 2`
Result: `print array`
	`[1, 2]`
	`[2, 1]`
		`[1, 2]`
		`[2, 1]`
			`[1, 2]`
			`[2, 1]`
			`[1]/[2]`
			exception

Initial: `hash = {}`
`hash[:a] = 1` | `hash[:b] = 2`
Result: `print hash`
	`{a:1, b:2}`
	`{b:2, a:1}`
		`{a:1, b:2}`
		`{b:2, a:1}`
			`{a:1, b:2}`
			`{b:2, a:1}`
			`{a:1}/{b:2}`
			`{a:2}/{b:1}`
			exception

Thread Safety Requirements (5-7)

<table>
<tr><th colspan="2">Example</th><th>GIL</th><th>Goal</th><th>Unsafe</th></tr>
<tr><td colspan="2">Initial: <code>a = [0, 0]; result = -1</code></td><td rowspan="3">1</td><td rowspan="3">1
0</td><td rowspan="3">1
0</td></tr>
<tr><td><code>a[0] = 1</code>
<code>a[1] = 2</code></td><td><code>wait() until a[1] == 2</code>
<code>result = a[0]</code></td></tr>
<tr><td colspan="2">Result: <code>print result</code></td></tr>
<tr><td colspan="2"><code>key = Object.new; h = {key => 0}</code></td><td rowspan="3">2
1</td><td rowspan="3">2
1</td><td rowspan="3">2
1</td></tr>
<tr><td><code>h[key] += 1</code></td><td><code>h[key] += 1</code></td></tr>
<tr><td colspan="2">Result: <code>print h[key]</code></td></tr>
<tr><td colspan="2">Initial: <code>array = [1]</code></td><td rowspan="2">1
2</td><td rowspan="2">1
2</td><td rowspan="2">1
2
0</td></tr>
<tr><td><code>array = [2]</code></td><td><code>print array[0]</code></td></tr>
</table>