University of Kent

Kent Academic Repository

Daloze, Benoit, Tal, Arie, Marr, Stefan, Mössenböck, Hanspeter and Petrank, Erez (2018) *Parallelization of Dynamic Languages: Synchronizing Built-in Collections.* Proceedings of the ACM on Programming Languages, 2 (OOPSLA). ISSN 2475-1421.

Downloaded from

https://kar.kent.ac.uk/69156/ The University of Kent's Academic Repository KAR

The version of record is available from

https://doi.org/10.1145/3276478

This document version

Publisher pdf

DOI for this version

Licence for this version

CC BY (Attribution)

Additional information

Versions of research works

Versions of Record

If this version is the version of record, it is the same as the published version available on the publisher's web site. Cite as the published version.

Author Accepted Manuscripts

If this document is identified as the Author Accepted Manuscript it is the version after peer review but before type setting, copy editing or publisher branding. Cite as Surname, Initial. (Year) 'Title of article'. To be published in *Title of Journal* , Volume and issue numbers [peer-reviewed accepted version]. Available at: DOI or URL (Accessed: date).

Enquiries

If you have questions about this document contact ResearchSupport@kent.ac.uk. Please include the URL of the record in KAR. If you believe that your, or a third party's rights have been compromised through this document please see our Take Down policy (available from https://www.kent.ac.uk/guides/kar-the-kent-academic-repository#policies).

Parallelization of Dynamic Languages: Synchronizing Built-in Collections

BENOIT DALOZE, Johannes Kepler University Linz, Austria
ARIE TAL, Technion, Israel
STEFAN MARR, University of Kent, United Kingdom
HANSPETER MÖSSENBÖCK, Johannes Kepler University Linz, Austria
EREZ PETRANK, Technion, Israel

Dynamic programming languages such as Python and Ruby are widely used, and much effort is spent on making them efficient. One substantial research effort in this direction is the enabling of parallel code execution. While there has been significant progress, making dynamic collections efficient, scalable, and thread-safe is an open issue. Typical programs in dynamic languages use few but versatile collection types. Such collections are an important ingredient of dynamic environments, but are difficult to make safe, efficient, and scalable.

In this paper, we propose an approach for efficient and concurrent collections by gradually increasing synchronization levels according to the dynamic needs of each collection instance. Collections reachable only by a single thread have no synchronization, arrays accessed in bounds have minimal synchronization, and for the general case, we adopt the Layout Lock paradigm and extend its design with a lightweight version that fits the setting of dynamic languages. We apply our approach to Ruby's *Array* and *Hash* collections. Our experiments show that our approach has no overhead on single-threaded benchmarks, scales linearly for Array and Hash accesses, achieves the same scalability as Fortran and Java for classic parallel algorithms, and scales better than other Ruby implementations on Ruby workloads.

CCS Concepts: • **Software and its engineering → Data types and structures**; **Concurrent programming structures**; *Dynamic compilers*;

Additional Key Words and Phrases: Dynamically-typed languages, Collections, Concurrency, Ruby, Truffle, Graal, Thread Safety

ACM Reference Format:
Benoit Daloze, Arie Tal, Stefan Marr, Hanspeter Mössenböck, and Erez Petrank. 2018. Parallelization of Dynamic Languages: Synchronizing Built-in Collections. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 108 (November 2018), 34 pages. https://doi.org/10.1145/3276478

1 INTRODUCTION

Dynamically-typed languages are widely accepted as a tool for convenient and fast software development. Originally, interpreters for dynamic languages in the tradition of Python, Ruby, and JavaScript did not provide high efficiency, but over the years this has improved considerably. There has been a growing body of research improving the efficiency of program execution for these languages [Bolz et al. 2013; Bolz and Tratt 2013; Chambers et al. 1989; Clifford et al. 2015; Daloze

Authors' addresses: Benoit Daloze, SSW, Johannes Kepler University Linz, Austria, benoit.daloze@jku.at; Arie Tal, Computer Science, Technion, Israel, arietal@cs.technion.ac.il; Stefan Marr, School of Computing, University of Kent, United Kingdom, s.marr@kent.ac.uk; Hanspeter Mössenböck, SSW, Johannes Kepler University Linz, Austria, moessenboeck@ssw.jku.at; Erez Petrank, Computer Science, Technion, Israel, erez@cs.technion.ac.il.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2018 Copyright held by the owner/author(s).
2475-1421/2018/11-ART108
https://doi.org/10.1145/3276478

```ruby
array = Array.new()
# append integers to a global array
t1 = Thread.new { array.append(4) }
t2 = Thread.new { array.append(7) }
# wait for threads and print result
t1.join(); t2.join()
print array.size()
```

```
Output:

# On MRI, CPython, Jython, PyPy, etc
2   # completely sequentialized
# On JRuby, Rubinius, TruffleRuby: two possible outputs
ConcurrencyError in RubyArray # low-level error
1   # incorrect result, appends were lost
```

Listing 1. A Ruby example illustrating the state of the art for collections in dynamic language implementations. The state of the art either supports concurrent modification of collections but then sequentializes every operation on the collection, or supports parallel access to the collection but not concurrent modifications.

et al. 2016; Degenbaev et al. 2016; Würthinger et al. 2017; Wöß et al. 2014]. A major opportunity to increase performance is parallelism, allowing programmers to implement parallel algorithms and supporting their execution in the runtime.

Daloze et al. [2016] made significant progress by ensuring that objects can be safely and efficiently represented in multithreaded runtimes. However, one major problem remains unsolved: how can we represent collections safely and efficiently that use optimizations such as storage strategies [Bolz et al. 2013]? To give just one example, in today's dynamic language implementations, it is possible that multiple threads append to an array, and the result is either that all appends are successful, that one gets an error because of a race condition in the underlying implementation, or that appends are lost (cf. Listing 1). In this work, we present a safe, efficient, and scalable approach that can be applied to synchronize a wide range of collections.

As indicated with Listing 1, most dynamic language implementations do not support both concurrent modification of collections and parallel access to the collection. When they do support concurrent modifications, there is an overhead for ensuring thread safety, for instance by degrading single-threaded performance or by not scaling even on workloads that do not require synchronization. JavaScript prevents sharing of arrays[1] and dictionaries between threads. Ruby and Python support sharing, but their standard implementations (MRI, CPython) rely on *global interpreter locks* (GIL) preventing parallel code execution. Jython does not use a GIL but synchronizes every object and collection access [Juneau et al. 2010], which degrades scalability. PyPy-STM [Meier et al. 2016] emulates the GIL semantics and enables scaling but incurs a significant overhead on single-threaded performance. JRuby [Nutter et al. 2018] and Rubinius [Phoenix et al. 2018] aim for scalability but provide no thread safety for built-in collections. Instead, developers are expected to synchronize all accesses to `Array` or `Hash` objects, even for operations that appear to be atomic at the language level. This does not fit with our expectation of what dynamic languages should provide.

In this paper, *we make the next step for parallelism in dynamic languages* by enabling safe, efficient and parallel access to collections. Our new design for parallel collections builds on two contributions: a new gradual synchronization mechanism for collections and a refined version of the Layout Lock [Cohen et al. 2017]. The gradual synchronization mechanism migrates collections between three levels of synchronization, depending on the dynamic needs of each collection instance. The first level avoids synchronization entirely for collections that are accessible only by a single thread. This is done by adopting the approach of Daloze et al. [2016], which distinguishes between local and shared collections. Local collections, which are only reachable by a single thread, do not need synchronization, and therefore their performance is not reduced.

When a collection becomes shared between multiple threads, the second synchronization level is activated. Concurrent algorithms often use fixed-size arrays and rely on parallel read and write

[1]ECMAScript 2018 introduces `SharedArrayBuffer`, which allows sharing primitive data but not objects.

accesses. For such usage patterns, we employ a minimal synchronization strategy that monitors such arrays only for violations of their assumed fixed size and storage.

Finally, for the general case of shared collections, which must support dynamic *layout changes* (e.g., when growing an array to accommodate more elements), we move to the third synchronization level. There, we adopt the Layout Lock, which synchronizes concurrent accesses to the collection with layout changes. The Layout Lock enables parallel accesses to a collection in an efficient and scalable manner when there are no layout changes involved.

However, the original Layout Lock design expects few layout changes and, as a result, is inefficient for frequent layout changes, which happen, e.g., when `append` operations are executed in a loop. Therefore, we designed a new *Lightweight Layout Lock* handling the needs of dynamic language collections by making frequent layout changes efficient and improving the memory footprint.

We implemented our approach by extending TruffleRuby [Seaton et al. 2017], a high-performance Ruby implementation and obtained thread safety for the two main collections in Ruby: `Array` and `Hash`. The gradual synchronization mechanism is executed automatically and transparently, enabling developers to use the language's built-in collections safely in a concurrent setting, without paying a performance or scalability penalty. The thread-safety guarantees are at the level of single collection operations, and are guaranteed even in the presence of data races in the user program. Synchronizing multiple operations remains the user's responsibility.

Finally, we provide an extensive evaluation of the proposed design. Measuring performance of parallel programs for a concurrent dynamic language implementation is tricky, because benchmarks have not yet been written and selected by the community. Another contribution of this paper is a proposal for a set of benchmarks that can be used to measure the performance of parallel dynamic language implementations. We have ported known benchmarks and we have written concurrent programs that make use of collections in parallel. We also created micro-benchmarks to assess specific behaviors of the implementation.

Our experiments show that our approach has no overhead on single-threaded benchmarks, scales linearly for `Array` and `Hash` accesses, achieves the same scalability as Fortran and Java for classic parallel algorithms, and scales better than other Ruby implementations on Ruby workloads.

Based on these results, we hope that memory models for dynamic languages will include built-in collections as an integral part of the memory model to give developers stronger guarantees.

The contributions of this paper are:

- A method of gradual synchronization, migrating collections between multiple levels of synchronization, depending on the dynamic needs of each collection instance, retaining single-threaded performance while providing thread safety and enabling scalability,
- the design and implementation of three *concurrent strategies* for synchronizing two central built-in collections, `Array` and `Hash`, that are common to many dynamic programming languages, of which the main principles are applicable to other collections,
- the Lightweight Layout Lock, which enables better scalability and efficiency than state-of-the-art locks for concurrent strategies that require the highest level of synchronization,
- a benchmark suite consisting of micro-benchmarks, the NAS Parallel Benchmarks [Bailey et al. 1991], and various Ruby workloads,
- an evaluation of the scalability and performance of our solution using the benchmark suite.

2 BACKGROUND

This section reviews the state of the art for concurrency in Ruby, and details work we rely on.

2.1 Concurrency in Ruby

While our proposed techniques are applicable to other dynamic languages and collections, we focus on Ruby to allow for more concrete discussions of issues.

Ruby applications use concurrency widely in production. For instance, the Ruby on Rails web framework [Hansson et al. 2018] uses multithreading by default [Rails Project 2018]. However, the Ruby language does not specify concurrency semantics, so different Ruby implementations provide different thread-safety guarantees. The reference implementation, Matz's Ruby Interpreter (MRI) employs a global interpreter lock (GIL) to protect VM data structures and Ruby's built-in collections (`Array` and `Hash`). The global lock is released during blocking I/O, allowing, e.g., database requests, to run concurrently. However, the GIL completely sequentializes the execution of Ruby code.

JRuby and Rubinius can run Ruby code in parallel, but do not synchronize built-in collections, most likely to avoid a performance overhead. Listing 1 shows that concurrently appending to an `Array` can *throw implementation-level exceptions* with JRuby and Rubinius. Consequently, developers using JRuby or Rubinius need to change their Ruby style to avoid triggering such problems, or need to rely on additional synchronization, which would not be necessary on MRI and therefore introduces additional overhead. Section 7.7 discusses an example we found in the benchmarks. It is just one instance, but we expect many such cases in the wild. Notably, Rubygems and Bundler, the standard tools for tracking dependencies, had multiple bug reports where the tools would throw exceptions caused by the unsafe built-in collections in alternative implementations [Kumar 2013; Shirai 2016, 2018]. Even though Ruby does not specify that collections should be thread-safe, developers rely on such characteristics.

TruffleRuby [Seaton et al. 2017] is a Ruby implementation on the JVM using the Truffle framework [Würthinger et al. 2012] and the Graal compiler [Würthinger et al. 2017]. TruffleRuby's thread-safe objects [Daloze et al. 2016] make reading, updating, adding, and removing fields thread-safe. However, collections are not thread-safe and can exhibit issues similar to JRuby and Rubinius. We address this problem in this paper.

Concurrent Ruby [D'Antonio and Chalupa 2017], an external library, provides thread-safe variants of the `Array` and `Hash` collections. However, this requires the use of a different collection for concurrent code (e.g., `Concurrent::Array.new`), which is error-prone and inconsistent with the notion of dynamic languages having few but versatile collections [Marr and Daloze 2018]. Furthermore, these collections do not scale as they use an exclusive lock per instance.

Therefore, one goal of this work is to make the versatile built-in collections safe, efficient, and scalable, so that they provide the same convenience and semantics developers are used to from MRI, while enabling parallel applications.

2.2 Storage Strategies for Collections

Bolz et al. [2013] proposed storage strategies as an optimization for dynamic languages. They represent homogeneous collections of primitive values such as integers and floats using compact encodings of raw values, thus avoiding extra indirections and boxing or tagging overhead. The *storage strategy* of a collection such as array, set, or dictionary adapts at run time based on the values stored in it. Collection that hold only integers can encode them directly in memory. However, if a collection mixes objects and integers, it falls back to representing integers as boxed values uniformly with the other objects. Since objects and primitives are rarely mixed, the compact representations improves performance and reduces memory overhead [Bolz et al. 2013].

The flexibility and efficiency gains of storage strategies led to their adoption in high-performance dynamic language implementations such as PyPy [Bolz et al. 2009], V8 [Clifford et al. 2015], and TruffleRuby. Unfortunately, storage strategies complicate accessing such collections concurrently.

For example, an array with a storage strategy for integers needs to change to a generic strategy when another thread stores an object in the array. All threads immediately need to use the new storage strategy, which requires complex synchronization and restricts scalability.

2.3 Reachability and Thread-Safe Objects

Daloze et al. [2016] introduced a thread-safe object storage model for dynamic languages. Their solution enables efficient synchronization of accesses to objects that support specialized storage for primitive values [Wöß et al. 2014] and dynamic addition and removal of fields.

For efficiency, they track whether objects are *shared* between threads or *local* to a single thread. This tracking is based on the *reachability* of objects from global roots such as global variables accessible to all threads. Reachability is updated by a write barrier on shared objects. When a local object is written to a shared object, it becomes reachable, and is marked as shared, as well as all local objects that can be reached from it. This mark is part of the object's *shape*, the meta-data describing the fields of an object. Marking as shared means changing the object's shape to an identical shape but with an extra *shared* flag set to true. To maintain correctness of the reachability information, the whole object graph that becomes reachable is traversed. For the common case of small object graphs, Daloze et al. used write barriers that specialized themselves and minimize the overhead.

Using shapes to distinguish between shared and local objects enables Daloze et al. to reuse existing shape checks, without adding run-time overhead. For example, shape checks done for method calls and object field accesses determine also whether an object is shared without additional cost. In theory, this doubles the maximum number of shapes in a system, and could thereby increase the polymorphism of method calls. However, in practice, they did not observe such negative effects.

Local objects do not need any synchronization, which means there is no overhead until an object becomes shared. Shared objects require synchronization on all write operations, and on operations that add or remove fields from the object. Their approach uses a monitor per shared object, so all write operations on the same object are serialized. This is undesirable for collections, where concurrent write operations are common and parallel execution is required to achieve scalability.

2.4 Layout Lock

Cohen et al. [2017] proposed the Layout Lock, a synchronization paradigm and scalable locking algorithm that distinguishes between three types of accesses to a data structure: *read* accesses, *write* accesses, and *layout changes*. Layout changes are complex modifications of the data structure that require exclusive access, such as reallocating the internal storage to accommodate more elements [De Wael et al. 2015; Xu 2013]. The Layout Lock design strongly depends on the assumption that read operations are very frequent, write operations are frequent and layout changes are *rare*. As a result, the Layout Lock is optimized for concurrent read and write operations, but not for layout change operations which carry a high synchronization overhead.

The low-overhead and scalability of read and write operations with the Layout Lock are highly desirable for built-in collections. Unfortunately, for standard collections such as arrays or dictionaries in dynamic languages, we cannot make the assumption that layout changes are rare, since they might happen much more frequently. For instance in Ruby, it is common to create an empty array from a literal `[]` expression[2] and then append elements to it, or even use it as a stack.

[2]In the Ruby standard library alone, there are more than 800 empty literals.

Table 1. Semantics of GIL-based and unsafe implementations showing possible outcomes, and our goals.

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

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

4	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

5	Initial: `a = [0, 0]; result = -1`
	`a[0] = 1`
	`a[1] = 2`
	|
	`wait() until a[1] == 2`
	`result = a[0]`
	Result: `print result`
		`1`	`1`
			`0`
				`1`
				`0`

6	`key = Object.new; h = {key => 0}`
	`h[key] += 1` | `h[key] += 1`
	Result: `print h[key]`
		`2`
		`1`
			`2`
			`1`
				`2`
				`1`

3 THREAD SAFETY REQUIREMENTS

The semantics of GIL-based and unsafe implementations can differ significantly. This section uses examples to illustrate the semantics of representative operations. Based on these, we define the goals for this paper and then discuss the requirements to be fulfilled.

Desired Semantic Intuition. Table 1 gives six examples to guide the discussion. While such a list is inherently incomplete, we included representative cases. Each example starts with an initial state on the first line, and then two threads run the two actions below in parallel. The last line specifies how the result is obtained. The columns *GIL* and *Unsafe* shows the possible outcomes. *Goal* represents the semantics that can be achieved with our approach.

Example 1 updates distinct indices of an `Array` in parallel, with the same type of elements (integers). This has the expected outcome for all categories since all implementations only need to update two separate memory locations and then read those locations.

In Example 2, the first thread writes a `String` (i.e., an `Object`, not an integer), while the second writes an integer. *Unsafe* implementations can lose updates, because of storage strategies (cf. Section 2.2). When performing `array[0] = "s"`, the storage array is migrated from an `int[]` to an `Object[]` storage. This requires copying values from the `int[]` storage. If the second thread writes the integer after the copy but before the new `Object[]` storage is used, then that update is lost. Since there are no races at the application level, our goal is to ensure that such updates are not lost.

In Example 3, the two threads appends concurrently to an `Array`, similar to Listing 1. These appends resize the underlying storage if it is not large enough. While the ordering of the elements is non-deterministic, the *GIL* guarantees both elements to be in the `Array`. However, *Unsafe* can lose one of the appends due to resizing, or it may throw an index-out-of-bounds exception, when

another thread sets a too small storage concurrently. On the application level, such implementation details should not be visible, and thus, lost updates and exceptions should not happen.

Example 4 adds two entries to a dictionary concurrently. Note that dictionaries in dynamic languages typically maintain insertion order. Similarly to Example 3, adding entries can cause resizing of the underlying storage. Therefore, *Unsafe* can also lose updates or observe out-of-bounds exceptions. Furthermore, it is possible to observe *out-of-thin-air* values, such as key `:a` mapped to `2` even though the application never writes `2` to key `:a`. This can happen when the `Hash` representation uses a single array for storing both keys and values (Section 4.5 uses such a representation). The two threads can race and both decide to use `storage[0]` (key) and `storage[1]` (value) for the new entry (since the entry index is not incremented atomically), and we can end up with the key of thread 1 and the value of thread 2. We consider the different dictionary keys as unrelated and the example as race free on the application level. Thus, the goal is to prevent these issues.

Example 5 illustrates racing accesses to disjoint array indexes. Index `0` is used to communicate a value, while index `1` is used to indicate a condition. Under the *GIL*, sequential consistency is guaranteed. *Unsafe* does not give this guarantee since the underlying system may reorder the effects without explicit synchronization. We consider this an application level race and explicit synchronization or high-level mechanisms for communication (e.g., promises) need to be used to ensure correctness. Thus, we allow both possible results of *Unsafe*.

Example 6 shows a typical case of an application-level race condition. Even the *GIL* does not protect against application-level race conditions. Both reads from the `Hash` can happen first, resulting in both threads writing `1` to the `Hash` and losing an increment. A similar race condition happens for `Array` (e.g., concurrent `array[0] += 1`) in most implementations, including CPython.[3]

Requirements. As discussed in Section 1, we aim to enable language implementations to provide thread safety guarantees for built-in collections that are similar to those provided by a GIL, yet without inhibiting scalability. Built-in collections should not expose thread safety issues seen for *Unsafe* in Table 1, which do not exist in the application code. Thus, we want to prevent lost updates, out-of-bounds errors, out-of-thin-air values, and internal exceptions for all built-in collection operations. Such problems expose implementation-level choices that should be invisible to application developers.

Additionally, built-in collections should provide consistency [Harris et al. 2010, p. 24], such that each operation transforms the data structure from one consistent state to another.

We further specify additional guarantees per category of operation: *basic*, *iteration* and *aggregate* operations. *Basic* operations, that is, operations not invoking arbitrary Ruby code and not aggregating multiple operations, should be atomic. With atomic we mean that operations are indivisible and have no observable intermediate state.

Iterators such as `Array#each` and `Hash#each_pair` should be weakly consistent [Peierls et al. 2005], similar to the iterators of the `java.util.concurrent` collections, such as `ConcurrentHashMap`. Weakly consistent means that iterators traverse all elements and tolerate concurrent modifications. Ruby reflects updates during iteration in a single thread and therefore concurrent iterators should reflect concurrent modifications, too.

Aggregate operations such as `hash.merge!(other)` (merging the key-value pairs of `other` to `hash`) should behave as if each step of the aggregate operation is performed individually, such that the keys of `other` are added to `hash` one by one with `hash[key] = value`. This follows the behavior of MRI with a GIL, since the `hash` and `eql?` methods need to be called on the keys, and these methods can be user-supplied and observe the current state of the `Hash`.

[3] MRI seems to make this specific `Array` example atomic, due to not releasing the GIL for intrinsified VM operations. However, simple modifications like calling an identity function around `1`, or using coercion for the index loses the atomicity.

Section 5 addresses these requirements. While our analysis focuses on Ruby, the same issues are present in Python and similar dynamic languages.

4 GRADUAL SYNCHRONIZATION WITH CONCURRENT STRATEGIES

This section presents our main contribution, the method of gradual synchronization using the novel concurrent strategies, which enables scalable and efficient synchronization of built-in collections in dynamic languages. First, we give a brief overview of the key elements of our approach. Then, we describe our gradual synchronization and the concurrent strategies for arrays and dictionaries. We focus on these two collections, because they are the most widely used [Costa et al. 2017].

4.1 Overview of Key Elements for Concurrent Strategies

Before detailing our approach, its key elements and implementation techniques are

- gradual synchronization, i.e., no synchronization for local collections, and choosing synchronization strategies based on usage scenarios for shared collections;
- local and shared collections are distinguished by shape, avoiding introducing new checks;
- correctness of collection operations relies on safepoints and strategy checks already performed by storage strategies;
- the *Lightweight Layout Lock* enables scalable synchronization, by enabling parallel read and write accesses and only sequentializing execution for layout changes.

The remainder of this section and Section 5 explain how concurrent strategies are realized based on these key ideas.

4.2 Tracking Reachability for Collections

Parallel algorithms aiming for good performance typically use local data structures for some part of the work, and only work with shared data structures when necessary [Herlihy and Shavit 2011]. Based on this observation, we adopt the idea of tracking reachability of objects by Daloze et al. [2016] (cf. Section 2.3) and apply it to track whether collections are shared between multiple threads or only accessible by a single thread. This enables parallel algorithms to take advantage of local collections without incurring any synchronization overhead.

In most dynamic languages, collections are also objects. This means that we can track whether they are shared in the same way we track sharing of other objects, namely in their *shape*. When a collection becomes shared, all its elements are marked as shared, as well as all local objects that can be reached from these elements. Shared collections also need to use a write barrier when elements are added to them, as these elements become reachable through the collection (see Figure 1). By using *Storage Strategies* for primitive types, such as an `int[]` strategy, we can avoid overhead for sharing elements. We do not need the write barrier for sharing, since the array contains only primitives like integers, which cannot reference objects.

4.3 Changing the Representation on Sharing

Collections can benefit from knowing whether they are local to a thread or shared. Specifically, we can change a collection's representation and implementation when it becomes shared to optimize for concurrent operations. This is safe and does not need synchronization, because this transformation is done while the collection is still local to a thread, before sharing it with other threads.

This change of representation and implementation allows local collections to keep an unsynchronized implementation, while allowing a

```
a = Object.new
array = [a, 1, 3.14]
# shares array and a
$global_variable = array
b = Object.new
# shares the Hash and b
array << { b => 42 }
```

Fig. 1. Sharing a collection and its elements.

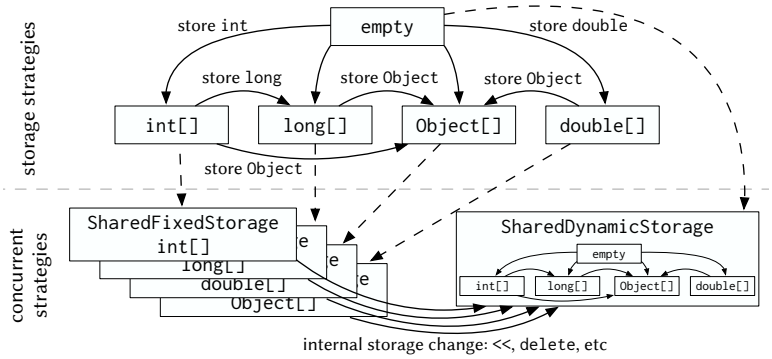

Fig. 2. Strategies for arrays in TruffleRuby. Concurrent strategies are combined with a storage strategy. Plain arrows show storage transitions and dashed arrows represent transitions when sharing an array.

different implementation for shared collections. Therefore, tracking sharing of collections enables us to keep *zero overhead* on local collections, and *gradually synchronize* collections when they become shared.

4.4 Strategies for Array-like Collections

A major challenge for synchronization is that arrays in dynamic languages are not a simple *disciplined* data structure. An `Array` in Ruby can be used as a fixed-size array as in static languages, but it can also be used as a stack, a queue, or a set. Overall, `Array` has more than 100 methods, including operations that insert and remove elements from the middle or replace the entire content, which can even happen concurrently while iterating. As a result, the synchronization mechanism needs to be versatile to adapt to the different supported use cases. While we focus on Ruby, our design applies to other languages as well, e.g., Python has `list` and JavaScript has `Array`.

Storage Strategies. For an optimal data representation and as a foundation for concurrent strategies, TruffleRuby uses five storage strategies for `Array`, illustrated in the top-half of Figure 2. An `empty` strategy is used for empty arrays. If all elements are integers in the value range of `int`, an `int[]` storage is used. If larger integers are required, a `long[]` storage is used. Arrays of floating point numbers use a `double[]` storage. For objects or non-homogenous arrays, an `Object[]` storage is used. To minimize the transitions between strategies, adding an element to the array that cannot be stored with the current strategy causes a migration to a more general strategy, e.g., `Object[]`.

The design of these strategies is based on common ways to create an `Array`. Many arrays start as an empty array literal `[]` for which the `empty` strategy is used. As elements are added, the array is migrated to the most appropriate strategy. Another common constructor is `Array.new(size, initial)` which builds an array with the given `size` with all its elements set to the `initial` value. In such a case, the storage strategy of the array is chosen based on the type of the `initial` value.

Concurrent Strategies. We designed two concurrent strategies for `Array`, adding the concurrency *dimension* to the existing strategies to enable gradual synchronization. Concurrent strategies contain a nested storage strategy to optimize the data representation (see Figure 2). As detailed in Section 4.3, we need to consider concurrency issues only when an array becomes accessible by multiple threads. Thus, right before the array is shared, it changes its strategy to a concurrent strategy. Operations on an array with a concurrent strategy ensure that all accesses are appropriately synchronized.

We anticipate two predominant usage patterns for arrays used concurrently in dynamic languages. On the one hand, we expect classic parallel algorithms to use arrays as fixed-sized abstractions, for

instance in scientific computing. Ruby supports this use case with the mentioned `Array` constructor that initializes the array to a given size. On the other hand, we expect arrays to be used in more dynamic ways, for instance to communicate elements between threads in a consumer/producer style. Hence, we designed two concurrent strategies to optimize for these usage patterns.

SharedFixedStorage. For cases where the array size remains *fixed* and does not need to change, we designed the `SharedFixedStorage` strategy. This is often the case when the array is created with a given size (`Array.new(size)`), or with the `Array#map` method. Thus, this strategy expects that elements will not be added or removed from the array as that would change the size, and that elements stored in the array will be of the same type, e.g., `int` for an `int[]` storage, so that the storage strategy does not need to change. This strategy is designed to have zero overhead over non-shared arrays as it does not need any synchronization. This is the reason we refer to our strategies as *gradual synchronization*. However, if this speculation turns out to be wrong, e.g., when an element of an incompatible type is stored in the array or the size of the array needs to change, the array will migrate to the `SharedDynamicStorage` strategy, which can handle storage strategy changes and all array operations safely, at the expense of some synchronization overhead. But since the array is already shared, the migration must be done carefully so that all threads observe this strategy change atomically. This is detailed in Section 5.2.

SharedDynamicStorage. For the conservative case, where no assumptions are made about how the array is used, we designed the `SharedDynamicStorage` strategy. With this strategy, all operations on the array use synchronization. Various locks could be used for this purpose including exclusive locks or read-write locks. To achieve scalability, the strategy categorizes operations in three types based on whether they correspond to read, write, or layout-changing accesses, as in the Layout Lock design. This is detailed in Section 5.3. Section 7.2 evaluates various locks for this strategy.

Strategy Transitions. When an array becomes shared, the `SharedFixedStorage` strategy is chosen for all non-empty arrays. Empty arrays transition to the `SharedDynamicStorage` strategy. In case elements are added to empty arrays, its storage strategy needs to change. For other uses cases, the additional synchronization has minimal impact since most operations cause out-of-bounds behavior or are no-ops. The transitions to concurrent strategies are depicted in Figure 2 with dashed arrows. The figure shows that `SharedDynamicStorage` adapts its storage strategy dynamically, while the storage strategy is fixed for `SharedFixedStorage`.

4.5 Strategies for Dictionary-like Collections

The second major collection type of dynamic languages is a collection of key-value pairs, often called map or dictionary. JavaScript uses `Map` or generic objects, Python has `dict`, and Ruby has `Hash`. In Ruby, a `Hash` maintains the insertion order of pairs, so that iterating over the keys yields a deterministic order. The hash-code of a key is determined by its `hash` method and keys are compared with the `eql?` method. Thus, a key can be any Ruby object that implements these two methods.

Storage Strategies. TruffleRuby uses three storage strategies to optimize `Hash`, based on how many pairs can be stored. The `Empty` strategy minimizes memory usage of empty dictionaries. The `PackedArray` strategy contains up to three key-value pairs,[4] stored in a single nine-elements `Object[]` array. Each key-value pair is represented as a `(hash, key, value)` triple. The simple structure of `PackedArray` leads to fast access for small dictionaries and enables the compiler to apply for instance escape analysis [Stadler et al. 2014]. The `Buckets` strategy is a traditional hash

[4] TruffleRuby found three to be a good size in benchmarks, because small `Hash` objects are used for keyword arguments.

table implementation using chaining for collisions. This strategy also maintains a doubly-linked list through the entries to preserve insertion order and to enable $O(1)$ delete operations.

ConcurrentBuckets. Similar to `Array`, we devised a concurrent strategy for `Hash`. Before a `Hash` becomes accessible by multiple threads, it is marked as shared and changes its strategy to a `ConcurrentBuckets` strategy. This strategy is similar to the `Buckets` strategy, but uses a different representation and uses synchronization to ensure thread safety (cf. Section 5.4). For simplicity, we use only a single concurrent `Hash` strategy. Concurrent versions of other `Hash` storage strategies would require complex synchronization and do not seem to provide benefits because escape analysis no longer applies for `PackedArray` as the `Hash` already escaped when shared and minimizing memory usage with `Empty` has less impact since few empty `Hash` objects are shared.

4.6 Strategies for Other Built-in Collections

Different languages offer a wide range of different collection types. While we focused on array and dictionary-like collections, our approach to design concurrent strategies as an orthogonal dimension to storage strategies makes it possible to apply them flexibly to other collections. Using strategies that gradually increase synchronization enables optimizing for different use cases and switching at run time based on the observed behavior. However, as seen for dictionaries, it can be sufficient to provide one strategy. Often this is a good tradeoff, because the implementation of concurrent strategies itself can add substantial complexity to a language implementation.

5 REALIZATION

This section details the concurrent strategies for `Array` and `Hash`, how we switch between concurrent strategies, and how we meet the thread safety requirements of Section 3. Furthermore, it describes the novel *Lightweight Layout Lock* and its improvement over the Layout Lock [Cohen et al. 2017].

5.1 The SharedFixedStorage Strategy

The storage for the `SharedFixedStorage` strategy is a Java array. Read and write accesses are performed using plain loads and stores. The Java Memory Model [Pugh et al. 2004, Section 11] specifies that each read and write operation on arrays are themselves atomic, except for `long[]` and `double[]` arrays. In practice, this also holds for `long[]` and `double[]` arrays on 64-bit platforms with HotSpot, which is required to run TruffleRuby. This means it is guaranteed that there are no word tearing effects and out-of-thin-air values, as required in Section 3. However, there are no ordering guarantees between multiple array operations and data races such as stale reads are possible, as shown in example 5 of Table 1. Data races when updating array elements (e.g., two threads concurrently incrementing an element, see example 6) cannot be addressed by the array implementation and an application should use synchronization in such cases.

Since `SharedFixedStorage` only handles read and write accesses, guarantees for other operations are achieved by migrating to `SharedDynamicStorage`.

5.2 Migration to SharedDynamicStorage

The `SharedFixedStorage` strategy speculates that the size and the storage strategy of an `Array` do not change (cf. Section 4.4). If these assumptions are violated, the `Array` migrates to the `SharedDynamicStorage` strategy. We expect these violations to be rare, as detailed in Section 6.1.

Since it is using `SharedFixedStorage`, the `Array` is already shared and the strategy must be changed atomically so that all threads use the new strategy for all further operations on this array. This is achieved with *guest-language safepoints* [Daloze et al. 2015], which suspend all threads executing guest-language (i.e., Ruby) code by ensuring that each thread checks regularly

whether a safepoint has been requested. Guest-language safepoints reuse the existing safepoint mechanism of the JVM and therefore do not introduce additional checks for safepoints in compiled code, but only in the interpreter. Therefore, guest-language safepoints have no overhead on peak performance [Daloze et al. 2015]. The strategy of the `Array` is changed inside the safepoint. Since all threads synchronize together on a barrier at the end of a safepoint, all threads observe this change atomically and use the new strategy for all further operations on the `Array`.

Basic array operations check for safepoints before but not during their execution, thereby guaranteeing that the `Array` strategy cannot change during their execution. Basic means here that they do not call back to Ruby code. `Array` operations that need to call back into arbitrary Ruby code may contain safepoints, which needs to be carefully considered when designing the safety guarantees for such operations. For example, the `Array#each` operation calls a closure for each element. With *storage strategies* (cf. Section 2.2), there is a check for the strategy after each call to Ruby code, since the strategy could have changed. The Ruby callback could for instance have stored an object into an `Array` that had an `int[]` storage strategy. Since these checks are already performed in single-threaded code, *concurrent strategies* do not add extra strategy checks.

5.3 The SharedDynamicStorage Strategy

The `SharedDynamicStorage` strategy safely synchronizes changes to the `Array` storage or size with concurrent read and write operations using a Lightweight Layout Lock (see Section 5.6 below). It enables efficient and scalable read and write operations as well as successive layout change operations (e.g. a loop that appends to an `Array`). Read operations that have no side-effects use the lock's *optimistic reads* mode and retry when the lock detects a concurrent layout change. Write operations acquire the lock's *write* mode, which allows concurrent write operations while preventing layout change operations. Finally, operations that affect the `Array` size and storage acquire the lock in *layout change* mode, blocking all other operations.

Basic operations with the `SharedDynamicStorage` strategy are atomic, as required by Section 3. Basic `Array` operations either use a layout change or access (read or write) the array at an index. Operations using layout changes are atomic as they execute exclusively and block all other operations. Operations calling back to Ruby code behave the same as for `SharedFixedStorage` and check the strategy after each callback (cf. Section 5.2). In addition to that they only perform reads and writes. Since `SharedDynamicStorage` also uses a Java array as storage, read and write operations at an index are atomic as described in Section 5.1.

5.4 The ConcurrentBuckets Strategy

When a `Hash` becomes shared, it changes its strategy to a `ConcurrentBuckets` strategy. This strategy supports parallel lookup, insertion, and removal. This is achieved by using a Lightweight Layout Lock (see Section 5.6) for synchronizing the buckets array and a lock-free doubly-linked list to preserve the insertion order. The Lightweight Layout Lock is used so that *lookups* use optimistic reads, *put* and *delete* are treated as write accesses, and a layout change is used when the buckets array needs to be resized. A `Hash` in Ruby can be switched at any time to the compare-by-identity mode, a feature to compare keys using reference equality instead of calling the `eql?` method. A layout change is used when switching to the compare-by-identity mode, so that all operations on the Hash notice this change atomically and start using reference equality from that point on.

The implementation of the doubly-linked list for preserving the insertion order is inspired by Sundell and Tsigas [2005], which marks pointers to adjacent nodes as unavailable for insertion while a node is being removed. For Java, instead of marking pointers, we use dummy nodes for the duration of the removal. This technique allows appending to the list with just two compare-and-swap (CAS) operations, which might need to be retried if there is contention. Hash collisions are

handled by chaining entries, forming a singly-linked list per bucket, for which we use a lock-free implementation based on Harris [2001].

The four *basic* `Hash` operations are atomic, as required by Section 3. Hash lookups and updates for an existing key respectively read and write the volatile `value` field of the entry, so they are atomic and sequentially-consistent. Adding a new entry first adds the entry in the bucket with a CAS operation, making it appear atomically to lookups, and then appends the entry to the doubly-linked list with a CAS operation on `lastEntry.next`, making iterators notice the new entry atomically. Removing an entry is similar but it is first removed from the doubly-linked list.

5.5 Thread Safety Guarantees

The goal of concurrent strategies is to enable language implementations to provide thread safety guarantees for built-in collections (cf. Section 3). Specifically, we want to ensure that optimized collection implementations do not lead to lost updates, out-of-bounds errors, or out-of-thin-air values as a result of a used implementation approach.

We avoid such errors by using safepoints to atomically switch between `SharedFixedStorage` and `SharedDynamicStorage` (cf. Section 5.2), and by synchronizing operations on `Array` and `Hash` with the Lightweight Layout Lock as part of `SharedDynamicStorage` and `ConcurrentBuckets`. We ensure that operations only see a consistent state by using *layout changes* for any operation that could cause inconsistencies in collections (e.g., $size > storage.length$ for an `Array`).

The previous sections detail how *basic* operations are atomic, from which we build all other operations. Thus, aggregate operations are implemented as calling multiple basic operations and therefore behave like performing each basic operation individually. We also make sure that iterators always make progress: `Array` iterators walk through elements of increasing indices and `Hash` iterators follow the doubly-linked list. Metaprogramming operations as for instance reflective read or write operations are based on the basic operations, which ensures that they are also atomic.

We also considered a variant of `SharedFixedStorage` using volatile accesses on the Java array to provide sequential consistency over all reads and writes. However, we decided against it because of its significant overhead (cf. Section 7.2).

5.6 Lightweight Layout Lock

The Lightweight Layout Lock improves over the Layout Lock (cf. Section 2.4) and we designed it to support our primary contribution of gradual synchronization with concurrent strategies (Section 4). Compared to state-of-the-art locks, it provides better overall scalability and efficiency for the `SharedDynamicStorage` and `ConcurrentBuckets` strategies, as we show in Section 7.

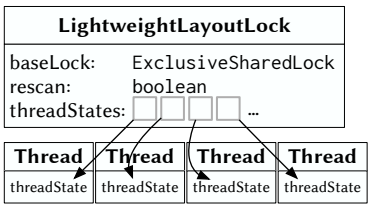

Fig. 3. The Lightweight Layout Lock is composed of a base lock that supports shared versus exclusive accesses, an optimization flag that indicates whether the lock needs to change thread states, and a list of references to the thread states of registered threads.

Structure and Design. First, we describe the overall structure of the lock, illustrated in Figure 3, and its design from a high-level view. The Lightweight Layout Lock uses a *base lock* that supports shared versus exclusive locking. It augments the base lock with a synchronization protocol that is optimized for efficient parallel access to data structures that might need to change their internal representation. As previously mentioned, the lock design distinguishes between read accesses, write accesses, and layout changes. This design allows both reads and writes to be executed

in parallel, and only layout changes are executed exclusively, blocking read and write accesses. To simplify its design, the Lightweight Layout Lock does not allow nesting of lock operations.

Since reads are typically the most common operations, their overhead is minimized by allowing them to be optimistic, which means the data is read first and then validated. If the validation succeeds, there was no layout change concurrent with the read operation and the data is valid. Otherwise, the read operation must be retried as the read data might be invalid.

The lock achieves parallel write accesses by storing a per-thread state for each thread using the lock. Each thread that uses the lock has its own `threadState` per lock, which is represented as an `AtomicInteger`. When starting a write access, the current thread's `threadState` is set to `WR`, a flag to indicate a write access is happening and layout changes must wait until the write operation is finished. This way, a write access only modifies the current thread's `threadState`, but does not contend by modifying shared state, which would hinder scalability of writes as is the case for many state-of-the-art locking mechanisms.

The Lightweight Layout Lock optimizes two common cases in dynamic language collections: (a) concurrent read and write accesses with rare or no layout changes, and (b) sequences of consecutive layout changes that can be initiated by different threads. To optimize consecutive layout changes, an extra boolean `rescan` flag is added. This flag indicates whether there were any read or write access between layout change operations, and is used to facilitate the layout change fast path described below.

Implementation. We now describe the implementation of the lock along with the code shown in Listing 2. We first detail the fast path, i.e., the most common and performance-sensitive cases. Afterwards, we present the less common slow path and the protocol for dynamic thread registration.

Fast Path. The overhead of read operations is minimized by allowing them to be concurrent and optimistic. Therefore, reads on the fast path only validate the correctness of the read data by checking that the thread's `threadState` does not indicate a concurrent layout change (cf. `finishRead()` in Listing 2), i.e. that the `threadState` is 0 (inactive). Note that at `finishRead()`, the `threadState` value can be either 0 or `LC`, since a thread can either read, write, or perform a layout change at a given time, and at `finishWrite()` the thread clears the `WR` bit from its `threadState`.

Write operations are concurrent, and indicate their activity by using a `compareAndSet` (CAS) to change their respective thread's `threadState` from 0 (i.e. inactive) to `WR` at the start (cf. `startWrite()`), which blocks concurrent layout changes *without* needing to lock the `baseLock`. If the CAS fails, the write operation enters its slow path, which is discussed below. Upon completion, write operations clear the `WR` bit using an atomic subtraction (cf. `finishWrite()`), since the `threadState` may have the `LC` bit turned on due to a pending layout change operation, as described below.

A layout change operation locks the `baseLock` in exclusive mode to block concurrent layout changes as well as the read and write slow paths, and then checks the `rescan` flag. If the `rescan` flag is not set, the layout change operation was preceded by a layout change operation without any intervening read or write operation, thus all the `threadStates` still have their `LC` flags set, and the layout change operation may proceed immediately.

Slow Path. In their slow path, read and write operations reset their thread's `threadState` to recover from layout changes. They do so by calling the `slowSet()` method in Listing 2, which locks the `baseLock` in shared mode. Locking the `baseLock` in shared mode allows multiple readers and writers to go through the slow path concurrently, if there is no active layout change operation that holds the `baseLock` in exclusive mode. Otherwise, readers and writers are blocked until the lock is released from exclusive mode. During the call to `slowSet()`, readers and writers also set the `rescan` flag to force a subsequent layout change operation to go through its slow path. When the

```
1  class LightweightLayoutLock {
2    final static int WR = 0x01, LC = 0x02;
3    ExclusiveSharedLock baseLock;
4    boolean rescan = false;
5    List<AtomicInteger> threadStates;
6
7    /* Read operation */
8    void startRead(AtomicInteger threadState) { }
9
10   boolean finishRead(AtomicInteger threadState) {
11     // Prevent data reads to float below
12     Unsafe.loadFence();
13     if (threadState.get() == 0)
14       return true; // no LC so data is valid
15     // set to idle after LC done
16     slowSet(threadState, 0);
17     return false; // signal to retry the read
18   }
19
20   /* Write operation */
21   void startWrite(AtomicInteger threadState) {
22     if (!threadState.compareAndSet(0, WR))
23       // LC happened or pending,
24       // wait for LC to complete
25       slowSet(threadState, WR);
26   }
27
28   void finishWrite(AtomicInteger threadState) {
29     // subtract since LC might be pending
30     threadState.addAndGet(-WR);
31   }
32
33   /* Common slow path for read and write operations */
34   void slowSet(AtomicInteger threadState, int n) {
35     // wait for LC to complete
36     baseLock.lockShared();
37     // subsequent LC must scan thread states
38     rescan = true;
39     threadState.set(n);
40     baseLock.unlockShared();
41   }
42   /* Layout change operation */
43   void startLayoutChange() {
44     // wait for slowSet or other LC
45     baseLock.lockExclusive();
46     if (rescan) {
47       // if write or read slowSet executed
48       for (AtomicInteger threadState : threadStates)
49         if (threadState.get() != LC) {
50           // if not already LC
51           threadState.addAndGet(LC);
52           // wait for current write to complete
53           while (threadState.get() != LC) {}
54         }
55       // rescan only after slowSet
56       rescan = false;
57     }
58   }
59
60   void finishLayoutChange() {
61     // Do not reset threadStates
62     // to optimize consecutive LCs
63     baseLock.unlockExclusive();
64   }
65
66   /* Thread registration/unregistration */
67   void registerThread(AtomicInteger threadState) {
68     threadState.set(LC);  // Initial value
69     baseLock.lockExclusive();
70     threadStates.add(threadState);
71     baseLock.unlockExclusive();
72   }
73
74   void unregisterThread(AtomicInteger threadState) {
75     baseLock.lockExclusive();
76     threadStates.remove(threadState);
77     baseLock.unlockExclusive();
78   }
79 }
```

Listing 2. A Lightweight Layout Lock implementation.

`rescan` flag is set, a layout change operation enters its slow path, where for each `threadState` in the `threadStates` list, it turns on the layout change indicator bit (`LC`) atomically (using `addAndGet`) and busy-waits until the thread's current write operation completes, if needed,[5] before moving on to the next `threadState` in the `threadStates` list. When the `LC` bit has been turned on for a thread's `threadState`, a subsequent `startWrite()` operation by that thread will fail at the `CAS` (since the `threadState` will no longer be 0, i.e., inactive), thereby invoking its slow path. In essence, the `LC` flag also acts as a "pending" indicator for writers, thus avoiding starvation for layout changers.

Dynamic Thread Registration. To acquire an object's Lightweight Layout Lock for read, write, or layout change operations, a thread must first register with the lock by allocating a `threadState` and adding it to the lock's `threadStates` list using the `registerThread()` method in Listing 2. During thread registration, the `baseLock` is held in exclusive mode, to prevent a race between layout changers which iterate the `threadStates` list and registering threads, which modify it. However, concurrent read and write fast paths are not affected by the `baseLock`, and therefore can continue unhindered. A new `threadState` is initially set to `LC` in order to avoid triggering the layout change slow path when the object experiences consecutive layout changes while threads register with its

[5]Recall that a writer resets its `threadState`'s `WR` bit upon completion.

lock. Thus, for example, threads may dynamically register and start appending to a collection via the layout change fast path.

Memory Footprint. Each Lightweight Layout Lock contains a base lock, a `rescan` flag, and a list of thread states. Since the thread states are `AtomicInteger` objects, the `threadStates` list contains references to these objects. Each thread registering with the lock allocates its lock-specific `threadState` for the lock. Thus the amount of memory required for a Lightweight Layout Lock is a function of the number of threads that register with the lock.

Correctness. We sketch the correctness of the Lightweight Layout Lock protocol by describing invariants that are preserved by the lock in the supplementary material [Daloze et al. 2018].

6 DISCUSSION

Safepoints are a crucial element to ensure correctness, but have a potential performance impact, which we discuss in this section. Furthermore, we discuss how the overall design depends on design decisions of the underlying virtual machine.

6.1 The Impact of Safepoints

When the fixed storage assumption fails for an array that uses the `SharedFixedStorage` strategy, it needs to be migrated to the `SharedDynamicStorage` strategy. However, since such an array is reachable by multiple threads, we use guest-language safepoints (cf. Section 5.2) to perform the migration atomically. Safepoints stop all threads during the operation. Thus, if they are too frequent, they could severely degrade a program's scalability and performance.

In parallel programs, we see two dominant usage patterns for shared arrays. We avoid safepoints for both cases. The first is pre-allocating an array with a given size, accessing elements by index, updating elements with values of the same type, and never changing the size of the array. This usage pattern is common for many parallel programs with shared memory, e.g., for image processing or scientific computing, and is similar to using fixed-size arrays in statically-typed languages. In such a case, the `SharedFixedStorage` strategy is used and never needs to be changed. The second usage pattern, observed in idiomatic Ruby code, is one where an empty array is allocated and elements are appended or removed. As detailed in Section 4.4, our heuristic chooses the `SharedDynamicStorage` strategy immediately when an empty array is shared, thus avoiding safepoints.

Using Allocation Site Feedback to avoid Safepoints. Our heuristic avoids frequent safepoints for all programs in our evaluation. However, in other programs arrays may get pre-allocated and then have elements added or removed, or have elements of an incompatible type stored, and thus violate their fixed storage assumption. For such cases, it is possible to extend our heuristic using allocation site feedback such as *mementos* [Clifford et al. 2015], which allows arrays to remember their allocation site. When an array needs to be migrated with a safepoint, it notifies the allocation site. If arrays allocated at a specific site cause frequent safepoints, further arrays allocated at that site will start directly with a `SharedDynamicStorage` strategy. The overhead of `SharedDynamicStorage`'s synchronization should be offset by reducing the much larger cost of frequent safepoints.

6.2 Services Required from the Underlying VM

While the general ideas of concurrent strategies and the Lightweight Layout Lock are independent of a specific implementation technology, we rely on a number of properties for the implementation, which also have an impact on the performance results seen in Section 7.

The Lightweight Layout Lock only relies on a shared-exclusive lock and atomic variables with CAS and `addAndGet()`. The similar Layout Lock by Cohen et al. [2017] was implemented both in

Java and C++. We therefore believe the Lightweight Layout Lock could be ported to other memory models in addition to the JMM.

For concurrent strategies, we rely on more aspects of the underlying virtual machine. Generally, we assume a garbage collected system with shared-memory multithreading. Garbage collection is not strictly necessary, but simplifies the implementation of concurrent data structures. Furthermore, it comes typically with an optimized safepoint mechanism. We rely specifically on the synchronization semantics of safepoints for migrating between concurrent strategies. In other systems, adding a similar mechanism would possibly add run time overhead [Lin et al. 2015]. We also assume that accessing array elements is atomic (cf. Section 5.1).

6.3 Performance Predictability

The wide range of dynamic features offered by dynamic languages makes it hard to predict the performance profile of a program. Much of the unpredictability comes from the various heuristics employed by VMs, which includes storage strategies or in our case concurrent strategies. While some of the dynamic behavior might be genuinely useful to solve complex problems, others might be undesirable and indicate performance issues. For instance, when implementing a classic parallel algorithm (cf. Section 7.6), one would not expect the need to switch between different concurrent strategies for an array. More generally, races on the layouts of data structure can indicate performance bugs. For these kind of issues, we imagine tools similar to the work of Gong et al. [2015] and St-Amour and Guo [2015], which could use the information on safepoints and strategy changes to guide performance optimizations.

7 EVALUATION

This section evaluates the performance of concurrent strategies. We verify that adding concurrent strategies has no impact on the performance of single-threaded code. Furthermore, we evaluate the suitability of our `Array` strategies for different scenarios, assess the cost of switching between strategies, and how the Lightweight Layout Lock compares to alternative locks. For `Hash`, we measure the performance of different locks for the `ConcurrentBuckets` strategy.

Since there is no standardized benchmark suite for Ruby, we collated existing benchmarks for the evaluation, some of which were ported from other languages. With our benchmark suite, we demonstrate the usability of concurrent strategies for an idiomatic parallel Ruby program. We evaluate the scalability of our approach on classic parallel benchmarks including the NAS Parallel Benchmarks [Bailey et al. 1991], a benchmark from PyPy-STM [Meier et al. 2016], and one from JRuby. Furthermore, we use larger Ruby benchmarks to show how our approach works in realistic scenarios.

Methodology. We implemented our three concurrent strategies in TruffleRuby. We designed the `SharedDynamicStorage` strategy to use different locks, which allows us to compare the Layout Lock [Cohen et al. 2017], Java's ReentrantLock and Java's StampedLock with our Lightweight Layout Lock. The benchmarks were executed on a machine with 2 NUMA nodes, each with a 22-core Intel® Xeon® CPU E5-2669 v4 with hyperthreads, operating at 2.20GHz with disabled frequency scaling. The machine has 378GB RAM, 32KB L1 cache and 256KB L2 cache per core, and 55MB L3 cache per NUMA node. We used Oracle Linux 6.8 (kernel version 4.1.12-37.5.1.el6uek), Java HotSpot 1.8.0u141, GNU Fortran 6.1.0, MRI 2.3.3p222 and JRuby 9.1.12.0. We based our work on TruffleRuby revision `5d87573d`. We report median peak performance within the same JVM instance, by removing the first 2 iterations of warmup. Error bars indicate the minimum and maximum measurements. We make sure every benchmark iteration takes at least a second. For micro-benchmarks, each iteration measures the throughput during 5 seconds. We use weak scaling

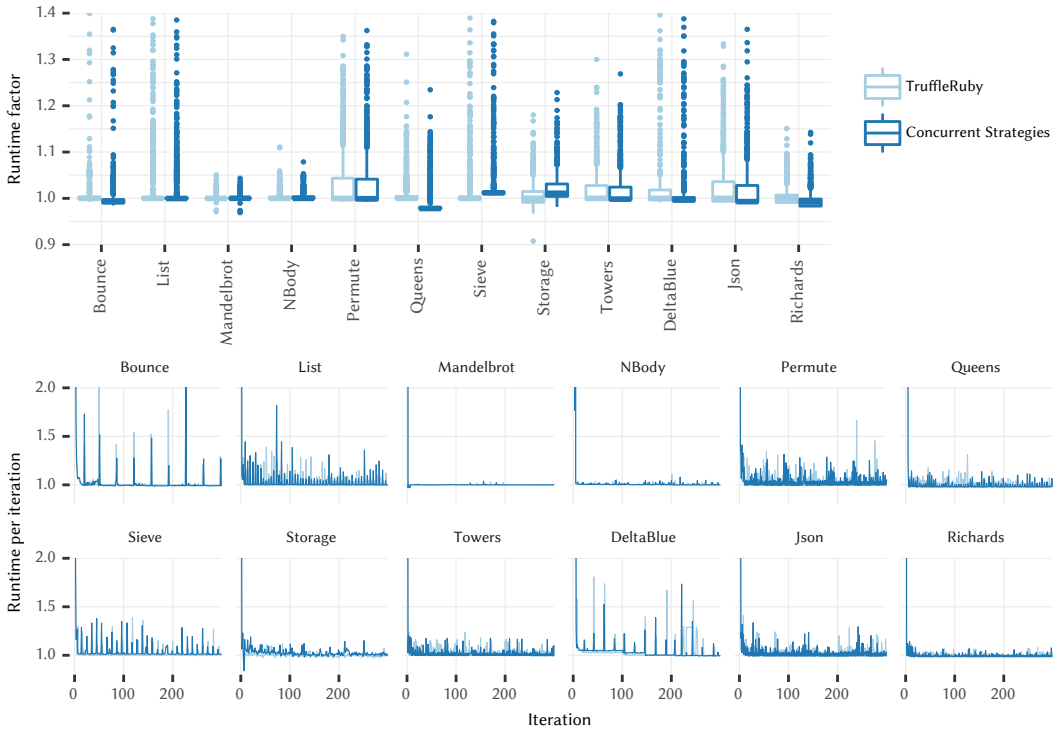

Fig. 4. Performance of `Concurrent Strategies` on 12 single-threaded benchmarks. Results are normalized per benchmark to the median of unmodified `TruffleRuby`. The box plot indicates median and quartiles of the run time factor. On average, performance is identical within measurement errors and outlier behavior is similar. Outliers are run time spikes for instance caused by garbage collection and also visible in the line plots. Small speedups, for instance 2% on Queens, are caused by compiler heuristics triggering slightly differently. The line plot shows the warmup behavior for the first 300 iterations as iteration times per benchmark. Warmup behavior is similar for both `Concurrent Strategies` and unmodified `TruffleRuby`.

for micro-benchmarks, that is we increase the problem size together with the number of threads. We use strong scaling and use a fixed-size workload for other benchmarks.

7.1 Baseline Single-Threaded Performance

We use the *Are We Fast Yet* benchmark suite [Marr et al. 2016], to show that concurrent strategies do not affect single-threaded performance. This suite contains classic single-threaded benchmarks in different languages, notably Ruby. We run 1000 iterations of each benchmark to assess peak performance. Figure 4 shows the results for the unmodified *TruffleRuby* and our version with *Concurrent Strategies*. The performance difference between them is within measurement errors, which confirms that the performance of sequential code is not affected. This is expected since these benchmarks do not write to shared state and only use local collections, even though global constants and class state are shared during startup. The warmup behavior, also shown in Figure 4, is similar for both *TruffleRuby* and *Concurrent Strategies*.

Single-threaded performance remains important as many applications are written in a sequential way, partly due to the poor support for parallel execution from dynamic language implementations. This relevance of single-threaded performance is also reflected by the continuous work on

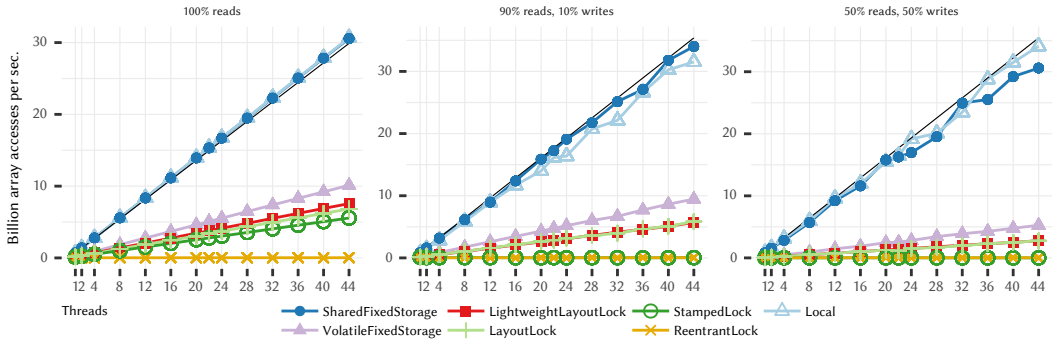

Fig. 5. Throughput scalability of concurrent array accesses. The straight black lines denote ideal scalability for `Local` based on its 1-thread performance. On the last two benchmarks, `LightweightLayoutLock` and `LayoutLock` have the same performance, and `StampedLock` and `ReentrantLock` do not scale. A table with the raw data is available in the supplementary material. [Daloze et al. 2018]

implementations such as TruffleRuby, PyPy, and V8. Marr et al. [2016] showed that TruffleRuby and V8 reach about the same performance on their benchmarks, illustrating that the TruffleRuby single-threaded performance is among the best.

7.2 Micro-Benchmarks for Array Performance

We created a set of micro-benchmarks for array operations to compare the synchronization overhead of our strategies. First, we measure read and write operations that do not cause layout changes to compare `SharedFixedStorage` with `SharedDynamicStorage`, for which we compare different types of locks. These benchmarks sum the array elements to avoid optimizing out the reads. As Figure 5 shows, there is a clear advantage of using `SharedFixedStorage`, because it does not require any synchronization for the array accesses. In fact, its performance is the same as `Local` within measurement errors, which represents unsafe storage strategies. Accessing the array of the fixed storage strategy with volatile semantics (cf. Section 5.5) has a significant overhead (3x to 7.3x) as shown by `VolatileFixedStorage`, which is only slightly faster than locks. The best-performing lock for `SharedDynamicStorage` is the Lightweight Layout Lock, with a relative overhead ranging from 4x to 12.6x over `SharedFixedStorage`. Figure 5 also shows that array accesses scale perfectly (a speedup of n on n threads) with `SharedFixedStorage`, Layout Lock, and Lightweight Layout Lock, while others locks scale poorly with concurrent write operations.

Next, we measure the `SharedDynamicStorage` strategy with concurrent appends. Since appends require the size of the storage to change, the `SharedFixedStorage` strategy is not applicable. Figure 6 shows that concurrent appends do not scale well, because the threads contend on the lock and write to adjacent memory. In contrast to the previous benchmarks, `ReentrantLock` and `StampedLock` perform relatively well, and `LayoutLock` performs poorly. Because of our optimization for layout changes, the Lightweight Layout Lock performs well in both cases.

7.3 Impact of Global Safepoints on Performance

To evaluate the cost of guest-language safepoints used for migrating an array safely from the `SharedFixedStorage` strategy to the `SharedDynamicStorage` strategy (cf. Section 6.1), we use a throughput benchmark that causes the migration of 1000 arrays every 5 seconds (by removing the last element of each array) and then reading and summing elements from each array during the remaining time. We compare this benchmark with a variant in which arrays *start* in the

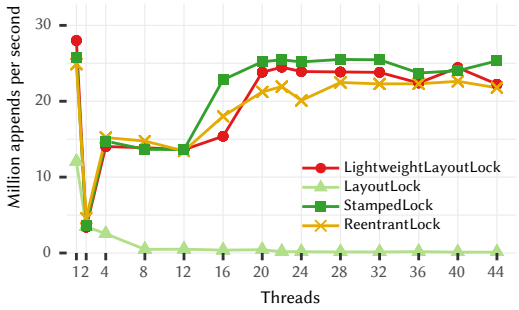

Fig. 6. Throughput of concurrent array appends. The `LayoutLock` performs poorly due to frequent layout changes.

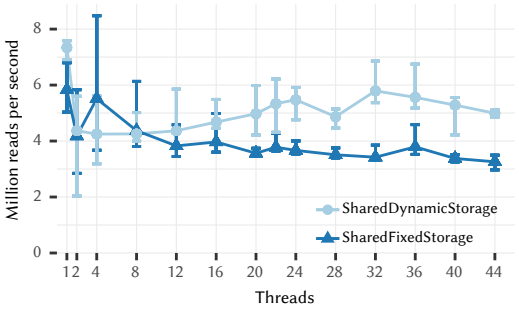

Fig. 7. Throughput of array reads when migrating 1000 `SharedFixedStorage` arrays every 5 seconds compared to starting with `SharedDynamicStorage`.

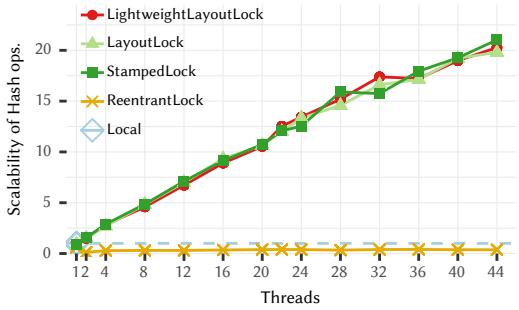

Fig. 8. Scalability of the Ruby `Hash` with 80% contains, 10% put, and 10% delete operations over 65,536 keys, relative to 1-thread performance of `Local`.

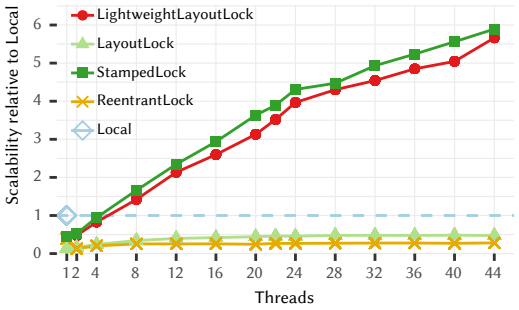

Fig. 9. Scalability of `group_by` benchmark inserting 40 millions random values into 65,536 bins. The dashed line shows the `Local` (single-threaded) baseline.

`SharedDynamicStorage` strategy, so that they do not need migrations and safepoints. The `SharedDynamicStorage` strategy uses the `LightweightLayoutLock` as it is the fastest lock for array reads. Figure 7 shows that `SharedFixedStorage` is slower than `SharedDynamicStorage`, particularly on a large number of threads, as it needs to perform the extra safepoints, which block all threads while performing the migration. However, we expect applications that care about performance to be designed so that frequent migrations are avoided and the benefit of `SharedFixedStorage` offsets the potential cost of migration.

7.4 Micro-Benchmarks for Hash Performance

We measure the scalability of `Hash` using various locks with 80% contains, 10% put, and 10% delete operations over a range of 65,536 keys. We also measure `Local`, i.e., the strategy used for non-shared collections. Figure 8 shows that the different locks scale similarly except for ReentrantLock, which does not support parallel operations. However, they do not scale perfectly and achieve a speedup of 20x on 44 cores over `Local`. This is due to maintaining the insertion order which creates contention for put/delete operations. The Lightweight Layout Lock has 14% overhead over `Local` for one thread and reaches 603 million `Hash` operations per second on 44 threads.

7.5 Idiomatic Combination of Array and Hash

To evaluate the combination of `Hash` and `Array` on an idiomatic program, we implement a parallel `group_by` operation. `Array#group_by` categorizes elements into bins and returns a `Hash` mapping

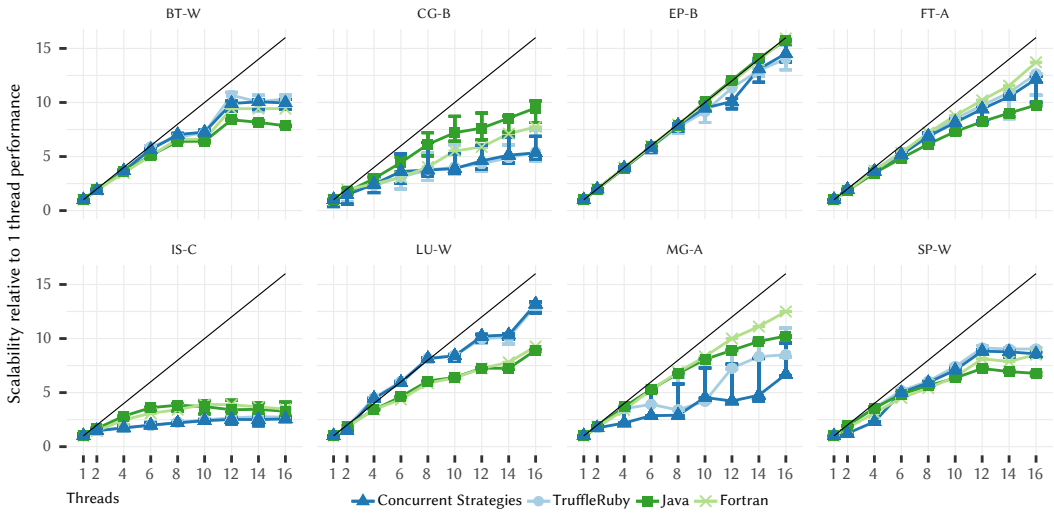

Fig. 10. Scalability relative to 1-thread performance per language on the NAS Parallel Benchmarks. The black lines denote ideal scalability based on 1-thread performance. All implementations show similar scalability.

each category to a bin represented as an `Array`. Elements are added to their bin using an `Array` append operation. For instance, we can group an `Array` of test results by their grade. We measure grouping 40 million uniformly-distributed values into 65,536 bins, categorizing by their range. `Hash` and `Array` use the `Local` strategy for single-threaded execution, and the different locks otherwise. The concurrent strategies provide sufficient correctness guarantees for this benchmark, thus no application-level synchronization is required. Figure 9 shows that Lightweight Layout Lock and `StampedLock` scale similarly well up to 44 cores, although not perfectly due to contention on `Array` appends. ReentrantLock serializes `Hash` operations and Layout Lock is slow due to frequent `Array` appends. Thus, we can run programs relying on the GIL for thread-safety of collection operations and run them up to 6x faster by running them in parallel.

7.6 Classic Parallel Benchmarks

We evaluate the scalability of our approach on 12 classic benchmarks and compare to other implementations where possible. Shared arrays in these benchmarks are allocated with a fixed size and their storage is never changed, therefore they all use the `SharedFixedStorage` strategy.

These benchmarks represent the class of standard parallel algorithms, as found in text books or actual implementations. Thus, they are designed with languages in mind that are much less dynamic than Ruby and Python. We want to compare with less dynamic languages such as Java and Fortran because they are known to be highly scalable. These benchmarks exercise only a small subset of behaviors that one could see in Ruby and Python code. This means for instance that they can run correctly on TruffleRuby without concurrent strategies, since they do not exhibit any problematic behavior that would cause concurrent layout changes.

Since such algorithms are widely available, we assume that applications that want to utilize parallelism in dynamic languages will start by adopting them and therefore it is worthwhile to optimize for them. Another observation made by Bolz et al. [2013] is that performance-sensitive code in dynamic languages often shows homogeneous behavior (e.g., the storage strategy of an `Array` does not change), making these kind of optimizations more broadly applicable.

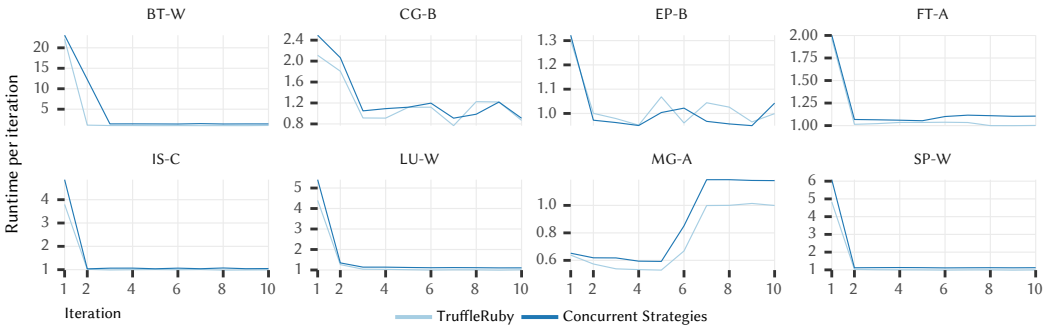

Fig. 11. Runtime per iteration with 8 threads relative to the median run time for `TruffleRuby`, illustrating the warmup of `TruffleRuby` and `Concurrent Strategies` on the NAS Parallel Benchmarks. Warmup graphs for different number of threads are similar. Lower is better.

NAS Parallel Benchmarks. We use the NASA Advanced Supercomputing Parallel Benchmarks 3.0 (NPB) [Bailey et al. 1991] to evaluate our approach. They were designed to evaluate the performance of parallel computers and are derived from computational fluid dynamics applications focusing on arithmetic operations and array accesses. We use the OMP variant of the Fortran benchmarks as it is the source of the translation to Java. The Ruby version was automatically translated from Java by Nose [2013]. We had to adapt the Ruby version to be more faithful to the Java version, so constants in the Java version are constants in Ruby and not mutable variables. We ported EP ourselves from Fortran to Ruby and Java, as no translation was available. Note that the Fortran version is much more optimized. For instance, the benchmark parameters are compiled in as constants, while they are read from object fields in Java and Ruby. Consequently, we focus on scalability.

The benchmarks for Java and Ruby did not include warmup, so we modified them to run 10 iterations in the same process and remove the first 2 as warmup. We picked the largest benchmark classes such that a benchmark iteration takes at most 9 minutes when run with one thread.

Figure 10 shows the scalability of the benchmarks, relative to the single-threaded performance per language. We observe that all languages scale similarly on these benchmarks. EP and FT scale almost perfectly, while other benchmarks have a significant part of serial work.

Figure 11 shows the warmup behavior with 8 threads. On MG, TruffleRuby unfortunately does not warmup, which leads to a slowdown after a few iterations. Other benchmark warmup as expected within the first 2 iterations. While the global warmup behavior is similar, we observe an overhead between 2% and 28% (geometric mean: 13%) on the first iteration for Concurrent Strategies over unmodified TruffleRuby for the 8 benchmarks. However, first iteration run time can vary significantly from run to run due to many factors including the scheduling of compiler jobs in the background and the point when execution switches to optimized code.

In terms of absolute performance on one thread, Java is between 0.9x and 2.4x slower than Fortran (1.5x by geometric average). The unmodified unsafe TruffleRuby is between 1x and 4.5x slower than Java (2.3x by geometric average). Our version with concurrent strategies and their safety guarantees is on par with unsafe TruffleRuby, which is visible in the plots by having overlapping error bars. Overall, we conclude that our approach provides a safe and scalable programming model enabling classic parallel algorithms.

PyPy-STM Mandelbrot. We ported the parallel Mandelbrot benchmark from PyPy-STM [Meier et al. 2016] to Ruby. The benchmark renders the Mandelbrot set and distributes 64 groups of rows between threads using a global queue. The benchmark uses arrays in a disciplined manner that

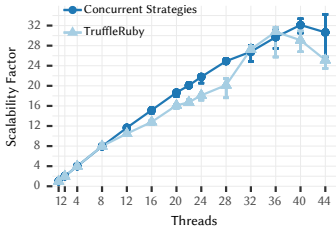

Fig. 12. Scalability of the parallel Mandelbrot benchmark, relative to 1-thread performance of TruffleRuby.

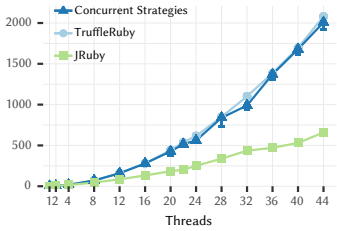

Fig. 13. Scalability of the threaded-reverse benchmark, relative to 1-thread performance *of each implementation.*

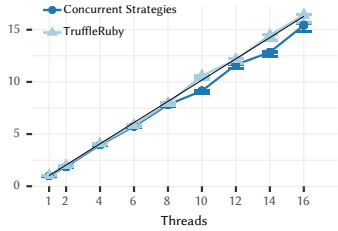

Fig. 14. Scalability of the Sidekiq Sum benchmark with Concurrent Strategies, relative to 1-thread performance of TruffleRuby.

does not cause storage changes and thus does not require synchronization. Therefore, we can safely use it to measure the overhead of the `SharedFixedStorage` strategy compared to the original thread-unsafe TruffleRuby. As shown in Figure 12, the benchmark scales up to 20x faster on 22 threads and keeps scaling on 2 NUMA nodes up to 32x with 40 threads. The results show that the `SharedFixedStorage` strategy has no significant overhead on this benchmark while providing thread safety for shared collections.

JRuby threaded-reverse. We run the `threaded-reverse` benchmark from JRuby [JRuby 2008], except that we reverse arrays instead of strings. The benchmark is a fixed workload reversing 240 arrays each containing 100,000 integers. As shown in Figure 13, both `SharedFixedStorage` and JRuby achieve super-linear speedup (up to 2007x on 44 cores). As more threads are added, each thread has a smaller number of arrays to reverse so that they progressively fit in each processor's cache. In terms of absolute performance, our approach is faster than JRuby by 62x on one thread and by 189x on 44 threads on this benchmark. Thus, we achieve significantly better performance while also providing thread safety.

7.7 Realistic Ruby Benchmarks

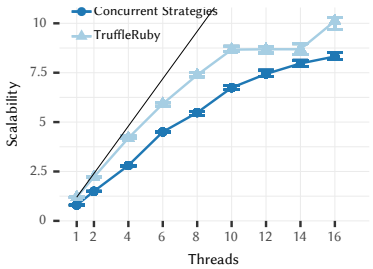

Fig. 15. Scalability of the Sidekiq PDF Invoice benchmark with Concurrent Strategies, relative to 1-thread performance of TruffleRuby.

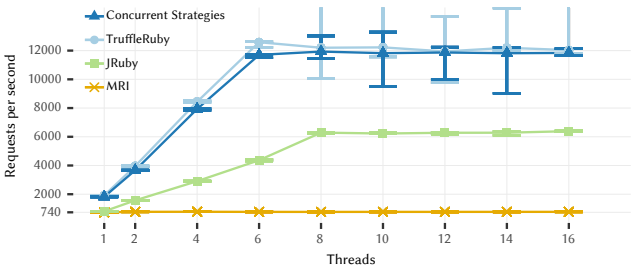

Fig. 16. Throughput of the WEBrick benchmark for "Hello World" requests. No implementation scales beyond 8 threads as WEBrick creates a new thread for each request. Error bars indicate the first and third quartiles as the throughput has many outliers.

We run 3 larger Ruby benchmarks to see how our approach works in realistic scenarios. Unfortunately, TruffleRuby is not yet compatible enough to run Ruby on Rails applications. Instead, we benchmark background processing and a HTTP server.

Sidekiq. We run a simplified version of the popular Ruby background processing library, Sidekiq [Perham 2017], without an external Redis instance. Each thread has a queue of jobs and each job

is passed as a JSON string, which is deserialized and contains the information about the method call to be performed. We run two variants, one where the job is a loop summing integers to assess scalability and one where the job is to generate a PDF invoice using the Prawn pure-Ruby library (of around 8500 lines of code with dependencies) [Brown et al. 2017a]. Figure 14 shows that the sum variant scales perfectly and Figure 15 shows that the invoice variant achieves a 8x speedup on 16 threads, generating 3286 invoices per second.

We see that PDF invoice generation can be 8-33% slower compared to the unsafe version. While most benchmarks see no slowdown because of our optimization for local collections, Prawn does many accesses to shared `Hash` instances: 2.7 million reads and 1.6 million writes per second on one thread. Consequently, it sees some overhead for the now thread-safe `Hash` implementation with the `ConcurrentBuckets` strategy, which uses lock-free algorithms for read and write accesses.

Mutex Not Needed with Thread-Safe Collections. Interestingly, we found a `Hash`-based cache [Brown et al. 2017b] in the Prawn library that was protected by a `Mutex` for JRuby but does not need it with thread-safe collections. This cache is used while loading fonts during the setup phase of the benchmark and therefore removing the `Mutex` has no impact on peak performance.

WEBrick. As third benchmark, we measure the throughput of WEBrick, the HTTP server in the standard library, simply delivering "Hello World". Figure 16 shows that WEBrick does not scale beyond 8 threads due to spawning a new thread for each request. Yet, we achieve a speedup of 15.5x over MRI and 1.9x over JRuby on 8 threads.

7.8 Summary

The evaluation shows that when a collection does not need to undergo layout changes, the `SharedFixedStorage` is a fast, yet safe strategy. However, when the collection's storage needs to be changed dynamically, our Lightweight Layout Lock provides a balanced performance in all evaluated scenarios and is better than the Layout Lock. For concurrent writes, the Lightweight Layout Lock improvements significantly over StampedLock and ReentrantLock and enables scalability (cf. Figure 5). As a result, we use it for the `SharedDynamicStorage` strategy and the `ConcurrentBuckets` strategy.

8 RELATED WORK

Section 2 already discussed the basic techniques on which our work relies. Here, we further detail how our work relates to adaptive collections and synchronization techniques.

Adaptive Data Representations and Collections. Daloze et al. [2016] introduced thread-safe objects for dynamic languages, distinguishing local and shared objects. They track sharing of built-in collections similarly to other objects. However, their collection implementations are not thread-safe.

To optimize the data representation of collections, Bolz et al. [2013] use storage strategies to avoid boxing in homogeneous arrays, dictionaries, and sets. This reduces the memory footprint of collections and avoids the computational overhead of boxing. However, Bolz et al. used PyPy, which uses a GIL, and thus did not consider concurrency issues.

A generic approach to adaptive data representations was proposed by De Wael et al. [2015] enabling general representation changes at run time. While their approach can optimize aspects such as memory locality in addition to boxing, they did not consider concurrency issues either.

In contrast to these works, this paper proposes an approach to address concurrency for optimized collection representations. As detailed earlier, concurrent collection strategies enable the efficient synchronization of adaptive data representations, and with our approach to gradual synchronization, they adapt themselves to the concrete usage, too.

Newton et al. [2015] demonstrate the soundness of dynamically transitioning between multiple representation of collections to address scalability. Their solution is limited to purely functional data structures that have lock-free counterparts. Purely functional data structures do not scale due to contention on a single mutable variable. Lock-free data structures introduce overhead in low contention scenarios. Therefore adapting the representation based on contention helps improve performance in both scenarios. Their solution is not applicable to dynamic programming languages due to the strict limitation of relying on purely functional data structures.

Layout Lock. As detailed in Section 5.6, the Lightweight Layout Lock is an extension of Cohen et al. [2017]'s Layout Lock designed for synchronizing collections in dynamic languages. The original Layout Lock was too restricted due to its static design for thread registration and handling of consecutive layout changes. It expects to know the maximum number of threads in advance and allocates a fixed-length array of that size to store the lock's state. This also affects the memory footprint of the Layout Lock, which depends on the maximum number of threads. This memory footprint issue becomes worse when applying padding to reduce contention due to false sharing. The Lightweight Layout Lock fixes these issues and improves on three major points. Consecutive layout changes are optimized to perform as fast as with an exclusive lock instead of severely degrading scalability (cf. Figure 6). Threads register dynamically with the lock, which is needed as there is no known upper bound on the number of threads in dynamic language runtimes. Finally, as a consequence, the memory footprint of the lock is improved as only threads registered with the lock are tracked in the lock's state.

VM-level Synchronization Techniques. Thread safety of data representations for dynamic language implementations has seen only limited attention so far. As mentioned earlier, JavaScript avoids the issue by not allowing sharing of objects or collections and Ruby and Python rely on GILs, which sequentialize all relevant execution.

Odaira et al. [2014] use hardware transactional memory (HTM) to replace the GIL in MRI. Their approach simulates the GIL semantics and even though it adds extra yield points, per-bytecode atomicity is preserved. However, their evaluation with the NAS Parallel Benchmarks shows lower scalability as well as an overhead for single-threaded performance.

Meier et al. [2016, 2018] use software transactional memory (STM) to simulate GIL semantics in PyPy-STM, allowing certain workloads to achieve parallel speedups. Their approach replicates the GIL semantics faithfully by monitoring all memory accesses and breaking atomicity only between bytecodes. However, their STM system adds an overhead on sequential performance and scalability.

Such STM and HTM systems add overhead to all shared memory accesses, since they do not distinguish between potentially benign interpreter memory accesses and the application accessing its data. Our approach confines guarantees to the implementation-level of collections and thus does not have the overhead of monitoring memory accesses of the interpreter. Furthermore, our concurrent strategies adapt to specific usage patterns, which provides more optimization potential.

Synchronization in Lisp Systems. Common Lisp runtimes such as SBCL and LispWorks support parallelism, but the handling of built-in collections such as vectors and hash tables is implementation dependent. On SBCL, these collections are unsynchronized by default, causing exceptions and lost updates when used concurrently. A `:synchronized` constructor argument can be used to create a synchronized hash table.[6] LispWorks synchronizes by default, paying a performance cost that can be avoided by passing the `single-thread` argument.[7] With our technique, this manual approach could be avoided, sources for errors removed, and performance for single threaded execution improved.

[6]SBCL User Manual: http://www.sbcl.org/manual/#Hash-Table-Extensions
[7]Documentation of `make-hash-table`: http://www.lispworks.com/documentation/lw60/LW/html/lw-608.htm

For Racket, Swaine et al. [2010] introduced *futures* to add parallelism to the existing runtime with minimal effort. They distinguish *safe*, *unsafe*, and *synchronized* primitive operations. Only *safe* operations run in parallel. Collection operations in futures are considered as *unsafe* or *synchronized* and therefore are serialized. In other work, Tew et al. [2011] proposed *places*, an actor-like concurrency model for Racket that allows sharing mutable fixed-size arrays of primitive values (bytes, integers, floats). *Places* prevent sharing mutable data structures containing objects (e.g., hash tables) or arrays with variable-length and therefore avoid thread safety issues such as in Table 1.

Shared-Memory JavaScript. In JavaScript, only primitive data can be shared between parallel workers using a `SharedArrayBuffer` [Mozilla 2018]. Pizlo [2017] proposed an idea for efficient shared-memory including arbitrary objects that is tightly coupled to garbage collection and object representation properties of the underlying VM. It is based on *transition-thread-local* objects, which are objects and built-in collections that do all layout changes on the thread that created the object. On access, such objects and collections check that the current thread is the creator thread. In contrast, our notion of *local* implies no overhead, but applies to fewer objects. He proposes either using exclusive per-array locking or *compare-and-swap* for array resizes. However, it is not detailed how the index for a new element and the new array size are computed for concurrent appends. A single CAS on the storage is insufficient, because appends could overwrite each other. Other operations such as deleting an element migrates arrays to a dictionary representation. For dictionaries, he proposes using an exclusive lock on all accesses. Our approach enables scaling for dictionaries and allows all array operations without the need to fallback to a less efficient representation.

9 CONCLUSION

Implementations for languages such as Python, Ruby, and JavaScript have limited support for parallelism. They either sequentialize all operations on a collection or do not support safe concurrent modification causing for instance lost updates or low-level exceptions. In this paper, we made the next step for parallelism in dynamic languages by enabling safe, efficient, and parallel access to collections. We introduced a method of gradual synchronization, migrating collections between multiple levels of synchronization, depending on the dynamic needs of each collection instance.

Collections reachable only by a single thread do not need to be synchronized and retain single-threaded performance. When a collection is shared with other threads, the collection switches to a *concurrent strategy*. We designed and implemented three concurrent strategies for synchronizing two central collections, arrays and dictionaries, that are common to many dynamic programming languages. The main principles of these strategies are applicable to other collections.

We designed our strategies based on usages patterns of collections. Concurrent algorithms often use fixed-size arrays and rely on parallel read and write accesses. Therefore, the first strategy supports parallel access to fixed-size arrays, with no overhead as long as the array size and storage do not need to change. When the fixed-size assumptions no longer holds, the array is migrated to a more general strategy, which handles arbitrary changes to the array. This strategy uses the novel Lightweight Layout Lock, a refined version of the Layout Lock [Cohen et al. 2017], which enables parallel read and write accesses to the array and synchronizes them with *layout changes*, i.e., larger structural changes such as resizing the array, changing the storage type, etc.

For dictionaries, we designed a single concurrent strategy for simplicity. The strategy uses a lock-free list to maintain the insertion order as well as the Lightweight Layout Lock to synchronize accesses to the buckets array. This enables parallel lookups and concurrent insertions and removals.

We apply these techniques to TruffleRuby's built-in collections `Array` and `Hash`. Using our method of gradual synchronization, the implementations of these collections now provide thread safety guarantees that are similar to those provided by a global interpreter lock, yet without inhibiting

scalability. Specifically, they do not expose implementation details in the form of internal thread-safety issues such as lost updates, out-of-bounds errors, and out-of-thin-air values. This avoids causing data races that do not exist in the application code.

With such thread safety guarantees, dynamic language developers can now rely on their typical programming idioms for building parallel applications and take advantage of scalable collections. We believe that this is by far superior compared to requiring them to use different collection types for concurrency when moving to language implementations without a global lock. Since our approach is transparent, thread safety guarantees for built-in collections are always provided without requiring error-prone application changes to protect against implementation-level data races (e.g., by changing the used collection type or adding manual synchronization). While this automatic synchronization of collections is particularly important for dynamic languages with their few and versatile collections, we believe it could also be beneficial for static languages, where thread safety of basic collections could be guaranteed while preserving performance.

The evaluation shows that our approach does not impact the performance of single-threaded programs and collections that are only reachable by a single thread. Array accesses with the fixed-size strategy show zero overhead and scale linearly, and shared Hash operations show 14% overhead compared to the unsynchronized Hash. We show that our approach scales similarly to Fortran and Java on the NAS Parallel Benchmarks. Finally, we observe that the safety provided by our approach comes with an overhead of 5% for the geometric mean over all non-micro benchmarks, from 33% slower up to 1% faster, by comparing results on 1 thread. We consider this overhead acceptable since it brings the safety and convenience known from GIL-based implementations, while at the same time enabling scalable parallelism. We hope that these results inform memory models for dynamic languages and encourages them to provide stronger guarantees for built-in collections.

Given the success of our experiments we plan to integrate this work into TruffleRuby. By providing efficient and thread-safe collection implementations for dynamic languages, we hope to bring a parallel and safer multi-threaded programming experience to dynamic languages, paving the way to a better utilization of today's multi-core hardware.

Future Work. Combining thread-safe collection implementations and thread-safe objects yields two of the most important guarantees of a GIL, without its drawbacks. Future work could explore which other guarantees provided by a GIL are useful to the user or conversely, which implementation details of the VM should not leak to the user level. Such work should also investigate how to formalize these guarantees to precisely define the tradeoff between GIL-semantics and our approach.

Another open question is how the internal synchronization of collections and objects can be combined safely and efficiently with user-level synchronization to avoid redundant synchronization.

It would also be worthwhile to investigate further strategies in addition to the proposed ones. While the dynamic language collections are very versatile, we assume that concrete instances use only a subset of operations, e.g., to use an array as a stack. Strategies optimized for such use cases might provide further performance benefits. Similarly, behavior might change in phases, which might make it useful to consider the resetting of a strategy.

ACKNOWLEDGMENTS

We gratefully acknowledge the support of the Virtual Machine Research Group at Oracle Labs, the Institute for System Software at JKU Linz and everyone else who has contributed to Graal and Truffle. We would like to thank Manuel Rigger, Lukas Stadler and Chris Seaton for their careful reviews. Benoit Daloze was funded by a research grant from Oracle. Stefan Marr was partially funded by a grant of the Austrian Science Fund, project number I2491-N31. Arie Tal and Erez Petrank were supported by the Israel Science Foundation under Grant No. 274/14.

REFERENCES

David H. Bailey, Eric Barszcz, John T. Barton, David S. Browning, Robert L. Carter, Leonardo Dagum, Rod A. Fatoohi, Paul O. Frederickson, Thomas A. Lasinski, Rob S. Schreiber, et al. 1991. The NAS Parallel Benchmarks. *The International Journal of Supercomputing Applications* 5, 3 (1991), 63–73. https://doi.org/10.1177/109434209100500306

Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the Meta-level: PyPy's Tracing JIT Compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS '09)*. ACM, 18–25. https://doi.org/10.1145/1565824.1565827

Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. 2013. Storage Strategies for Collections in Dynamically Typed Languages. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, 167–182. https://doi.org/10.1145/2509136.2509531

Carl Friedrich Bolz and Laurence Tratt. 2013. The Impact of Meta-Tracing on VM Design and Implementation. *Science of Computer Programming* (2013). https://doi.org/10.1016/j.scico.2013.02.001

Gregory Brown, Brad Ediger, Alexander Mankuta, et al. 2017a. Prawn: Fast, Nimble PDF Generation For Ruby. http://prawnpdf.org/

Gregory Brown, Brad Ediger, Alexander Mankuta, et al. 2017b. Prawn::SynchronizedCache with a Mutex for JRuby. https://github.com/prawnpdf/prawn/blob/61d46791/lib/prawn/utilities.rb

Craig Chambers, David Ungar, and Elgin Lee. 1989. An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89)*. ACM, 49–70. https://doi.org/10.1145/74878.74884

Daniel Clifford, Hannes Payer, Michael Stanton, and Ben L. Titzer. 2015. Memento Mori: Dynamic Allocation-site-based Optimizations. In *Proceedings of the 2015 International Symposium on Memory Management (ISMM '15)*. ACM, 105–117. https://doi.org/10.1145/2754169.2754181

Nachshon Cohen, Arie Tal, and Erez Petrank. 2017. Layout Lock: A Scalable Locking Paradigm for Concurrent Data Layout Modifications. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. ACM, 17–29. https://doi.org/10.1145/3018743.3018753

Diego Costa, Artur Andrzejak, Janos Seboek, and David Lo. 2017. Empirical Study of Usage and Performance of Java Collections. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE'17)*. ACM, 389–400. https://doi.org/10.1145/3030207.3030221

Benoit Daloze, Stefan Marr, Daniele Bonetta, and Hanspeter Mössenböck. 2016. Efficient and Thread-Safe Objects for Dynamically-Typed Languages. In *Proceedings of the 2016 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '16)*. ACM, 642–659. https://doi.org/10.1145/2983990.2984001

Benoit Daloze, Chris Seaton, Daniele Bonetta, and Hanspeter Mössenböck. 2015. Techniques and Applications for Guest-Language Safepoints. In *Proceedings of the 10th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS '15)*. 1–10. https://doi.org/10.1145/2843915.2843921

Benoit Daloze, Arie Tal, Stefan Marr, Hanspeter Mössenböck, and Erez Petrank. 2018. Supplementary Material for: Parallelization of Dynamic Languages: Synchronizing Built-in Collections. In *Proceedings of the 2018 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'18)*. https://doi.org/10.1145/3276478

Jerry D'Antonio and Petr Chalupa. 2017. Concurrent Ruby – Modern concurrency tools for Ruby. http://www.concurrent-ruby.com

Mattias De Wael, Stefan Marr, Joeri De Koster, Jennifer B. Sartor, and Wolfgang De Meuter. 2015. Just-in-Time Data Structures. In *Proceedings of the 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! '15)*. ACM, 61–75. https://doi.org/10.1145/2814228.2814231

Ulan Degenbaev, Jochen Eisinger, Manfred Ernst, Ross McIlroy, and Hannes Payer. 2016. Idle Time Garbage Collection Scheduling. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16)*. ACM, 570–583. https://doi.org/10.1145/2908080.2908106

Liang Gong, Michael Pradel, and Koushik Sen. 2015. JITProf: Pinpointing JIT-unfriendly JavaScript Code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, 357–368. https://doi.org/10.1145/2786805.2786831

David Heinemeier Hansson et al. 2018. Ruby on Rails - A web-application framework that includes everything needed to create database-backed web applications according to the Model-View-Controller (MVC) pattern. http://rubyonrails.org/

Tim Harris, James Larus, and Ravi Rajwar. 2010. *Transactional Memory, 2nd Edition* (2nd ed.). Morgan and Claypool Publishers.

Timothy L. Harris. 2001. A Pragmatic Implementation of Non-Blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC '01)*. Springer, 300–314. https://doi.org/10.1007/3-540-45414-4_21

Maurice Herlihy and Nir Shavit. 2011. *The Art of Multiprocessor Programming*. Morgan Kaufmann.

Mark A. Hillebrand and Dirk C. Leinenbach. 2009. Formal Verification of a Reader-Writer Lock Implementation in C. *Electronic Notes in Theoretical Computer Science* 254 (2009), 123–141.

JRuby. 2008. JRuby's threaded-reverse benchmark. https://github.com/jruby/jruby/blob/a0a3e4bd22/test/bench/bench_threaded_reverse.rb

Josh Juneau, Jim Baker, Frank Wierzbicki, Leo Soto, and Victor Ng. 2010. *The Definitive Guide to Jython: Python for the Java Platform.* Apress.

Hemant Kumar. 2013. Bug report on the RubyGems project: Thread issues with JRuby. https://github.com/rubygems/rubygems/issues/597

Yi Lin, Kunshan Wang, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. 2015. Stop and Go: Understanding Yieldpoint Behavior. In *Proceedings of the 2015 International Symposium on Memory Management (ISMM '15)*. ACM, 70–80. https://doi.org/10.1145/2754169.2754187

Stefan Marr and Benoit Daloze. 2018. Few Versatile vs. Many Specialized Collections – How to design a collection library for exploratory programming?. In *Proceedings of the 4th Programming Experience Workshop (PX/18)*. ACM. https://doi.org/10.1145/3191697.3214334

Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-Language Compiler Benchmarking—Are We Fast Yet?. In *Proceedings of the 12th Symposium on Dynamic Languages (DLS'16)*. ACM, 120–131. https://doi.org/10.1145/2989225.2989232

Remigius Meier, Armin Rigo, and Thomas R. Gross. 2016. Parallel Virtual Machines with RPython. In *Proceedings of the 12th Symposium on Dynamic Languages (DLS '16)*. ACM, 48–59. https://doi.org/10.1145/2989225.2989233

Remigius Meier, Armin Rigo, and Thomas R. Gross. 2018. Virtual Machine Design for Parallel Dynamic Programming Languages. In *Proceedings of the 2018 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '18)*. ACM.

Mozilla. 2018. SharedArrayBuffer Documentation. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer

Ryan R. Newton, Peter P. Fogg, and Ali Varamesh. 2015. Adaptive Lock-Free Maps: Purely-Functional to Scalable. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, 218–229. https://doi.org/10.1145/2784731.2784734

Takafumi Nose. 2013. Ruby version of NAS Parallel Benchmarks 3.0. https://github.com/plus7/npb_ruby

Charles Nutter, Thomas Enebo, Ola Bini, Nick Sieger, et al. 2018. JRuby – The Ruby Programming Language on the JVM. http://jruby.org/

Rei Odaira, Jose G. Castanos, and Hisanobu Tomari. 2014. Eliminating Global Interpreter Locks in Ruby Through Hardware Transactional Memory. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. ACM, 131–142. https://doi.org/10.1145/2555243.2555247

Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. 2005. *Java Concurrency in Practice.* Addison-Wesley Professional.

Mike Perham. 2017. Sidekiq – Simple, efficient background processing for Ruby. https://sidekiq.org/

Evan Phoenix, Brian Shirai, Ryan Davis, Dirkjan Bussink, et al. 2018. Rubinius – An Implementation of Ruby Using the Smalltalk-80 VM Design. https://rubinius.com/

Filip Pizlo. 2017. Concurrent JavaScript: It can work! https://webkit.org/blog/7846/concurrent-javascript-it-can-work/

William Pugh et al. 2004. JSR 133 - Java Memory Model and Thread Specification Revision. http://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf

Rails Project. 2018. Threading and Code Execution in Rails. http://edgeguides.rubyonrails.org/threading_and_code_execution.html

Chris Seaton, Benoit Daloze, Kevin Menard, Petr Chalupa, Brandon Fish, and Duncan MacGregor. 2017. TruffleRuby – A High Performance Implementation of the Ruby Programming Language. https://github.com/graalvm/truffleruby

Aleksey Shipilëv. 2014. Safe Publication and Safe Initialization in Java. https://shipilev.net/blog/2014/safe-public-construction/

Brian Shirai. 2016. Bug report on the Bundler project: Unsynchronized, concurrent modification of Set instance. https://github.com/bundler/bundler/issues/5142

Brian Shirai. 2018. Bug report on the Bundler project: Unsychronized concurrent updates of Hash. https://github.com/bundler/bundler/issues/6274

Vincent St-Amour and Shu-yu Guo. 2015. Optimization Coaching for JavaScript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015) (LIPIcs)*, Vol. 37. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 271–295. https://doi.org/10.4230/LIPIcs.ECOOP.2015.271

Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, 165–174. https://doi.org/10.1145/2544137.2544157

Håkan Sundell and Philippas Tsigas. 2005. Lock-Free and Practical Doubly Linked List-Based Deques Using Single-Word Compare-and-Swap. *Principles of Distributed Systems* (2005), 240–255. arXiv:arXiv:cs/0408016v1

James Swaine, Kevin Tew, Peter Dinda, Robert Bruce Findler, and Matthew Flatt. 2010. Back to the Futures: Incremental Parallelization of Existing Sequential Runtime Systems. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, 583–597. https://doi.org/10.1145/1869459.1869507

Kevin Tew, James Swaine, Matthew Flatt, Robert Bruce Findler, and Peter Dinda. 2011. Places: Adding Message-Passing Parallelism to Racket. In *Proceedings of the 7th Symposium on Dynamic Languages (DLS '11)*. ACM, 85–96. https://doi.org/10.1145/2047849.2047860

Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, 662–676. https://doi.org/10.1145/3062341.3062381

Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. 2014. An Object Storage Model for the Truffle Language Implementation Framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '14)*. ACM, 133–144. https://doi.org/10.1145/2647508.2647517

Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-Optimizing AST Interpreters. In *Proceedings of the 8th Dynamic Languages Symposium (DLS '12)*. 73–82. https://doi.org/10.1145/2384577.2384587

Guoqing Xu. 2013. CoCo: Sound and Adaptive Replacement of Java Collections. In *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP'13)*. Springer-Verlag, 1–26. https://doi.org/10.1007/978-3-642-39038-8_1

SUPPLEMENTARY MATERIAL FOR: PARALLELIZATION OF DYNAMIC LANGUAGES: SYNCHRONIZING BUILT-IN COLLECTIONS

This is supplementary material for main part of the paper.

A REASONING ABOUT CORRECTNESS FOR THE LIGHTWEIGHT LAYOUT LOCK

Similarly to the Layout Lock [Cohen et al. 2017], the Lightweight Layout Lock must satisfy the following properties: (a) layout change operations are mutually exclusive, (b) write operations are not concurrent with a layout change operation, and (c) optimistic read operations detect a concurrent layout change operation and fail, allowing for recovery.

In this section, we sketch the correctness of the Lightweight Layout Lock protocol by describing invariants, which correspond to the above correctness properties. We argue that if these invariants hold, the lock is providing the necessary synchronization for concurrent strategies.

The correctness of the Lightweight Layout Lock invariants depends on a `baseLock` that correctly satisfies shared vs. exclusive locking invariants, such as a Reader-Writer Lock [Hillebrand and Leinenbach 2009].

The Lightweight Layout Lock does not support nesting API calls, nor does it support upgrading (i.e. atomically turning a read operation into a write operation, or a write operation into a layout change operation). Similar to other locks, an operation is comprised of the following steps: (a) a *prologue*, usually an API call (a call to a `start` method in the case of the Lightweight Layout Lock), (b) a *body*, the user code that performs the actions that are synchronized by the lock, and finally (c) an *epilogue*, a symmetric API to the *prologue* that performs synchronization actions related to finishing the operation (a call to a `finish` method in the case of the Lightweight Layout Lock).

A `threadState` is an `AtomicInteger` that associates a thread and a Lightweight Layout Lock instance. The thread states associated with a specific lock instance are maintained in that lock's `threadStates` list. The value of a `threadState` is a combination of two bits: The `LC` bit and `WR` bit, representing four states: `0` (*Inactive*), `WR` (*Writing*), `LC` (*Layout Change*), and `LC + WR` (*Writing with Layout Change Pending*). The `threadState` value is modified by the various API calls, as shown in

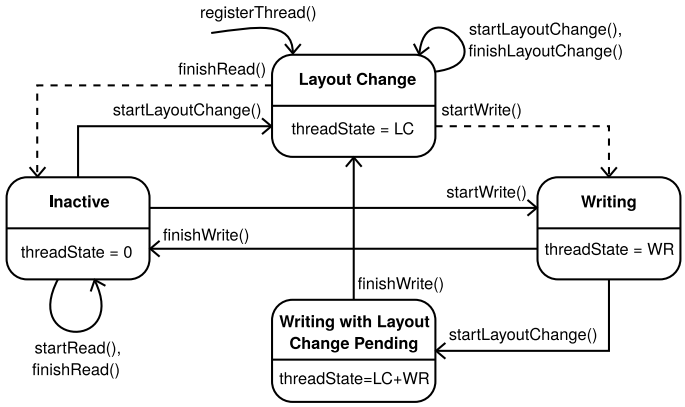

Fig. 17. This diagram shows all the state transitions of a `threadState`, as they are triggered by the different API calls of the Lightweight Layout Lock. The dashed arrows indicate the reader and writer slow paths (i.e. a call to `slowSet()` in Listing 2).

Supp. Figure 17. Each transition of the `threadState` is performed atomically either by actions of the owning thread or by a different thread that starts a layout change.

Invariant 1. *For some* `threadState` *of thread T and lock lll, if T clears the* `LC` *bit, then*

- *T called either* `finishRead()` *or* `startWrite()`.
- *the* `rescan` *flag was set to* `true` *prior to clearing of the* `LC` *bit.*
- *clearing of the* `LC` *bit was performed while the baseLock was held in shared mode.*

The `slowSet()` method is used to set `rescan` to `true` and clear the `LC` bit. It is not invoked directly by user code, but rather as part of the read operation epilogue and the write operation prologue. It is the only method provided by the Lightweight Layout Lock that can clear the `LC` bit. The `slowSet()` method locks the `baseLock` in shared mode prior to modifying `rescan` and the `threadState`, and releases the `baseLock` when done. Note that although `startWrite()` and `finishWrite()` update the `threadState` directly (at lines 22 and 30 in Listing 2, respectively), they only modify the `WR` bit.

Invariant 2. *For a given Lightweight Layout Lock instance lll,*

$$lll.\texttt{rescan} = \texttt{false} \implies \forall ts \in lll.\texttt{threadStates}, ts = \texttt{LC}\ \textit{(Layout Change)}$$

When the Lightweight Layout Lock is initialized, the `rescan` flag is set to `false` (at line 4 in Listing 2) and there are no threads registered with the lock (i.e. `threadStates = {}`). So this invariant holds when creating the lock since there are no `threadStates`.

When a new `threadState` is registered with the lock, it is first set to `LC` (*Layout Change*, at line 68), thus thread registration preserves this invariant.

When a reader calls `finishRead()` and detects (at line 13) that the `threadState` is not `0` (not *Inactive*), it calls `slowSet()` (at line 16) to set the `threadState` to `0` (*Inactive*). When a writer calls `startWrite()` and the CAS (at line 22) fails (i.e. the `threadState` value is `LC` (*Layout Change*)), it calls `slowSet()` (at line 25) to set the `threadState` to `WR` (*Writing*). By Invariant 1, `slowSet()` sets `rescan` to `true` prior to modifying the `threadState`. Therefore, in the only two cases that clear the `LC` bit, Invariant 2 is maintained.

The `rescan` flag can only be reset by `startLayoutChange()` when it detects (at line 46) that the `rescan` flag is `true`. It then scans all the registered `threadStates` and either verifies that they are already equal to `LC` (*Layout Change*), or turns on the `LC` bit, thus transitioning the `threadState` into either `LC` (*Layout Change*) or `LC + WR` (*Writing with Layout Change Pending*) (cf. Supp. Figure 17). If the thread that owns the `threadState` is concurrently performing a write operation, i.e. `threadState` equals `LC + WR` (*Writing with Layout Change Pending*), `startLayoutChange()` will busy-wait until the the thread calls `finishWrite()` and clears the `WR` bit.

After all `threadStates` have been set to `LC` (*Layout Change*), `startLayoutChange()` sets `rescan` to `false`. The scanning is performed while the `baseLock` is held in exclusive mode, and by Invariant 1, the `LC` bit is only cleared while holding the `baseLock` in shared mode (by `slowSet()`), thus clearing of the `LC` flag and setting the `LC` flag are mutually exclusive. The `LC` bits can only be set and the `rescan` flag can only be reset in `startLayoutChange()`. Therefore, when `startLayoutChange()` sets `rescan` to `false`, all registered `threadStates` must equal `LC` (*Layout Change*), satisfying Invariant 2.

Invariant 3. *For a given Lightweight Layout Lock instance lll,*

$$\exists ts \in lll.\texttt{threadStates}, ts \neq \texttt{LC}\ \textit{(Layout Change)} \implies lll.\texttt{rescan} = \texttt{true}$$

This is trivially true by a contraposition of Invariant 2.

Invariant 4. *The body of a write operation and the body of a layout change operations are mutually exclusive.*

For this invariant, we need to look at the two cases of a write operation: the fast path and the slow path. The write operation fast path begins by a successful CAS of the `threadState` from `0` (*Inactive*) to `WR` (*Writing*). If a layout change prologue runs concurrently, then by Invariant 3, the `rescan` flag must be `true`, and thus it loops over the registered `threadStates` and tries to set them to `LC` (*Layout Change*). However, since the CAS of `startWrite` succeeded for `threadState`, then the comparison at line 49 fails for that same `threadState` (either before or after the CAS), thus the value `LC` is atomically added to the `threadState` at line 51 resulting in `LC + WR` (*Writing with Layout Change Pending*). `startLayoutChange()` will then busy-wait until the thread calls `finishWrite()` and clears the `WR` bit, letting the `threadState` become `LC` (*Layout Change*) and releasing the busy-wait loop. Thus the layout change operation body cannot start until all write operations fast paths completed.

The write operation slow path begins by a failing CAS of the `threadState` from `0` (*Inactive*) to `WR` (*Writing*). The slow path proceeds by calling `slowSet()` with the value `WR` (*Writing*), which requires locking the `baseLock` in shared mode. However, since a layout change prologue begins by locking the `baseLock` in exclusive mode (`startLayoutChange()` at line 45), and releases the lock only after the layout change operation completed (at line 63), the write operation is suspended until the layout change operation is completed. Thus a write operation body cannot start before a layout change body completes.

Since `startWrite()` always sets the `WR` bit (either via a successful CAS or `slowSet()`), `finishWrite()` is the only operation clearing the `WR` bit, and `startLayoutChange()` waits until the `WR` bit is removed, the layout change operation body cannot start until all write operation bodies completed.

Invariant 5. *During a layout change operation body*

- `baseLock` *is held in exclusive mode.*
- `rescan` *is* `false`.
- *all registered* `threadStates` *equal* `LC` *(Layout Change).*

The layout change prologue in `startLayoutChange()` begins by acquiring the `baseLock` in exclusive mode. The lock is released at `finishLayoutChange()`, thus it is held in exclusive mode during the

layout change operation. When `startLayoutChange()` completes, `rescan` must equal `false`, since `startLayoutChange()` resets the `rescan` flag, and the flag can only be set by `slowSet()` which requires holding the `baseLock` in shared mode. However, the `baseLock` is held in exclusive mode during a layout change operation, thus `rescan` remains false at least until the end of the layout change operation. By Invariant 2 it is implied that all registered `threadStates` equal `LC` (*Layout Change*) when `rescan` is `false`.

Invariant 6. *For a given lock instance lll and a thread T, a call to lll.*`finishRead()` *by T returns* `false` *if a call to lll.*`startLayoutChange()` *completed since the previous call by T to lll.*`finishRead()` *or to lll.*`startWrite()`.

In other words, a read operation is made aware of a concurrent or previous layout change at `finishRead()`, and returns `false` to initiate a recovery (i.e. restart of the read operation).

By Invariant 5, all `threadStates` are set to `LC` (*Layout Change*) when `startLayoutChange()` completes and the `baseLock` is held in exclusive mode. Therefore, if `startLayoutChange()` completed since the previous call by T to `finishRead()` or `startWrite()`, then a subsequent call by T to `finishRead()` must go through the slow path (i.e. call `slowSet()`) since `threadState` equals `LC` (*Layout Change*). After `slowSet()`, `finishRead()` will return `false` (at line 17).

Conclusion. These invariants ensure that writing and layout changes are mutually exclusive, and that reading of a value after a layout change started is aware of the layout change and can perform the needed recovery. Thus, we argue that the Lightweight Layout Lock provides the necessary synchronization to ensure correctness of concurrent strategies.

B SAFE PUBLICATION OF SHARED COLLECTIONS

An interesting semantic example arises when sharing a newly-created collection, and reading its contents from another thread. We discuss this example here, because it needs low-level implementation details to be explained. In our implementation, it is guaranteed that the other thread will observe values at least as recent as set just before sharing the collection. In other words, safe publication [Shipilëv 2014] is guaranteed for newly-created collections shared to other threads. Supp. Table 2 illustrates an example where safe publication ensures the same outcome as a global lock.

Table 2. An example illustrating safe publication.

	Example	GIL	Goal	Unsafe
7	Initial: `array = [1]`	**1**	**1**	**1**
	`array = [2]` | `print array[0]`	**2**	**2**	**2**
				0

Unsafe has the additional outcome **0**, because it does not guarantee safe publication and as a result might read the element before it is initialized to **2**. In Java, memory is guaranteed to be zero-initialized for allocations, so only zero values can be observed (including `null` for an `Object[]`). We consider **0** an out-of-thin-air value as it is never written by the program. The program only sets the first array element to **1** and **2**, but never to **0**. Therefore, we want to prevent this outcome.

Our implementation prevents observing **0** by having safe publication semantics for shared objects and collections. This is guaranteed by the process of sharing an object or collection, which involves having a memory barrier between marking the object or collection as shared and the actual assignment making the object or collection reachable from other threads. This process is detailed in Supp. Figure 18. Daloze et al. detail the write barrier but do not mention the memory

```
// assuming array is a variable shared between threads,
// the Ruby code array = [2] becomes:

RubyArray tmp = new RubyArray();
tmp.storage = new int[] { 2 };

tmp.shared = true; // mark as shared
Unsafe.storeFence(); // or just a StoreStore barrier
array = tmp; // publish the array to other threads
```

Fig. 18. Pseudo-code illustrating the implementation of the write barrier sharing a collection.

barrier, which was an omission in that paper. The barrier is also required to ensure other threads correctly see this object or collection as shared.

C RAW DATA TABLE FOR FIGURE 5

Table 3. Raw data table for Figure 5, showing the throughput relative to `Local` with 1 and 44 threads, as well as the scalability with 44 threads normalized to the 1-thread performance of the configuration.

Benchmark	100% reads			90% reads, 10% writes			50% reads, 50% writes		
	Throughput		Scaling	Throughput		Scaling	Throughput		Scaling
Threads	1	44	44	1	44	44	1	44	44
`Local`	1.000	1.000	45.1	1.000	1.000	39.2	1.000	1.000	42.4
`SharedFixedStorage`	1.030	0.997	43.7	1.036	1.079	40.8	0.997	0.897	38.1
`VolatileFixedStorage`	0.351	0.330	42.4	0.255	0.300	46.1	0.137	0.154	47.6
`LightweightLayoutLock`	0.259	0.247	43.0	0.162	0.181	43.9	0.079	0.081	43.7
`LayoutLock`	0.230	0.223	43.6	0.164	0.185	44.4	0.078	0.080	43.7
`StampedLock`	0.189	0.181	43.4	0.136	0.001	0.4	0.062	0.000	0.1
`ReentrantLock`	0.045	0.001	0.9	0.038	0.001	0.9	0.039	0.001	0.9

Figure 5 shows how each configuration scales from 1 to 44 threads but it is hard to observe some data points due to overlapping lines. Therefore we provide the most relevant data in Supp. Table 3. As the table shows, the throughput of `ReentrantLock` is very low even on a single thread. `StampedLock` does not scale when there are concurrent writes, and is even significantly slower on 44 threads than on 1 thread. Other locks scale well on these 3 benchmarks, but the throughput of `Local` and `SharedFixedStorage` is many times faster than other configurations.