JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Benoit Daloze, M.Sc.

Submitted at
Institute for System Software

Supervisor and
First Examiner
o.Univ.-Prof. Dr. Hanspeter Mössenböck

Second Examiner
Dr. Jeremy Singer

Co-Supervisor
Dr. Stefan Marr

October 2019

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

Doctoral Thesis
to obtain the academic degree of
Doktor der technischen Wissenschaften
in the Doctoral Program
Technische Wissenschaften

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

We are in the multi-core era. Dynamically-typed languages are in widespread use, but their support for multithreading still lags behind. Specifically, their runtime systems either lack support for parallelism completely or they have many drawbacks, such as not scaling, having a significant overhead for single-threaded applications and exposing implementation-level races to the user. Runtime systems use sophisticated techniques to efficiently represent the dynamic objects and versatile collections present in dynamically-typed languages. One of the major reasons for not supporting well parallelism is these techniques are often unsafe in multi-threaded environments.

In this thesis, we present a solution for efficient and thread-safe shared-memory parallelism in dynamically-typed language runtimes. First, we design a new efficient and thread-safe object model for dynamic languages, only requiring synchronization when writing to objects shared between multiple threads. Accesses to objects reachable by only a single thread remain unsynchronized and maintain single-threaded performance.

Then, we devise a thread-safe and scalable design for collections in dynamic languages, which we apply to array-like and dictionary-like collections. We use multiple levels of synchronization to optimize for the usage pattern of each collection instance. Collections operations are unsynchronized until the collection becomes reachable by multiple threads. For array accesses within bounds, we use a new mechanism for global synchronization, avoiding overhead for this usage pattern. For the general case, we introduce a new lock that provides scalability for both read and write accesses, while supporting the versatile operations that dynamic languages provide and retaining the sophisticated techniques for representing collections efficiently in memory.

We evaluate our work in TruffleRuby, a state of the art high-performance implementation of the Ruby programming language. We show that our thread-safe objects and collections do not cause any overhead when they are reachable by only a single thread. Our experiments show that our approach scales linearly for object, array and dictionary accesses, achieves the same scalability as statically-typed languages for classic parallel algorithms, and scales better than other Ruby implementations on Ruby workloads.

Kurzfassung

Wir befinden uns in der Multicore-Ära. Dynamisch typisierte Sprachen sind weit verbreitet, aber ihre Unterstützung für Multi-Threading hinkt immer noch hinterher. Insbesondere ihre Laufzeitumgebungen haben entweder keine vollständige Unterstützung für Parallelität oder viele Nachteile, wie z.B. mangelnde Skalierung, Laufzeitnachteile von Programme die in einem einzigen Thread ausgeführt werden oder Race-Conditions der Implementierung die Einfluss auf die Benutzeranwendung haben können. Laufzeitumgebungen verwenden ausgefeilte Techniken, um die dynamischen Objekte und vielseitigen Collections-Frameworks von dynamischer Sprachen zu implementieren. Einer der Hauptgründe für die mangelnde Unterstützung von parallelen Sprachfeatures ist, dass diese Techniken in Multi-Threading-Umgebungen oft zu Abstürzen führen.

In dieser Arbeit stellen wir eine Lösung für die effiziente und threadsichere Ausführung von Programmen mit gemeinsamen Speicherbereichen in Laufzeitumgebungen für dynamisch typisierte Sprachen vor. Zuerst stellen wir ein neues, effizientes und threadsicheres Objektmodell für dynamische Sprachen vor. Dieses Objektmodell erfordert Synchronisierung nur für Schreiboperationen auf Objekte die von mehreren Threads genutzt werden. Zugriffe auf Objekte, die nur von einem einzigen Thread erreicht werden können, bleiben unsynchronisiert und erreichen die gleiche Performance wie single-threaded Programme.

Dann entwickeln wir ein threadsicheres und skalierbares Design für Collections in dynamischen Sprachen, die wir auf Array- und Dictionary-Datenstrukturen anwenden. Wir verwenden mehrere Synchronisationsebenen, um das Nutzungsmuster jeder Collection-Instanz zu optimieren. Die Collections-Vorgänge werden unsynchronisiert ausgeführt, bis die Instanz von mehreren Threads erreichbar wird. Für Array-Zugriffe innerhalb der Speichergrenzen verwenden wir einen neuen Mechanismus zur globalen Synchronisation, der den Overhead für dieses Nutzungsmuster vermeidet. Für den allgemeinen Fall stellen wir eine neue Sperrtechnik vor, die Skalierbarkeit für Lese- und Schreibzugriffe bietet und gleichzeitig die vielseitigen Operationen unterstützt, die dynamische Sprachen bieten. Diese Lösung unterstützt die bereits existierenden Techniken, Collections effizient im Speicher abzulegen.

Wir bewerten unsere Arbeit in TruffleRuby, einer hochmodernen, leistungsstarken Implementierung der Programmiersprache Ruby. Wir zeigen, dass unsere threadsicheren Objekte und Collections keinen Overhead verursachen, wenn sie nur von einem einzigen Thread erreichbar sind. Unsere Experimente zeigen, dass unser Ansatz linear für Objekt-, Array- und Dictionary-Zugriffe skaliert. Die Lösung erreicht die gleiche Skalierbarkeit wie statisch typisierte Sprachen für klassische parallele Algorithmen, und skaliert besser als andere Ruby-Implementierungen auf Ruby-Workloads.

Acknowledgments

I would like to thank Stefan Marr, my co-supervisor, for the good times spent doing this research, be it passionate discussions, helping with the general research direction or just nitpicking on English grammar and statistics.

I would like to thank my supervisor Hanspeter Mössenböck for continuously encouraging me to progress on this thesis, for fruitful discussions about many of the details of this research and for reviewing in great details my papers and this thesis.

I would like to thank Jeremy Singer for graciously accepting to be my second examiner, attending my defense and reviewing this thesis.

I would like to thank the Institute for System Software at JKU Linz for providing a nice environment for research in close communications with the industry, allowing to work on exciting and large-scale projects.

I would like to thank Oracle Labs for providing funding for this PhD and investing in great projects such as GraalVM and TruffleRuby.

I would like to thank David Leopoldseder, Christian Humer, Christian Wirth and Josef Eisl for helping me with the process of submitting this thesis.

I gratefully acknowledge the contributions of the TruffleRuby team including Chris Seaton, Kevin Menard, Petr Chalupa, Brandon Fish and Duncan MacGregor for contributing to making TruffleRuby more compatible and more performant, which allowed me to run more experiments.

Contents

Chapter 1

Introduction

Dynamically-typed languages are popular for their productivity, expressive syntax and convenient core libraries. The first implementations for dynamic languages were not very efficient, but over the years, academia and industry have improved the performance of dynamic languages considerably. There has been a growing body of research improving the efficiency of dynamic languages with various techniques [5, 7, 11, 12, 16, 17, 98, 99]. The popularity of dynamically-typed languages motivated the creation of more efficient implementations with notable examples existing for JavaScript (SpiderMonkey [59], V8 [28], JavaScriptCore [2], Nashorn [62]), for Python (PyPy [6], Jython [41]), as well as for Ruby (TruffleRuby [81], JRuby [65], Rubinius [72]).

A major research challenge in computer science is to take advantage of modern hardware and specifically of multi-core processors, which are now available in many devices. Using multi-core processors, programs can run multiple tasks in parallel, speeding up the computation and taking better advantage of the resources. To run in parallel, programs need support from their runtime systems. While there has been much effort to improve the single-threaded performance of dynamically-typed languages, we observe that their multi-threaded performance is still poor. Specifically, the support for shared-memory parallelism from runtime systems is still lacking and has many drawbacks, such as not scaling, having a significant overhead for single-threaded applications or exposing implementation-level races to the user. In this thesis, we present a solution for efficient and thread-safe shared-memory parallelism in dynamically-typed language runtimes.

There are essentially three approaches to parallelism in dynamic languages. First, some languages like Clojure, Erlang and R use mostly immutable data structures. Therefore, they avoid the concurrent mutation of data. Since data structures cannot be modified, these languages require a different programming approach for parallelism, keeping this constraint in mind. Second, some approaches to parallelism such as JavaScript Workers, Lua Lanes and Racket Places provide mutable data structures but do not allow sharing them between multiple threads.[1] This works around the problem of concurrent mutations, at the cost of requiring a different programming model and restricting communication between threads to only use immutable values. Finally, the third category is *shared-memory* parallelism with languages such as Python, Ruby, Perl and Common Lisp. These languages have to address the problem of concurrent mutations when using multithreading.

[1]Some of these approaches allow sharing mutable arrays of primitive types, but they all forbid sharing objects, highly restricting what can be shared between multiple threads.

1.1 Problem Statement

Dynamic languages with *shared-memory* parallelism need to address the problem of concurrent mutations. Other forms of parallelism in dynamic languages avoid this problem at the cost of being unable to share mutable data structures between multiple threads. They either require to deep-copy mutable data structures or to share only immutable data structures and use explicit communication to update data.

There are two levels for the problem of concurrent mutations: language implementation level and user code level. Synchronizing concurrent mutations at the user code level remains the user's responsibility, as it is the case in most shared-memory languages. We expect users to correctly synchronize access to the data structures they define and it is not the goal of this thesis to provide more synchronization mechanisms for the user. Users therefore need to define what set of operations should be synchronized and thus performed together atomically, e.g., by using locks. To the best of our knowledge, no language runtime can guess how to appropriately synchronize user code.

Instead, we focus on concurrent mutations at the language implementation level because this is where runtime systems differ, and where the decisions can lead to surprising and hard-to-explain results for the user. Specifically, we look at concurrent mutations of core data structures, that is data structures provided by the runtime system such as built-in objects, arrays and dictionaries. State-of-the-art runtime systems for shared-memory dynamic languages either always use synchronization for core data structures or they do not provide guarantees when using core data structures concurrently. Always using synchronization provides well-defined behavior but incurs a significant overhead. Runtime systems not providing guarantees when using core data structures concurrently expose data races from the implementation level to the user. For instance, writing to a field of a built-in object in these systems can be "lost" and have no effect instead of updating the value of that field. This can also happen for other core data structures such as arrays and dictionaries where concurrent operations might be lost, corrupt the data structure, throw exceptions, or even crash the program. We argue that these systems are unsafe, because they expose race conditions which do not exist at the language level. To limit such surprising behavior, these systems expect and rely on users to manually synchronize all operations on core data structures, including built-in objects, arrays and dictionaries. Manually synchronizing all operations on such core data structures is error-prone and cumbersome, as it requires to add synchronization around every access to every object or core data structure that is potentially accessed concurrently. Categorizing whether a core data structure might be accessed concurrently is difficult, and a single omission can result in surprising behavior. Finally, it also seems unexpected for users to have to manually synchronize simple operations like writing to an object field (e.g., `obj.field = value`), which at the language level looks thread-safe and well defined.

Current shared-memory dynamic language implementations work around the concurrent mutations problem of core data structures in an inefficient or unsafe manner. The reference implementations of Ruby and Python still use a *global interpreter lock*, which prevents parallel execution of user code in a single process. On the plus side, the global lock ensures safe concurrent mutations of core data structures. Alternative implementations often support parallel execution but address the concurrent mutations problem inefficiently or not at all. Some implementations, such as Jython, PyPy-STM [55], LispWorks and Racket's Futures [89], trade performance for safety and synchronize on *every* core data structure operation, leading to considerable slowdowns even for single-threaded applications and preventing parallel access to core data structures. In contrast to that, Nashorn, Rubinius, JRuby and SBCL trade safety for performance and provide core data structures without any safety guarantees when they are accessed from multiple threads, leading to, e.g., lost updates and implementation-level exceptions. In summary, the state of the art either addresses the concurrent mutations problem inefficiently, by synchronizing on all core data structure operations and introducing a significant overhead, or it does not address the problem at all, leading to unsafe and surprising behavior for dynamic language users.

Additionally, the focus of the research community on improving the performance of single-threaded programs in dynamic languages has resulted in optimizations that are not safe for multi-threaded programs, further complicating the concurrent mutations problem. For instance, state-of-the-art dynamic language implementations often use *storage strategies* [7] for arrays and dictionaries, a technique to represent these core data structures more efficiently. Storage strategies cause the representation of these core data structures to change at run time, introducing new races when these core data structures are used concurrently.

1.2 Scope

We are working on thread-safety at the language implementation level. Thus, we do not try to address user code level synchronization, which remains the user's responsibility, as in most systems. For instance, while access to object fields and operations on built-in arrays and built-in dictionaries are automatically synchronized in our approach, user-defined data structures need to be synchronized by the user. We are not aware of any system synchronizing user-defined data structures automatically.

One goal of this thesis is to contribute to ongoing work on memory models for dynamic languages, by encouraging these memory models to not just cover basic memory accesses, but also to provide stronger guarantees for core data structures of dynamic languages.

1.3 Scientific Contributions

To address the problem of concurrent mutations of core data structures, we designed a solution which avoids synchronization when unnecessary, synchronizes appropriately when needed, and enables parallel access to core data structures such as built-in objects and collections. Together, this enables shared-memory dynamically-typed languages to run in parallel with thread-safe and efficient data representations. In this section, we detail each individual contribution as a part of this global contribution.

Guest-Language Safepoints

We introduce guest-language safepoints, a synchronization mechanism that enables powerful features such as interrupting a thread to execute arbitrary code. By ensuring that all threads synchronize together at the beginning and the end of the guest-language safepoint, we can ensure that the code executed in the safepoint and its side effects are observed atomically by all threads before they continue executing user code. We also use our safepoints for advanced synchronization of core data structures. Guest-language safepoints rely on existing safepoints in virtual machines and therefore do not add any overhead to normal execution.

A General Approach for Efficient Synchronization Based on Reachability

Programming languages represent data as objects and collections. Collectively, we refer to objects and collections as *entities*. We introduce the idea to categorize entities based on their reachability for the purpose of efficient synchronization and thread-safety. We distinguish between *local* entities — that are reachable only by a single thread — and entities that are *shared* between threads. *Local* entities never need synchronization, since they can only be accessed by a single thread and not concurrently. Thus, only *shared* entities need synchronization. We track this status of *local* and *shared* at run time, by using a write barrier on shared entities. When entities become accessible by multiple threads, we mark them as *shared* and all further operations on these entities will use some form of synchronization, e.g., to ensure thread safety. Thus, we can infer which entities need synchronization and which entities do not, based on reachability. This allows us to avoid any overhead on entities used only by a single thread.

A Thread-Safe and Efficient Object Model for Dynamic Languages

Dynamically-typed languages typically support adding and removing fields of an object at run time. Therefore, they require specialized object representations for efficient access [11, 99]. We present a novel language-agnostic and thread-safe object storage model for dynamic languages that is based on these specialized object representations. Our object model guarantees that reading, updating, adding, and removing fields is thread-safe. By applying our approach of efficient synchronization based on reachability, our thread-safe object model incurs no overhead for accessing local objects. Furthermore, we design the synchronization such that there is no overhead for reading fields of shared objects, which is by far the most frequent operation on shared objects [42]. We only synchronize when writing to a shared object. To the best of our knowledge, we introduced the first thread-safe and efficient object storage model for dynamic languages.

A Thread-Safe and Scalable Design for Collections in Dynamic Languages

We present a novel approach to automatically synchronize operations on built-in collections. Our design retains single-threaded performance by applying our efficient reachability-based synchronization to collections. Thus, *local* collections do not use synchronization and therefore are as performant as thread-unsafe collection implementations. When a collection becomes *shared*, we change its representation and add synchronization to provide thread safety for all operations, including those growing or shrinking the collection. Furthermore, we design multiple levels of synchronization for *shared* collections to adapt to the usage pattern of each collection instance, to reduce the overhead of synchronization, and to improve the efficiency of parallel accesses. Therefore, when a collection becomes *shared*, it either uses minimal synchronization if the accesses do not change the structure and the size of the collection or uses full synchronization if the collection needs to be restructured. We apply this approach to two central collections in dynamic languages: arrays and dictionaries. We show that array and dictionary accesses scale linearly, while providing thread-safety and retaining performance when these collections are accessed by a single thread. Our approach supports storage strategies [7], which are key to achieving the best performance and a low memory footprint for collections in dynamic languages. Specifically, our approach correctly synchronizes internal structural changes that are required to switch between different strategies.

1.4 Technical Contributions

We implemented and evaluated our approach to synchronization of core data structures in TruffleRuby [81], a state-of-the-art implementation of the Ruby programming language. The author contributed 7,171 commits to TruffleRuby over 4 years, changing around 275,000 lines of code (not counting renames and imports of external projects). He made the second largest contribution to TruffleRuby by number of commits after the creator, Chris Seaton (7,934 commits) [26]. The author contributed to various areas of TruffleRuby in addition to thread-safety and parallelism, notably to improve compatibility and performance, which enabled TruffleRuby to run more benchmarks and Ruby programs.

TruffleRuby

TruffleRuby is an implementation of the Ruby programming language using the Truffle framework [100] and the Graal just-in-time compiler [98]. TruffleRuby focuses on performance and reaches performance far beyond other implementations of Ruby on classic and image-processing benchmarks [30, 52, 98]. In fact, it is competitive with state-of-art implementations of JavaScript and Python [52].

We implemented the novel techniques introduced in this thesis in TruffleRuby, to consider all concerns relevant for high-performance implementations, and to measure performance on a level that is competitive with the state of the art. Therefore, small inefficiencies are revealed quickly as the performance of TruffleRuby is nearly optimal and there is very little overhead compared to the state-of-the-art performance for other dynamic language implementations. Similarly, our approach was carefully designed with the just-in-time compiler in mind, as it is the main component to achieve high performance.

TruffleRuby is also highly compatible with the reference implementation of Ruby, MRI [54], which forced us to look at all aspects of a Ruby implementation, including interesting edge cases in the language design which often need considerable thought to arrive at a proper solution. This makes our solution applicable in practice for the popular Ruby dynamic language. This way Ruby programmers at large can benefit from our advances. Our implementations of guest-language safepoints and thread-safe objects are in fact now part of TruffleRuby, where they are available to any user of this open-source implementation [81].

TruffleRuby is distributed as part of GraalVM [29], a state-of-the-art virtual machine by Oracle Labs that is able to run various languages such as Java, Ruby, JavaScript, Python, R and LLVM bitcode. Therefore, our work has now been adopted in a mainstream virtual machine.

1.5 Supporting Publications

These publications are the base of this thesis and are all closely related to the thesis subject. The author is first author on all of these publications.

Techniques and Applications for Guest-Language Safepoints

Benoit Daloze, Chris Seaton, Daniele Bonetta, and Hanspeter Mössenböck. Techniques and Applications for Guest-Language Safepoints. In *Proceedings of the 10th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS'15, pages 1–10, 2015. [15]

This paper details the first contribution, and forms Chapter 3. The author contributed a substantial part of the ideas, the design, the writing, the evaluation and the implementation. The paper was written in collaboration with Chris Seaton and Daniele Bonetta. This paper is also part of Chris Seation's thesis [78]. Chris Seaton came up with the original idea and acknowledges that the author refined and improved the idea and implementation in a very substantial way.

The author came up with the idea of deferred actions (cf. Section 3.2), the application of the API for guest-language signal handlers (cf. Section 3.3.2) and the handling of blocking operations with `Thread.interrupt` (cf. Section 3.4.4).

Efficient and Thread-Safe Objects for Dynamically-Typed Languages

Benoit Daloze, Stefan Marr, Daniele Bonetta, and Hanspeter Mössenböck. Efficient and Thread-Safe Objects for Dynamically-Typed Languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages & Applications*, OOPSLA 2016, pages 642–659, 2016. [16]

This paper details the second and third contributions, and forms Chapter 4. The author contributed most of the ideas and the design, as well as the entire implementation and evaluation.

Parallelization of Dynamic Languages: Synchronizing Built-in Collections

Benoit Daloze, Arie Tal, Stefan Marr, Hanspeter Mössenböck, and Erez Petrank. Parallelization of Dynamic Languages: Synchronizing Built-in Collections. In *Proceedings of the 2018 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages & Applications*, OOPSLA 2018, 2018. [17]

This paper details the fourth contribution, and forms Chapter 5. The author contributed most of the ideas, the design and the writing. The author contributed the entire implementation and evaluation, except for the design and implementation of the Lightweight Layout Lock which is contributed by Arie Tal.

1.6 Additional Publications

These additional publications cover broader topics around performance, design of collection libraries and efficient string representation in dynamically-typed languages.

Cross-Language Compiler Benchmarking: Are We Fast Yet?

Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. Cross-Language Compiler Benchmarking: Are We Fast Yet? In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS 2016, pages 120–131, 2016. [52]

This paper compares the performance of 10 dynamic language implementations on 14 benchmarks, notably evaluating the performance of TruffleRuby compared to other implementations of dynamic languages. The author contributed to the rules for translating benchmarks between languages, analyzed the performance of TruffleRuby and identified bottlenecks on that implementation.

Few Versatile vs. Many Specialized Collections: How to Design a Collection Library for Exploratory Programming?

Stefan Marr and Benoit Daloze. 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, 2018. [50]

This paper analyses the collection libraries of 14 languages, and highlights relevant aspects for exploratory programming and dynamic languages. It proposes a design with just 3 collection types: sequences, maps and sets, which correspond relatively well to what many dynamic languages provide. The author contributed to the design of collection types and properties and the mapping of language collection libraries to these 3 types.

Specializing Ropes for Ruby

Kevin Menard, Chris Seaton, and Benoit Daloze. Specializing Ropes for Ruby. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, ManLang'18, 2018. [57]

This paper applies the rope data structure, a persistent representation of character strings, to the *mutable* Strings of the Ruby programming language and refines ropes with metadata such as encoding and code range (i.e., whether all bytes are < 128 for optimization purposes), and additional rope operations such as repetition. The author helped to shorten the paper to fit the required number of pages and to improve clarity.

1.7 Funding

Benoit Daloze was funded by a research grant from Oracle Labs. 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.

1.8 Thesis Structure

This thesis is structured as follows: Chapter 2 introduces some necessary background. Chapter 3 explains our mechanism of global synchronization, i.e., guest-language safepoints. Chapter 4 details the approach for efficient synchronization based on reachability as well as its application to objects in dynamic languages. Chapter 5 introduces our multiple levels of synchronization for collections as well as the application of reachability-based synchronization to collections in dynamic languages. Chapter 6 discusses the related work. Chapter 7 summarizes and concludes this thesis and Chapter 8 exposes avenues for future work.

Chapter 2

Background and State of the Art

This chapter presents the necessary background about concurrency in dynamic languages as well as optimizations for efficient data representation in dynamic languages.

2.1 Concurrency and Parallelism in Dynamic Languages

First, we give some context about concurrency in dynamic languages. While our proposed techniques are also applicable to other dynamic languages and collections, we focus on Ruby to allow for more concrete discussions of issues.

2.1.1 Concurrency in Ruby

Ruby applications use concurrency widely in production. For instance, the Ruby on Rails web framework [31] uses multithreading by default [75]. 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. For example, concurrently appending to an `Array` can *throw implementation-level exceptions* with JRuby and Rubinius. Furthermore, Rubinius does not synchronize for object field accesses, leading to, e.g., field writes being lost and having no effect. Consequently, developers using JRuby or Rubinius need to change their Ruby programming 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 5.8.7 discusses an example of additional synchronization for JRuby that 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 [44, 84, 85]. Even though Ruby does not specify that collections should be thread-safe, developers rely on such characteristics.

TruffleRuby [81] is a Ruby implementation on the JVM using the Truffle framework [100] and the Graal compiler [98]. Initially, objects and collections were not thread-safe in TruffleRuby and could exhibit issues similar to JRuby and Rubinius. We addressed those problems in this thesis, by making objects thread-safe in Chapter 4 and collections thread-safe and scalable in Chapter 5 while maintaining efficiency.

Concurrent Ruby [18] is an external library that provides thread-safe variants of the `Array` and `Hash` collections. However, this requires the use of different collections for sequential and concurrent code (e.g., `Array.new` and `Concurrent::Array.new`), which is error-prone and inconsistent with the notion of dynamic languages having few but versatile collections [50]. Furthermore, these collections do not scale as they use an exclusive lock per instance.

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

2.2 Object Representations

Dynamic languages often provide object models with features similar to dictionaries, allowing developers to dynamically add and remove fields. Despite providing a very convenient programming abstraction, standard dictionary implementations such as hash tables are a suboptimal run-time representation for objects because they incur both performance and memory overhead. Accessing a hash table requires significantly more operations than accessing a field in a fixed object layout, as used by Java or Smalltalk. Moreover, the absence of static type information in dynamically-typed languages can cause additional overhead for handling primitive values such as integers and floats.

2.2.1 Self Maps

The Self programming language was the first to solve these issues by introducing *maps* [11]. With maps, the runtime system collects metadata describing how object fields are used by an application. Based on such metadata, a fixed object representation for an object can be determined, which enables optimal direct field access via a simple offset instead of an expensive hash-based lookup. Since the language supports dynamic changes to an object's fields, this fixed object representation is an optimistic, i.e., speculative, optimization. To avoid breaking the language's semantics, object accesses thus still need an additional step that confirms that the object has a specific *map* describing the current set of object fields. Thus, a field access first checks whether the object's *map* is the map that was used in the optimized code. If this test succeeds (as in most cases where the *map* at a given code location is stable), the read or write instruction can be performed directly. The read or write instruction accesses the field with a precomputed offset, which is recorded in the *map*, and can be inlined in the optimized machine code.

Verifying that a given object matches a given map is implemented with a simple pointer comparison, which can typically be moved out of loops and other performance-critical code. This approach has proven to be very efficient in Self [11] and has inspired many language implementations (e.g., PyPy [5], and Google's V8 [93]).

2.2.2 The Truffle Object Storage Model

Truffle [100] is a language implementation framework for the Java Virtual Machine (JVM). It is based on the notion of self-optimizing AST interpreters that can be compiled to highly-

```
class DynamicObject {
  // maps fields to storage locations
  Shape shape;

  // an object's storage locations
  long prim1;
  long prim2;
  long prim3;
  Object object1;
  Object object2;
  Object object3;
  Object object4;
  // stores further fields
  long[] primExt;
  Object[] objectExt;
}

int readCachedIntLocation(DynamicObject obj) {
  // shape check
  if (obj.shape == cachedShape) {
    // this accesses `obj.prim1'
    return cachedLocation.getInt(obj);
  } else { /* Deoptimize */ }
}
```

Figure 2.1: Sketch of Truffle's object storage model and a method reading a field.

optimized machine code by the Graal compiler [97], which uses partial evaluation. Language implementations using Truffle include JavaScript, R, Smalltalk, Python and Ruby [29].

The Truffle object storage model [99] is part of the Truffle framework, and can be used by language implementers to model dynamically-typed objects. The design of the object model is inspired by SELF's *maps*. Additionally, it introduces specializations for fields based on types. These type specializations avoid boxing of primitive types since the Java Virtual Machine imposes a strict distinction between primitive and reference types. The Truffle object model allows for an arbitrary number of fields by combining a small number of fixed field locations with extension arrays for additional fields.

Figure 2.1 depicts the `DynamicObject` class that provides the foundation for Truffle's object model. Consider an object `objA` with a field `f` in a dynamic language. This object is represented by an instance of `DynamicObject`. It has a specific shape that contains the metadata about field types and how to map field `f` to a concrete field of `DynamicObject`. As a usage example, if `f` is only seen to contain `long` values, it is mapped to a free `primN` storage location. If all of them are in use, `f` is mapped to an index in the `primExt` extension array instead.

When `objA.f` is assigned a value of a different type than its current one (e.g., changing from `long` to `Object`), the previously allocated `primN` storage location for the field `f` is no longer appropriate. Thus, the assignment operation needs to determine a shape that maps `f` to a storage location that can contain objects, such as `object1`. In case `objA` already has four other fields that contain objects, `f` needs to be stored in the `objectExt` extension array. Thus, the write to `f` will cause an update of the `shape` and might require the allocation of a new `objectExt` array to hold the additional storage location.

With the Truffle object storage model, objects of dynamic languages can be represented efficiently on top of the JVM. However, for small objects this model leads to a memory overhead compared to an exact allocation. For dynamic languages this is an appropriate design choice, because the model provides the support for dynamically-changing object shapes without requiring complex VM support.

Truffle offers a specialization mechanism by which the AST of the executing program adapts to the observed input. This mechanism allows Truffle to speculate on shapes and types to optimize for the specific behavior a program exhibits. In combination with the Graal just-in-time compiler [97], object accesses using the Truffle object storage model are then compiled to efficient native code.

2.3 Collection Representations

Collections in dynamically-typed languages may typically hold any value, whether it is a primitive integer or an object, i.e., there are no generic types. First, we explain how collection libraries in dynamically-typed languages differ from their statically-typed counterpart. Then, we explain more efficient ways to represent collections in dynamic languages.

2.3.1 Collections in Dynamic Languages

The design of collection libraries in dynamically-typed languages is quite different from the design of collection libraries in statically-typed languages. Through a survey of collection libraries for 14 different languages [50], we observed that dynamic languages have generally less collection types in their standard library, but those collection types are more versatile. Typically, dynamic languages provide three central collection types: a *sequence* abstraction (array-like or list-like), which can also be used as a vector or a stack, a *dictionary*-like abstraction and a *set*-like abstraction. Sometimes, the set-like abstraction itself is not part of the collection library, and the dictionary-like abstraction is used instead. This is for example the case for Ruby which only has 2 built-in collections: Array and Hash.

2.3.2 Storage Strategies for Collections

Dynamically-typed languages have historically suffered a performance and memory overhead for collections due to the use of either boxing or value tagging for primitive values such as integers and floats.

Bolz et al. [7] 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 an array, a dictionary, or a set adapts at run time based on the values stored in it. Collections that hold only integers can encode them directly in memory. However, if a collection uses the dynamic-typing property of the language, and 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 [7].

The flexibility and efficiency gains of storage strategies led to their adoption in high-performance dynamic language implementations such as PyPy [6], V8 [12], 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.

2.4 Synchronization Mechanisms

In this last section of the background, we discuss the Layout Lock, which is essentially an extended version of the read-write lock.

2.4.1 Layout Lock

Cohen et al. [13] proposed the Layout Lock, a scalable synchronization 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 [19, 101]. The Layout Lock design strongly depends on the assumption that read operations are very frequent, write operations are less 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 collections of dynamic languages. Unfortunately, for the versatile collections of dynamic languages such as arrays or dictionaries, we cannot make the assumption that layout changes are rare, in fact they can happen frequently. For instance in Ruby, it is common to create an empty array from a literal `[]` expression[1] and then to append elements to it, or even to use it as a stack, which triggers frequent layout changes. Therefore we designed our own variant of the Layout Lock in Section 5.6 to synchronize built-in collections.

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

Chapter 3

Guest-Language Safepoints

In this chapter, we present our mechanism for global synchronization, guest-language safepoints. This enables many interesting possibilities based on the fact that at any time, a thread can request all threads to synchronize, perform an arbitrary action while all threads are stopped, and then let the threads continue their execution. This mechanism for global synchronization is notably used for thread-safe collections in Chapter 5.

Guest-language safepoints are based on the traditional virtual machine safepoints. Virtual machine safepoints are a mechanism that allows one thread to suspend other threads in a known state so that runtime actions can be performed without interruption and with data structures in a consistent state. Many virtual machines use safepoints as a mechanism to provide services such as stop-the-world garbage collection, debugging, and modifications to running code such as installing or replacing classes. Languages implemented on these virtual machines may have access to these services, but not directly to the underlying safepoint mechanism.

We show that safepoints have many useful applications for the implementation of guest languages running on a virtual machine. We describe an API for using safepoints in languages that were implemented under the Truffle language implementation framework on the Java Virtual Machine and show several applications of the API to implement useful guest-language functionality. We present an efficient implementation of this API, when running in combination with the Graal dynamic compiler. We also demonstrate that our safepoints cause zero overhead with respect to peak performance and statistically insignificant overhead with respect to compilation time. We compare this to other techniques that could be used to implement the same functionality and demonstrate the large overhead that they incur.

3.1 Introduction

A virtual machine (VM) is a collection of services that support a running program at a higher level of abstraction than a physical architecture provides. This may include services such as automatic memory management through garbage collection, dynamic optimization through just-in-time compilation, as well as debugging, instrumentation, dynamic code loading and reloading.

To provide many of these services, the VM needs to be able to pause the running program in order to inspect and modify its state. Even in a single-threaded environment this can require coordination between the running application and the VM, as the trigger for the VM's action may be an external source such as attaching a remote debugger. However the problem becomes significantly more complex when there are multiple running application threads that all need to be coordinated with the VM. One example of a service that needs to coordinate application threads with the VM is the garbage collector. With the exception of some highly specialized collectors, a garbage collector will at some point need to pause all application threads to update the heap when objects are moved and collected. Conventional architectures and system libraries for threading such as `pthreads` generally do not provide a native mechanism to pause a running thread, so the VM must provide this itself by adding code to the application thread. Such a mechanism must have very low overhead. Reusing existing synchronization primitives such as locks would mean that application threads would continually be using these locks even though they are infrequently required. The implementation must also have low latency. When threads are requested to pause for garbage collection there should not be a long delay until the collection can start. In addition to being just paused, those threads must be paused in some kind of consistent state where modifications to the heap will not conflict with the work of the application thread.

3.1.1 Safepoints and Terminology

The common solution to the problem of VM-level synchronization and coordination is a technique called *safepoints* (sometimes also called *yieldpoints* [46]). A ***safepoint*** is a point in an application thread where it is safe to *pause* its execution. The VM and its services have special knowledge of safepoints, and can use them to perform several VM-level operations.

The word ***safepoint*** refers to several components of the system, therefore we disambiguate the meaning by qualifying the term. We define ***VM safepoint*** as the time span during which host threads are paused by the VM in their respective safepoints. The alternative — our contribution — is a ***guest-language safepoint*** or simply *guest safepoint*, the time span during which guest threads are paused by the guest-language implementation. For clarity, we will refer to the location within an instruction stream where any of these safepoints can be entered as a ***safepoint check*** rather than just *safepoint*.

3.1.2 Guest-Language Safepoints

What we contribute is not a mere transposition of VM safepoints to guest languages. This already exists to some extent with Truffle Assumptions [98] or JSR 292 SwitchPoints [68], which we will detail later as possible underlying mechanisms.

What we propose is an API which allows us to interrupt *any guest-language thread* in order to run *arbitrary code*, with low latency and zero overhead on peak performance. In contrast, VM safepoints are restricted to run predefined code in only a subset of the threads.

Applications for this API are numerous and very powerful: efficient inter-thread communication, signal handling in an existing thread, enumeration of all live objects, call stack examination, always-available debugging, and even deterministic parallel execution.

3.1.3 Implementation Context

Our implementation has been developed as part of the open source TruffleRuby project [81]. JRuby [65] is an existing effort to implement the Ruby language on top of the JVM using conventional techniques such as bytecode generation and the invokedynamic instruction. Truffle is a framework for writing self-optimizing AST interpreters on top of the JVM [100], and can be seen as an alternative to the conventional approach used in JRuby. The Truffle framework is normally used in combination with the GraalVM [97] — a modification of OpenJDK that includes the Graal dynamic compiler and other additions. As Graal is written in Java, Truffle can use it like a library, creating, manipulating and compiling the compiler's intermediate representation. Truffle's Assumption is used to implement guest safepoints, meaning that all compiled guest-language code is invalidated when a guest safepoint is triggered and needs recompilation later. For our current use-cases, this is not critical as guest safepoints are triggered only on exceptional and rare events.

3.1.4 Contributions

The contributions presented in this chapter are:

- A simple but powerful API for guest-language safepoints that can be conveniently used in languages implemented on top of a virtual machine.
- Example applications of this API to implement existing and new features in the Ruby language.
- An efficient implementation of this API that has provably zero overhead on peak performance and statistically insignificant overhead on compilation time.

3.2 An API for Guest-Language Safepoints

This section describes our novel API for using safepoints in a guest-language implementation. What we propose is a mechanism that pauses guest-language threads at guest safepoint checks and allows us to interrupt *any thread* to run any *safepoint action*, that is arbitrary code written in the host or the guest language. This is an extension to the functionality of VM safepoints as they are restricted to run predefined code in only a subset of the threads. An API for such a mechanism has many application opportunities. Our API is composed of two main parts, illustrated in Listing 3.1.

```
// Guest-language threads call
poll();

pauseThreadsAndExecute(threads, -> {
    // Action to run in every thread.
    // All threads wait for each other to complete their actions.
});

pauseThreadsAndExecuteLater(threads, -> {
    // Deferred action to run in every thread,
    // after the thread has exited the guest safepoint.
});
```

Listing 3.1: API for guest-language safepoints.

The first part of the API is used to respond to safepoint requests from other threads. To this end, application threads make a call to the `poll()` method at every location that is a *guest safepoint check*. These calls are inserted in the guest-language implementation code and not in the application code. For example, a guest-language method will call `poll()` once at the start of each method, and again within every loop[1]. As will be described in Section 3.4 and later evaluated, the number of calls to `poll()` does not affect peak performance. Only threads controlled by the guest-language implementation must use `poll()`. Other threads do not need to know about guest safepoints. The polling threads should not perform blocking operations that cannot be interrupted through normal JVM thread interruption, otherwise guest safepoints could be delayed. We will discuss these limitations in Section 8.1.

The second part of our API is for triggering a guest safepoint and for asking the threads to run the given *safepoint action*. We call the thread requesting the guest-language safepoint the *initiator* thread since it initiates the whole process (activating the safepoint guard, interrupting other threads, sending them the action, etc).

The list of threads affected by a safepoint action is a subset of the threads running guest-language code. However, for all our use cases, we either want to interrupt all guest-language threads or just a single one. Therefore, example usages of our API below will use either a single thread or `allThreads` to mean all guest-language threads.

Different safepoint actions require different forms of synchronization between the affected threads. In any case the *initiator* thread has to wait for all affected threads to reach the guest safepoint checks. We distinguish whether the safepoint *action* is run inside the guest safepoint or immediately after the application thread leaves the guest safepoint and resumes normal execution — in which case we call it a *deferred action*. When we want to run the *action* within the safepoint we call `pauseThreadsAndExecute()`. This method accepts a list of threads to be paused and a lambda expression to be executed in the guest safepoint by each thread. All affected threads wait until all actions have been run to completion. Running the action in the guest safepoint ensures that no guest-language thread runs anything else than the *safepoint action*, which can be used to prevent uncontrolled modifications of the heap, similarly to a VM safepoint.

When the initiator does not need these properties, it calls `pauseThreadsAndExecuteLater()` for a deferred action. The action runs after the guest safepoint has finished and the application has left the safepoint, resuming normal execution concurrently with other threads. This is useful for actions that run guest-language code such as callbacks which may themselves use safepoints or run for a longer time. We still have the guarantee that the action will be executed by each thread before the thread returns from `poll()` and before the thread continues executing its original code.

It is worth noting that `pauseThreadsAndExecute()` can be expressed in terms of `pause-ThreadsAndExecuteLater()` by adding a barrier synchronization at the end of the provided action. However, having a separate method reveals the intent and the desired properties in a better way.

These higher-level concepts are illustrated in Figure 3.2. The guest safepoint actions that are run in each thread are shown with other threads waiting until all are ready to run their action, and until all are finished. Deferred actions are shown running after the main action, concurrently with other threads running normal code. The figure also shows that guest-language safepoints use the host VM safepoint mechanism as well as the host VM deoptimization mechanism to trigger a guest-language safepoint. Those two concepts are detailed later in Section 3.4.

[1]As a concrete example, the TruffleRuby interpreter performs a call to `poll()` only at seven locations across the whole TruffleRuby code base which implements the majority of a large, complex existing language. These locations were the common nodes for method preludes and loop constructs.

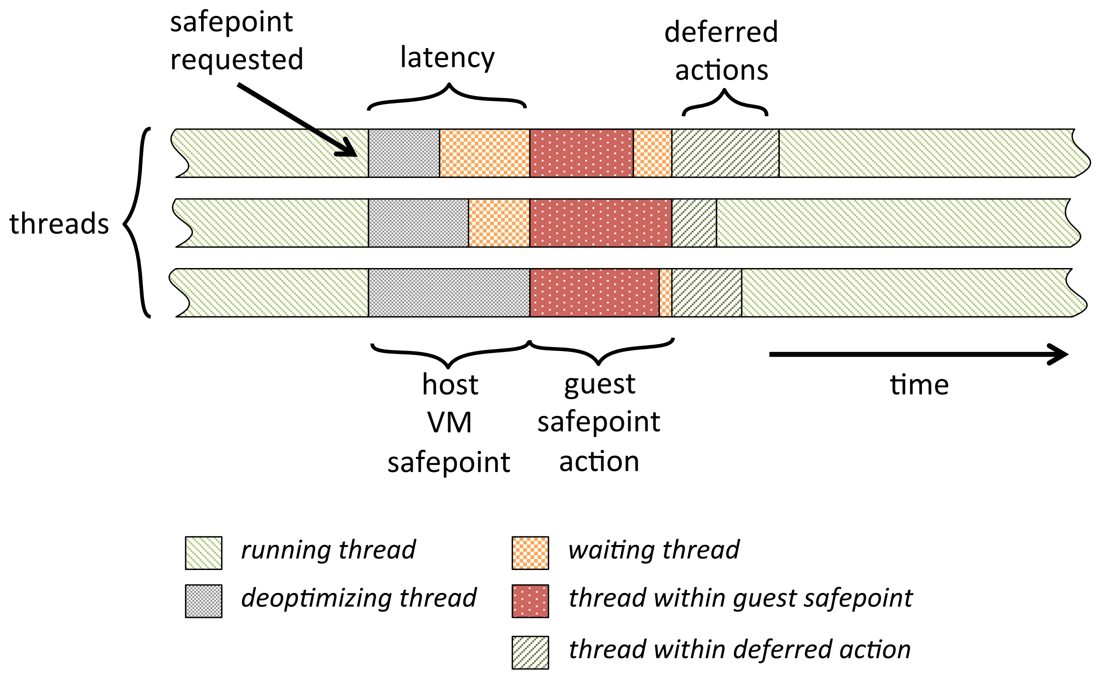


Figure 3.2: Phases of a guest-language safepoint

3.3 Applications

In this section we describe how we have applied our API in TruffleRuby. We also discuss another application currently under development in our Truffle-based JavaScript engine [69].

3.3.1 Inter-Thread Communication

Some languages allow a thread to cause some action to run on another thread. This is quite a good fit for our API as we can run any action in any thread. For example, a frequent requirement is to be able to stop a thread, e.g., because it is blocked or the task is no longer needed. In Ruby this functionality is provided by the `Thread#kill` method. To implement this in TruffleRuby, the first thread initiates the safepoint:

```
pauseThreadsAndExecuteLater(targetThread, -> {
    throw new KillException();
});
```

The action (the lambda in the call) will be run on the target thread in the next call to `poll()` where it will throw the exception.

We use an exception as the target thread should not just immediately die — the language semantics are that any cleanup actions such as `ensure` blocks (Ruby's equivalent of Java's `finally`) should be run first. This means that even if there was a primitive to just kill the underlying thread it would still not be appropriate.

Any `ensure` clauses will be run to free resources, until the exception terminates the thread execution by reaching the bottom of the execution stack. We use the deferred version of the API call because throwing an exception may cause arbitrary Ruby code to be executed, which is not designed to run inside a guest safepoint. In Ruby, it is also possible to raise an arbitrary exception in another thread with `Thread#raise(exception)`. This is just a variant of the previous example, in which the exception is provided by the user.

3.3.2 Guest-Language Signal Handlers

Signal handling on a traditional JVM typically spawns a new thread per signal, for example when using the `sun.misc.Signal` package. This is rather inefficient as it needs to spawn a new thread for every signal, which incurs some latency as the thread needs to start up, be scheduled and the threads to interrupt need to react to the notification. Furthermore, the newly spawned thread has no generic way to pause or interrupt other threads. With our API we implement a mechanism for signals that is closer to how C handles them, i.e., by running the signal handler on top of a thread's call stack (in `poll()`) and returning to normal execution when finished. This allows an application to respond in a more timely manner to signals and to run the handler in a well-known thread. This is in turn very useful to implement Ruby signals as the semantics require the possibly user-supplied trap handler to run in the main thread. For example, the SIGINT signal handler is defined by Ruby code and runs in the main thread, performing the appropriate action to interrupt it such as stopping the running server or raising an `Interrupt` exception.

3.3.3 Enumerating Live Objects

Heap walking, that is enumerating all live objects in a VM has always been a challenge to implement efficiently. While its utility for general applications might be questioned, it is often useful for debugging, especially when searching for memory leaks. It is one of the most powerful features a language might have and can, for example, help in upgrading a live system by migrating its objects to a new version. In Ruby, access to heap walking is provided by the `ObjectSpace.each_object` method. Implementing such a feature requires exclusive access to the heap so that objects are not created or destroyed while the list of live objects is created. It also requires access to the stack of all running threads. Our API provides the functionality to pause all threads and by running an action on each thread we can access all of their stacks. The existing Truffle API, unrelated to safepoints, already provides a mechanism to access values on the guest-language stack.

```
Set<Object> liveObjects;
pauseThreadsAndExecute(allThreads, -> {
    synchronized (liveObjects) {
        visitCallStack(liveObjects);
    }
});
```

The safepoint action then consists of walking the stacks of all threads as well as the thread-local memory in order to look for root pointers, similar to what the marking phase of a GC would do. Each thread adds the objects it can reach to a common set, which is given to the caller once all threads have completed the action.

3.3.4 Examining Call Stacks

One simple way of finding out what a program is doing is to inspect its threads' call stacks. `jstack` does this for Java programs, but its implementation requires VM support and so it is normally not possible to implement the same functionality for a guest language.

Using our API, we implemented a version of `jstack` for TruffleRuby. We added a VM service thread that listens on a network socket. Our equivalent of the `jstack` command sends a message to this socket, and the service thread uses our safepoint API to run an action on all threads that prints the current guest-language stack trace.

```
pauseThreadsAndExecute(allThreads, -> {
    printRubyBacktrace();
});
```

3.3.5 Debugging

Proper debugging support is probably amongst the most useful tools a programming language can provide. In almost all other platforms, including HotSpot, debugging comes with some overhead. This may be significant enough that a production system cannot be debugged.

In previous work [79] it was demonstrated that by reusing VM safepoints a debugger can attach and remove breakpoints in a running program with zero overhead until the breakpoints are triggered. In that work the debugger was in-process and could only be used by altering the application code and adding, e.g., `Debugger.enter`, to enter the debugger. Using our guest-language safepoints we can extend that functionality to allow a debugger to be attached remotely. We reused the same VM service thread as before that listens on a network socket. When a message is received, the service thread runs a safepoint action on the main thread telling it to enter the debugger, from where breakpoints can be added and removed and the program can be inspected and continued.

```
pauseThreadsAndExecuteLater(mainThread, -> {
    enterDebugger();
});
```

3.3.6 Deterministic Parallel Execution

A less conventional usage of safepoints can be found in the context of deterministic parallel programming models, i.e., programming models that guarantee deterministic parallel execution.

One example of such models is the RiverTrail [34] parallel API for JavaScript, an array-based data-parallel programming model providing implicit parallelism for array-based operations (e.g., via the `Array` built-in object). To enforce deterministic parallel execution, the RiverTrail runtime has to make sure that functions are not performing any "unsafe" operations during parallel execution. This is enforced via the following model of execution:

- Functions can read data from every object that is in their scope at the moment the parallel computation is started. Functions can also freely write to local variables, but they are not allowed to write to objects that are potentially in the scope of other threads as well.
- Parallel functions that attempt to modify an object potentially shared with other functions will cause the parallel execution to *bailout* to sequential mode. In this way, deterministic execution is enforced via sequential execution. In this case, parallel execution is aborted and

a single thread (usually the first thread trying to modify the shared object) is responsible for completing the rest of the computation.

This model of execution is called *temporal immutability* [53], and is implemented in the Truffle JavaScript engine [69] using our safepoint API.

In particular, every time a function performs an object field read operation, the `poll()` operation is invoked to ensure that no other thread wrote to this object (that is, to ensure that parallel execution is safe). Once a thread attempts to perform an unsafe access that will lead to a bailout (e.g., writing to a shared object), it will request a safepoint. In the safepoint action, the thread will kill all the other threads, and complete the computation sequentially, taking over the work of the aborted threads, if any.

A similar deterministic parallel programming model called Deterministic Parallel Ruby [47] has also been proposed in the context of Ruby. In contrast to RiverTrail, the Deterministic Parallel Ruby model does not enforce deterministic parallel execution using a bailout protocol, but uses a "debug" mode that dynamically checks for non-deterministic execution paths. An alternative implementation of the model could use our safepoints API to check for deterministic execution at very low overhead.

3.4 Implementation

We now describe the implementation of guest-language safepoints, first from a conceptual point of view and then gradually in more details. Figure 3.2 shows how the implementation details work together.

3.4.1 A Flag Check

Conceptually, to implement the proposed API, all threads need to regularly perform a check in `poll()`, for instance by reading a flag, which tells if the thread should enter a guest-language safepoint. A thread must be able to change the result of that check when it calls `pauseThreadsAndExecute()` or its variant.

A simple but high-overhead implementation could use a global volatile boolean variable, as shown in Listing 3.3. The variable needs to be volatile (this work was done with Java 8, so there is only plain or volatile access) as the JVM memory model allows non-volatile fields to be read once and the value reused unless there is a synchronization point. In practice this may mean that a method containing an infinite loop may only actually read the field once, and the check within the loop could just use a cached value. A thread running the infinite loop would never detect the guest safepoint.

For simplicity when describing the implementation, we remove the `threads` parameter and instead assume that there is a condition in the action to decide if it should be executed by the current thread.

We reset the flag as soon as all affected threads have reached the guest safepoint because the action could potentially call `poll()` and we would not want to keep entering the guest safepoint if there is no need.

```
volatile boolean guestSafepoint = false;

void poll() {
    if (guestSafepoint) {
        // enter guest safepoint
        // wait for other threads to enter the safepoint
        // execute action
        // wait for other threads to complete the action
    }
}

void pauseThreadsAndExecute(Action action) {
    guestSafepoint = true; // notify other threads
    // wait for other threads to enter the safepoint
    guestSafepoint = false;
    // execute action
    // wait for other threads to complete the action
}
```

Listing 3.3: Simple implementation of the API using a volatile flag.

One key limitation of a volatile flag (and the whole point of the `volatile` modifier) is that it prevents the compiler from performing some optimizations such as caching the value instead of performing multiple reads. When our Ruby code is compiled and optimized by Graal we generally inline a very large number of methods, as in Ruby all operators are method calls. This means that we are likely to end up with a large number of volatile reads in each compilation unit — at least one for each inlined method — and the compiler will not attempt to consolidate them, as this is exactly what the volatile modifier is for. Therefore, we would rather optimize the guest safepoint polls by reusing the existing VM safepoints.

3.4.2 Truffle and Graal

The Truffle language implementation framework provides an abstraction named `Assumption` to model these checks reusing VM safepoints. An `Assumption` provides the `isValid()` method to check its validity and an `invalidate()` method to invalidate the assumption.

The Graal dynamic compiler intensifies calls to `Assumption#isValid()`. To do so, it requires the receiver `Assumption` object to be a constant at compile time. This is usually achieved at the source level by annotating the field holding the `Assumption` with `@CompilationFinal`. This allows the compiler to inspect the actual `Assumption` object and know whether it is valid. In our case, the assumption is always valid when compiled by Truffle and Graal as it is replaced with a new and valid assumption before recompilation. The calls to `isValid()` are then simply replaced by their compile-time value (`true` in our case). This means that a condition such as the one in Listing 3.4 as well as the branch are omitted *from the compiled code* as the branch is known to never be taken. Finally, the compiler registers the assumption object with the generated code, such that the VM knows which generated code depends on a given assumption.

When `invalidate()` is called on an assumption, the compiled code depending on the assumption will be deoptimized (or marked as invalid) and all threads executing that code will transfer to the interpreter. The dependent code is known thanks to the previous registration. The VM will then trigger a VM safepoint, pausing all threads to deoptimize the dependent code.

```java
// The assumption to implement the guest safepoint
@CompilationFinal Assumption assumption =
    Truffle.getRuntime().createAssumption();

void poll() {
    if (!assumption.isValid()) {
        enterGuestSafepoint();
    }
}
```

Listing 3.4: Implementation of `poll()` with an `Assumption`.

This is how guest safepoints effectively reuse VM safepoints. Threads depending on the assumption (all guest-language threads in our case) will run in the interpreter once the assumption is invalidated. When running in the interpreter, calls to `isValid()` are actually performed and not omitted, returning `false` as the assumption is now invalid. In our case, this will cause threads to enter the guest safepoint with `enterGuestSafepoint()`.

In Listing 3.5, we show the core of `pauseThreadsAndExecute()` with an assumption. `barrier()` is a barrier to synchronize all threads in the guest safepoint. `interruptOtherThreads()` calls `Thread.interrupt()` on each thread in the guest safepoint except the current thread.

The code is simplified in that the `synchronized` modifier is not enough to invoke the `pauseThreadsAndExecute()` method concurrently. Instead, exclusive access to trigger a guest safepoint needs to be obtained in an interruptible way, calling `poll()` when interrupted.

Our implementation in TruffleRuby, in the class `SafepointManager`, is slightly more complex as we need to handle a few additional arguments to pass the execution context correctly. Our implementation, as the examples shown above, also does not accept a list of threads for the `pause` methods but rather lets threads register and unregister with guest safepoints. When we only want to affect a subset of the threads, we use a condition in the action. This is an intended restriction as currently the HotSpot JVM only provides “global” VM safepoints affecting all threads and therefore we need to pause all threads anyway.

3.4.3 Existing Implementation of VM Safepoints

In this section we describe how VM safepoints are implemented in our host VM, the OpenJDK JVM, on which GraalVM is based. A necessary property for VM safepoints is that they can be reached by all threads in a small amount of time (the latency). Once a VM safepoint is reached, the initiator thread is guaranteed to have exclusive access to all heap memory and VM data structures. In OpenJDK [67], VM safepoints are implemented using different techniques based on the kind of code each thread is running — Java bytecode in the interpreter, just-in-time compiled code or native library code.

OpenJDK has two bytecode interpreters, the simple C interpreter and the standard template interpreter. The former one is only used on architectures where the template interpreter is not supported. The C interpreter checks for safepoint requests by reading a volatile variable at method entries, return instructions and loop *back edges*. These locations are chosen for safepoint checks as that is where the code might *go back* to execute the same instructions again and

```java
// The assumption to implement the guest safepoint
@CompilationFinal Assumption assumption =
    Truffle.getRuntime().createAssumption();
volatile Action safepointAction;
volatile Thread initiator;

synchronized void pauseThreadsAndExecute(Action action) {
    safepointAction = action;
    initiator = Thread.currentThread();
    assumption.invalidate();
    interruptOtherThreads();
    enterGuestSafepoint();
}

void enterGuestSafepoint() {
    // wait for all threads to reach the safepoint
    barrier();

    // renew the assumption
    if (Thread.currentThread() == initiator) {
        assumption = Truffle.getRuntime().createAssumption();
    }
    // wait for all threads to see the new assumption
    barrier();

    try {
        safepointAction.run();
    } finally {
        // wait for all threads to finish the action,
        // unnecessary for deferred actions
        barrier();
    }
}
```

Listing 3.5: Implementation of `pauseThreadsAndExecute()` with an `Assumption`.

therefore where it spends most of the time. This is in fact similar to our flag check approach in Section 3.4.1 but at the VM level.

When a safepoint is requested in the template interpreter, the templated code of the affected threads is patched to contain an additional indirection checking for safepoints before executing normal behavior. This is achieved by replacing the normal dispatch table — the mapping between bytecodes and instructions — with a dispatch table calling back to the VM for checking safepoints and then resuming execution in the normal dispatch table. Once the safepoint is reached, the code is patched again to use the normal dispatch table to remove the overhead of this indirection.

In the machine code generated by one of the just-in-time compilers (client, server and also Graal for the GraalVM), checking for safepoint requests is achieved by performing a read on a special safepoint polling memory page. It is not the value of that read which is important but its side effects that are significant. To cause the thread to enter a safepoint, the permissions on the page are changed so to disallow read access to that page. This provokes an access violation on the next read, which gets reported as a SIGSEGV or SIGBUS signal (Linux, BSD) or an Exception (Windows). The corresponding handler then enters the safepoint if the faulty read location was the polling memory page.

To describe the specific instruction used for a safepoint check, we take the concrete example of the common AMD64 architecture. Instructions for other architectures are similar. The chosen instruction is `test`, which reads an address in memory, performs a bitwise conjunction and then sets flags. As this is not a branch instruction, no prediction is needed and the instruction does not modify any general purpose registers, meaning it can be pipelined efficiently by the processor. The locations to add VM safepoint checks is decided by the compiler and is entirely independent from the number and location of calls to `poll()` using our API. VM safepoint checks are normally inserted once in each generated machine code function (which could include multiple guest-language methods, or less than a guest-language method such as just a loop body), and once inside each loop. However, optimizations can remove some of these. For example, in a loop of a known number of iterations and with few instructions the body, safepoint checks may be removed because only a finite number of instructions can execute there before the loop is exited. Finally, when a thread is performing a blocking operation or similarly running native code (e.g., via JNI), it is considered as already in the safepoint as it may not modify the VM internals or the Java heap. On exit of such a blocking operation, the thread must wait until the safepoint action is complete. This is problematic for guest-language safepoints and we discuss it in the next section.

3.4.4 Running Code in any Thread

The existing VM safepoints alone are not expressive enough for our purpose because they do not allow paused threads to execute user-defined code before returning to normal execution. Having regular calls to `poll()` solves that issue.

While reusing VM safepoints provides an efficient implementation, they also do not pause *all* threads. Instead they consider some blocked threads to have reached the VM safepoint because they cannot manipulate the VM internals or the Java heap. This is insufficient for guest-language safepoints as we want to run arbitrary code in *any* of the threads participating in the guest-language safepoint. We discuss in this section how to circumvent this problem.

We identify three major cases in which threads might be blocked. The first case is a blocking call, such as sleeping for some time, trying to acquire a lock, waiting on a condition or waiting for another thread or process. Most of these blocking calls already provide a way to be interrupted, for instance via `Thread.interrupt` throwing an exception in the blocked thread. These calls are idempotent such that restarting them later behaves as if they were not interrupted. We can therefore interrupt blocked guest-language threads when initiating a guest safepoint. As we do not know which threads are blocked in such a case, we call `Thread.interrupt` on all guest-language threads. When the blocking call is interrupted, we `poll` and then restart the operation.

Another case is blocking IO. Many of these operations are not idempotent, so we need to be careful when restarting them and consider the partial work already performed. Some IO calls can already be interrupted like blocked calls above. If they are idempotent, the solution described above just works. It is also possible to interrupt some blocking IO calls with native signals. The standard implementation of Ruby sets up an empty signal handler for `SIGVTALRM` for this purpose, using `sigaction()` *without* the `SA_RESTART` flag, and that way is able to interrupt blocking system calls such as `read()` and `write()`. TruffleRuby uses the same technique to interrupt blocking system calls. Another solution is to simulate blocking IO with asynchronous IO. Asynchronous IO typically allows interruption and may be restarted. In Java this can be achieved via select-able channels. Finally, as blocking IO operations typically do not depend on the thread running them, we can spawn some host-language threads, invisible to the guest language, to actually perform the blocking IO. Guest-language threads send their requests and wait for the completion synchronously. This waiting operation is much easier to interrupt.

The third case is executing native code in which it might take an arbitrary amount of time for it to finish, if ever. This becomes a serious problem if native code is used for long operations. In practice, many usages of native code might prove non-problematic to interrupt if they do not perform long operations before calling back to the VM, which is typical for JNI and guest-language native extensions as they will likely interact often with the guest-language entities. However, this would impact latency.

In the context of Truffle, some of this "native code" (e.g., Ruby C extensions running on top of TruffleC [30]) is actually also executed by the same VM. In this case, that execution is not considered to be native code by the VM and it is possible to add calls to `poll()` in that program like in the main guest-language implementation. If none of that works, running native calls in separate threads remains a possibility. Ruby C extensions also typically use a specific API when they perform a long-running blocking call (`rb_thread_call_without_gvl()`), to which a callback to unblock is passed by the caller.

3.5 Evaluation

To evaluate our implementation we used a total of 55 benchmarks, both common synthetic benchmarks such as *fannkuch* and *mandelbrot*, as well as a suite of benchmarks produced by taking key methods from a pair of Ruby libraries for manipulating images, `chunky_png` and `PSD.rb` [82]. These benchmarks do represent an extreme of computational intensity — not all Ruby programs are CPU bound — but they are also real code being run in production today. All experiments were run on a system with 2 Intel Xeon E5345 processors with 4 cores each at 2.33 GHz and 64 GB of RAM, running 64-bit Ubuntu Linux 14.04. Benchmarks are run until they reach a steady state, determined by looking at the range of values over a moving window, and then 10 sample iterations are measured. Problem sizes were configured so that their fastest iteration takes at least 10ms and timing calls do not dominate. An arithmetic mean of the samples gives us the reported time. Reported errors are the standard deviation. When summarizing across multiple benchmarks we report a geometric mean of both the sample and the error. Our experiments use version 0.6 of Graal and are made available in a fork of the TruffleRuby repository [80].

3.5.1 Overhead on Peak Performance

We were interested in the overhead that TruffleRuby causes when our safepoint API is available but not used. This scenario is relevant because it is how we anticipate our safepoint mechanism to be used in most cases — ready to be used but usually not a common operation. Furthermore, we wanted to compare the overhead of our safepoint API implementation with that of alternative implementations.

We first measured the performance of TruffleRuby including references to the safepoint API (the *api* configuration). Then we removed these references and measured again (the *removed* configuration). As the API is simple and compact, we only had to modify 74 lines of code. We finally measured the performance of TruffleRuby when using alternative safepoint implementation techniques. For example, we tried an implementation that explicitly checks a volatile flag (the *volatile* configuration). We also tried an implementation of our API that uses a JSR 292 SwitchPoint object (the *switchpoint* configuration) [76].

Figure 3.6 shows our results. There is no statistically significant difference in performance between *removed*, *api* and *switchpoint*. This is because both *api* and *switchpoint* are reusing the existing lower level VM safepoints, whose overhead is already part of the VM. Thus, our API adds no penalty to peak performance, and so we refer to it as *zero-overhead*.

There is, however, a statistically significant difference between the performance of *api* and *volatile*. The geometric mean of the overhead for the *volatile* configuration is 25% ± 1.5%. Of course, this is the overhead on the whole benchmarks and if we could compare the overhead of just the safepoint `poll()` operations it would be much larger.

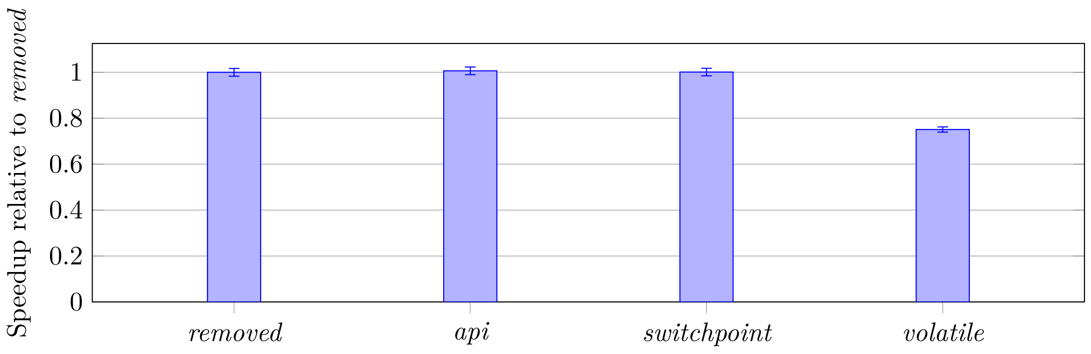


Figure 3.6: Geometric mean of peak performance over all benchmarks, normalized to the *removed* configuration. Higher is better.

3.5.2 Parallel JavaScript

To confirm the validity of our approach, we also performed an initial evaluation of the usage of safepoints in the context of the RiverTrail parallel programming model for JavaScript, as discussed in Section 3.3. Our implementation is based on the GraalJS JavaScript engine [69], and is currently under active development.

Safepoints are used in our RiverTrail prototype implementation to ensure that every read on potentially shared objects is consistent with other reads during parallel execution. This is needed to enforce the RiverTrail parallel model of execution, which permits reads to shared objects and writes to scope-local ones. The model also supports writes to shared objects, at the cost of parallel execution: when write operations are performed on objects that are potentially shared, parallel execution is aborted and the computation is completed in a single-threaded mode. Our API is used by every parallel thread to assert that parallel execution has not been aborted. As long as this is true, the threads will never enter the safepoint (i.e., they will never suspend). In the unlikely case of execution bailout, the aborting thread (i.e., the thread that is attempting to write to a potentially shared object) will initiate the safepoint operation, and will complete the parallel computation within the safepoint action, sequentially.

As done in the previous section, we have implemented safepoints by using a volatile check and a Truffle compiler assumption. The performance comparison of the two implementations is depicted in Figure 3.7, where we present the performance of five JavaScript benchmarks from the Ostrich benchmark suite [43], adapted to use the RiverTrail model. As expected, the overhead for the volatile flag check is considerably high. Since the volatile check is performed on read operations on potentially shared objects (i.e., on objects in the scope of multiple functions running in parallel), benchmarks that perform only thread-local computations are not affected by the volatile check. This is the case for the *Primes* and the *Mandelbrot* benchmarks in the figure. Conversely, benchmarks that rely on read-only shared state (such as *CRC*, *FFT* and *NBody*) benefit from our API.

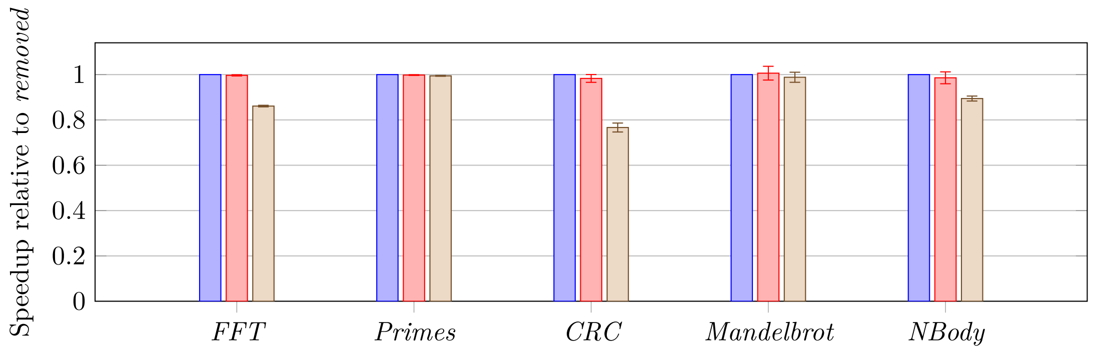


Figure 3.7: Geometric mean of peak performance for RiverTrail in JavaScript. The *removed* configuration (🟦) is compared against our *safepoint API* (🟥) and *volatile* (🟫). Higher is better.

3.5.3 Detailed Analysis

To further explain our results, we examined the machine code produced by the different configurations. We wrote a simple method that was run in an infinite loop to conveniently trigger the dynamic compiler. Our example code, written in Ruby and executed by TruffleRuby, is shown in Listing 3.8. It is intentionally kept small to improve the readability of the machine code and just contains a few arithmetic instructions in the loop body to better show the effect of the different configurations. Every arithmetic operator in Ruby (in this case, $+$, $\times$ and $<$) is a method call. Therefore, any of these operations is a call site and conceptually does the check for guest-language safepoints. The body of the loop simply adds 7 to the counter i at each iteration until i becomes greater or equal to n. The method is called with different arguments to prevent *argument value profiling*, which would eliminate the whole loop entirely in compiled code as it has no side-effects.

We present the machine code with *symbolic names* for absolute addresses and rename the specific registers to the uppercase *name* of the variable they contain. We use the Intel syntax, in which the destination is the first operand.

The machine code produced by the *removed*, *api* and *switchpoint* configurations is identical if we abstract from absolute addresses and specific registers, and is shown in Listing 3.9. This

```
def test(i, n)
  while i < n
    i += 1 + 2 * 3
  end
end

while true
  test(100, 200)
  test(200, 300)
end
```

Listing 3.8: Example code for detailed analysis of the generated machine code.

supports our measurements in that our API really has zero overhead, rather than just a low or difficult-to-measure overhead.

We can observe that the produced code is very close to the optimal code for such a loop. The operations in the body of the loop are reduced to a single addition thanks to constant propagation. There are only two redundant *move* instructions, which copy the variable i between registers I and I'. The value of i is copied in I' to perform the addition because, if the `add` overflows, the original value of i needs to be accessed by the code performing the promotion to a larger integer type. In theory, the promotion code could subtract the second operand from the overflowed i, but this is a fairly complex optimization to implement. The second *move* reunifies the registers.

The loop begins with a read on the safepoint polling page as described in Section 3.4.3, which checks for VM safepoints[2]. In the *api* and *switchpoint* configurations, this check is also used for guest-language safepoints at no extra cost. After the `mov`, we `add` 7 to i and then check for overflow with `jo`, an instruction that jumps to the given address if there was an overflow in the last operation. We then have the second `mov`, followed by the loop condition $i < n$. The order of the operands in the machine code is reversed, so we must jump to the beginning of the loop if n is greater than i.

```
loop:
  test    safepoint polling page, eax # VM safepoint
  mov     I', I
  add     I', 0x7
  jo      overflow
  mov     I, I'
  cmp     N, I # Is n > i ?
  jg      loop
```

Listing 3.9: Generated machine code for the *api*, *removed* and *switchpoint* configurations.

We now look at the machine code produced by the *volatile* configuration (Listing 3.10). The generated code is much larger. The loop starts by testing the condition $i < n$, again with reversed operands. The condition is negated, $n \leq i$, as the test is to break out of the loop. Otherwise we enter the loop body. The body begins with 4 reads of the volatile flag from memory, and if it is found to be 0, the code jumps to a deoptimization handler with `je`. Of these 4 checks, the first

[2]Actually, Graal moves this check out of the loop as it notices this is a bounded loop. We disabled that optimization for clarity.

is for the loop itself and the other 3 are for the different calls to +, + and × in the loop body. We then have the read on the safepoint polling page checking for VM safepoints. The remaining code is identical to Listing 3.9, except for the last two instructions. They perform a read on the volatile flag to check for guest-language safepoints at the call site of <, in the loop condition. If the flag is found to be valid, the control goes back to the beginning of the loop.

The 5 extra reads produced by the volatile flag are clearly redundant in the presence of the existing lower-level VM safepoints. They increase the number of instructions for the loop from 7 to 17, incurring a significant overhead as shown in Figure 3.6.

```
loop:
  cmp    N, I # Is n ≤ i ?
  jle    break out of loop
  cmp    VOLATILE FLAG, 0x0 # while loop safepoint
  je     deopt
  cmp    VOLATILE FLAG, 0x0 # i += 1
  je     deopt
  cmp    VOLATILE FLAG, 0x0 #      1 + 2
  je     deopt
  cmp    VOLATILE FLAG, 0x0 #          2 * 3
  je     deopt
  test   safepoint polling page, eax # VM safepoint
  mov    I', I
  add    I', 0x7
  jo     overflow
  mov    I, I'
  cmp    VOLATILE FLAG, 0x0 # i < n
  jne    loop
```

Listing 3.10: Generated machine code for the *volatile* configuration.

3.5.4 Overhead for Compilation Time

We also considered the time taken for dynamic compilation for benchmarks in different configurations by measuring the time taken to compile the main method from the *mandelbrot* benchmark. This is a relatively large method with a high number of method calls which need to be inlined and several nested loops, all of which add guest safepoints checks. We ran the benchmark 20 times for each configuration.

Figure 3.11 shows our results, with the columns showing the mean compilation time and the error bars showing the standard deviation. We found no significant difference in compilation time between *removed*, *api* and *switchpoint*. The compilation time for the *volatile flag* configuration appeared to be only slightly higher. All of the techniques explored require extra work in the compiler due to extra `poll()` calls, but this appears to be insignificant compared to the rest of the work being done by the compiler. The *volatile flag* is different to the other implementations in that the code is not removed in early phases and adds extra work to later phases of the compiler.

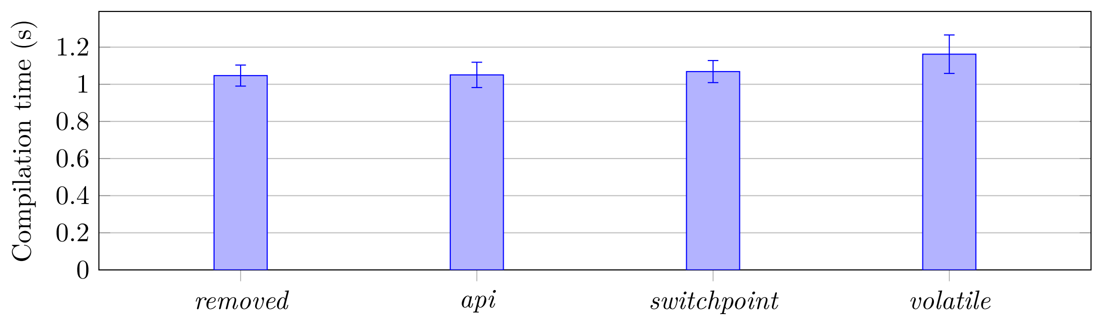


Figure 3.11: Mean compilation time for the *mandelbrot* method across different configurations. Lower is better.

3.5.5 Latency

Finally, we considered the time it takes for all threads to reach a guest-language safepoint after one thread requested it — the *latency*. Comparing the different configurations is not relevant here, as the costs can be primarily attributed to the VM safepoint latency and the necessary deoptimizations to enter the guest safepoint.

We ran the *mandelbrot* benchmark with a variable number of threads. After a steady state was reached due to JIT compilation, a separate thread requested all others to enter a guest safepoint.

Figure 3.12 shows our results, with the columns showing mean latency and the error bars showing the standard deviation. Latency was reasonable for most applications, responding to requests for a guest safepoint within about 1/100th of a second when running 8 threads. Variance was surprisingly high, which would make it hard to provide performance guarantees. Latency increases with the number of threads running concurrently, however the trend is sublinear. Deoptimization for multiple threads is a parallel task, so although multiple threads add extra work, they also add extra potential parallelism. For 1024 threads, a number well beyond what is typical for normal modern Ruby applications, latency was half a second. It should also be noted that due to deoptimization, the use of a guest safepoint brings peak performance down for some time before code can be reoptimized. Very frequent use of guest safepoints could cause an application to run entirely in the interpreter.

3.6 Summary

We have given the motivation for having a way to use safepoints as a guest-language implementation on a virtual machine, by showing how they can be used to implement several existing features in the Ruby language, and how they allow useful new features to be implemented. We described a design for this API that is high-level, and showed how we can implement it by reusing the underlying VM safepoint mechanism. We evaluated our implementation and found that it meets our requirements for low overhead on application threads — in fact it has zero overhead, which can be proved by inspection of the machine code. We found the overhead on compilation time to be statistically insignificant, and the latency to be low for a realistic number of threads. The design and implementation therefore meet the requirements that we had. Future work will look to move this API down into the language implementation framework and to continue to improve latency through improvements at the compiler level.

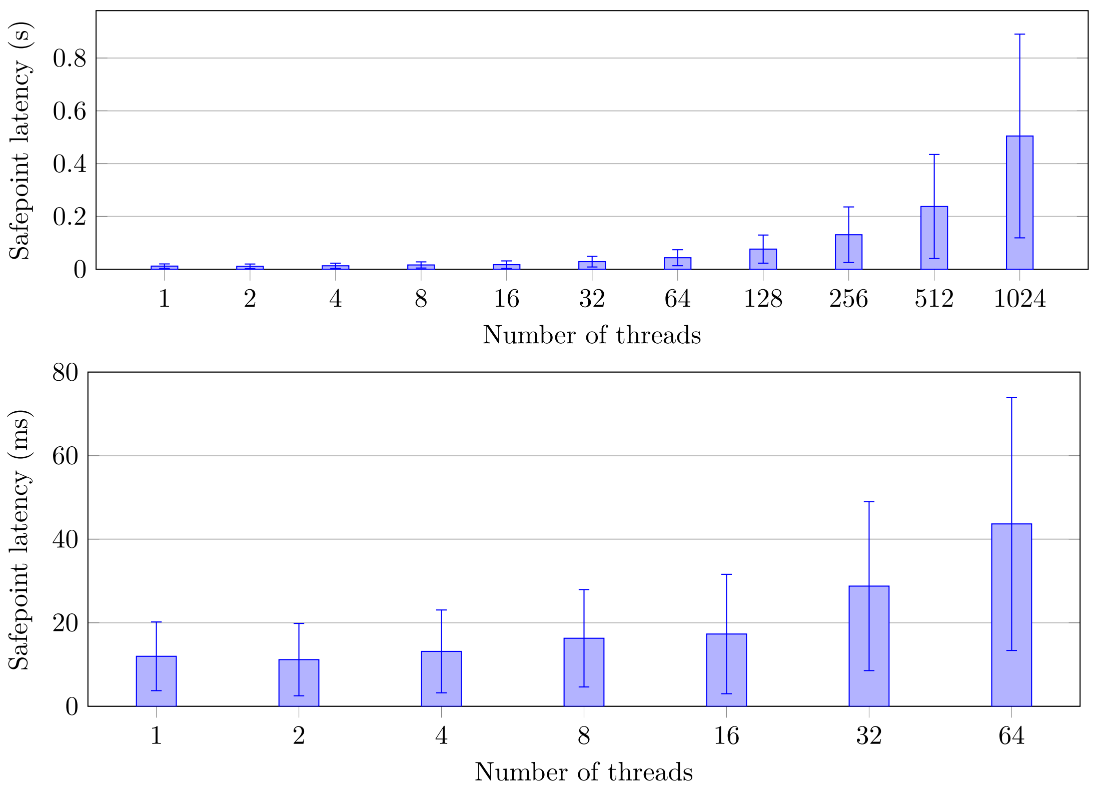


Figure 3.12: Safepoint latency for the *mandelbrot* for our implementation. Lower is better.

Chapter 4

Thread-Safe Objects

In this chapter, we present our approach for efficient synchronization of accesses to objects based on their reachability as well as its application to objects in dynamic languages, which support adding and removing fields at run time.

We are in the multi-core era. Dynamically-typed languages are in widespread use, but their support for multithreading still lags behind. One of the reasons is that the sophisticated techniques they use to efficiently represent their dynamic object models are often unsafe in multi-threaded environments.

This chapter defines safety requirements for dynamic object models in multi-threaded environments. Based on these requirements, a language-agnostic and thread-safe object model is designed that maintains the efficiency of sequential approaches. This is achieved by ensuring that field reads do not require synchronization and field updates only need to synchronize on objects which are shared between threads.

Basing our work on TruffleRuby, we show that our safe object model has zero overhead on peak performance for thread-local objects and only 3% average overhead on parallel benchmarks where updates of shared fields require synchronization. Thus, it can be a foundation for safe and efficient multi-threaded VMs for a wide range of dynamic languages.

4.1 Introduction

Dynamically-typed languages such as JavaScript, Ruby or Python are widely used because of their flexibility, prototyping facilities, and expressive power. However, a large majority of dynamic language implementations does not yet provide good support for multithreading. The increasing popularity of dynamically-typed languages has led to a need for efficient language runtimes, which in turn led to the creation of alternative implementations with notable examples existing for JavaScript (SpiderMonkey [59], V8 [28], Nashorn [62]), for Python (PyPy [6], Jython [41]), as well as for Ruby (JRuby [65], Rubinius [72]). However, most of the efforts have aimed at improving sequential performance. One of the key optimizations is the in-memory representation of objects based on Self's maps [11] or similar approaches [99]. By using maps (cf. Section 2.2.1), the runtime systems avoid using expensive dictionary lookups in favor of a more efficient representation of object layouts, leading to several benefits when combined with just-in-time compilation techniques, such as polymorphic inline caches [38].

Unfortunately, the focus on sequential performance has resulted in optimizations that are not safe for concurrent execution, and many language implementations have minimal or no support for parallel execution. The most popular implementations of Ruby and Python still limit parallelism using a *global interpreter lock*, while prominent JavaScript engines support only share-nothing models via memory isolation. This limits the support for concurrency and parallelism in modern multi-core machines, and leaves many potential benefits of using multithreading out of reach for dynamic languages. Implementations such as Nashorn, Rubinius, Jython, and JRuby have tried to address this issue with support for multi-threaded execution. Unfortunately, Nashorn and Rubinius provide object representations without any safety guarantees when accessing an object from multiple threads. Other engines, such as Jython and JRuby, trade performance for safety and synchronize on *every* object write, leading to considerable slowdowns even for single-threaded applications.

In this chapter, we introduce a novel technique to overcome the limitations of existing language runtimes. Our approach enables zero-overhead access on objects that are not shared between threads, while still ensuring safety when concurrent accesses occur. Safety is ensured without introducing any overhead for single-threaded programs, and with only 3% overhead on average for parallel programs. With this minimal performance cost, our approach guarantees that the object representation does not cause *lost field definitions and updates*, as well as *out-of-thin-air values* (cf. Section 4.2). These guarantees are of high practical value, because such problems can happen inside existing language runtimes despite application programs being seemingly free of data races. Thus, arguing that an application should use proper synchronization would be insufficient since the concurrency issues are caused by the actual implementation of a runtime's object representation.

Contributions To summarize, the contributions presented in this chapter are:

- a safety definition for adaptable object representations for multi-threaded environments,
- a thread-safe object storage model to make SELF's maps and similar object models safe for concurrent access,
- an approach to provide safety efficiently by only synchronizing write accesses to objects that are reachable by multiple threads,
- a structural optimization to efficiently update an object graph of bounded size by speculating on its structure,
- and an implementation of the optimizations and model for TruffleRuby, a state-of-the-art language runtime using the Truffle object storage model [99].

4.2 Safety in Existing Object Storage Models

Most of the existing object storage models for dynamically-typed languages are unsafe when used in combination with multithreading. We consider an implementation as *safe*, if and only if it does not expose properties of the implementation in form of exceptions, crashes, or race conditions to the language level. Thus, safety means that implementation choices do not have visible consequences at the language level. This includes the guarantee that the implementation behaves safely even in the presence of data races or bugs in the user program. In the following sections we detail the most relevant safety issues of today's object storage models.

4.2.1 State of the Art

Both SELF's *maps* (cf. Section 2.2.1) and the Truffle object storage model (cf. Section 2.2.2) have been engineered for single-threaded execution. As a result, in some situations, concurrent object accesses are unsafe. To the best of our knowledge, only a few language implementations use object representations that support accesses from multiple threads while still providing safety guarantees. Examples are Jython [41] and JRuby [65]. Since, for instance, JavaScript does not provide shared memory concurrency, V8's hidden classes [93] as well as Nashorn's `PropertyMap` [45] and SpiderMonkey's object model [61] do not provide safety guarantees for shared-memory multithreading. While Nashorn is not designed for multi-threaded use, it provides a convenient integration with Java that makes it easy to create threads and to expose them to JavaScript code. When used in such a way, programs can observe data races, exceptions, or even crashes, because the JavaScript object representation itself is not thread-safe [45]. We detail these issues in the remainder of this section, in which we use the terminology of the Truffle object storage model for consistency, but the same issues are also present in Self's maps as well as in derived variants.

4.2.2 Lost Field Definitions

The first safety problem is that concurrent additions of fields to an object can lead to only one of the fields being added. This is illustrated in Listing 4.1.

When two threads simultaneously add a new field to an object, they cause shape transitions, i.e., the object's internal shape is replaced with a new one describing the new object layout with the new field. In a concurrent scenario, only one of the two fields may be added, and the definition of the second field may be lost. The shape of an object is stored at the implementation level as a host field of the object, which we refer to as "shape pointer" for clarity (cf. Figure 2.1). When changing the shape, the access to the shape pointer is normally unsynchronized to avoid interfering with compiler optimizations. Thus, each thread will add its field to a separate new shape, and will then update the object's shape pointer without synchronization. This means that the first shape that was written into the object may be lost.

From the application's perspective, lost field definitions are inherently unsafe and not acceptable, because they are the result of an implementation choice. The program itself might even be free of data races, e.g., when the updates are done to different fields. However, it can still suffer from such implementation-level issues, which need to be avoided to guarantee correct program execution. Therefore, concurrent definitions must be synchronized to avoid losing fields.

```
obj = Foo.new # obj shape contains no fields
Thread.new {
  # (1) Find shape with a: {a}
  # (4) Update the object shape to {a}
  obj.a = "a"
}
Thread.new {
  # (2) Find shape with b: {b}
  # (3) Update the object shape to {b}
  obj.b = "b"
  # (5) obj shape is {a}
  obj.b # => nil (field b was lost)
}
```

Listing 4.1: The definition of field *b* can be lost when there are concurrent field definitions. The numbered comments indicate a problematic interleaving of implementation-level operations performed by the object model.

4.2.3 Lost Field Updates

Another race condition arises when the storage allocated to a given object needs to grow in order to accommodate new fields. Generally, objects can have an unbounded number of fields. Using a fixed memory representation thus requires a mechanism to extend the storage used for an object.

Assuming a state-of-the-art memory allocator, objects cannot be grown in-place, since they are allocated consecutively with only minimal fragmentation. Thus, an object cannot be grown directly. Instead, one of its extension arrays is replaced with a new array (cf. Section 2.2.2). This could cause updates on the old array being lost since they are racing with installing the new extension array. To avoid such lost updates on unrelated fields, which would not be data races based on the program's source code, proper synchronization is required. This is illustrated in Listing 4.2.

```ruby
obj = Foo.new
obj.a = 1
Thread.new {
  # (2) Write to old storage
  obj.a = 2
  # (4) Read from new storage
  obj.a # => 1 (update was lost)
}
Thread.new {
  # (1) Copy old storage [1], grow to [1,"b"]
  # (3) Assign the new storage to obj
  obj.b = "b"
}
```

Listing 4.2: The update to field *a* can be lost if growing the storage is done concurrently with the field update. The numbered comments indicate a problematic interleaving of implementation-level operations performed by the object model.

4.2.4 Out-Of-Thin-Air Values

In some unsafe object representations, it is possible to observe out-of-thin-air values [74], i.e., values that are not derived from a defined initial state or a previous field update.

This can happen if a memory location is reused for storing some other field's value. For instance, if a field is removed and its storage location is reused or if there are concurrent field definitions which both use the same storage. We illustrate the second case in Listing 4.3. When both fields are assigned the same memory location, it is possible that the value of field `b` is written first, then the update of field `a` completes (updating the value and the shape), and then the update of field `b` assigns the new shape. Any reader of field `b` will now read the value that was assigned to field `a`. As with the previous issues, this is a case that requires correct synchronization to avoid data races that are not present in the original program.

4.3 A Thread-Safe Object Model

To design a thread-safe object model without sacrificing single-threaded performance, a new approach to synchronization is required. Specifically, a safe object model has to prevent loss of field definitions, loss of updates, as well as out-of-thin-air values. To prevent these three types

```
obj = Foo.new
Thread.new {
  # (3) Find shape with a: {a @ location 1}
  # (4) Write "a" to location 1
  # (5) Update obj shape to {a @ location 1}
  obj.a = "a"
}
Thread.new {
  # (1) Find shape with b: {b @ location 1}
  # (2) Write "b" to location 1
  # (6) Update obj shape to {b @ location 1}
  obj.b = "b"
  # (7) Read location 1
  obj.b # => "a" (value of field a)
}
```

Listing 4.3: It is possible to get *out-of-thin-air* values when reading a field. When reading field b, the value of field a is returned instead, which was never assigned to field b in the user program. The numbered comments indicate a problematic interleaving of implementation-level operations performed by the object model.

of problems, the object model needs to guarantee that, even in the presence of application-level data races,

- any read of an existing object field returns only values that were previously assigned to that field,
- any read to non-existing object fields triggers the correct semantics of the language for handling an absent field such as returning a default value (like `nil`) or throwing an exception.
- a write to an object field is immediately visible in the same thread. The visibility of the write in other threads can require application-level synchronization.

Since current language implementations forgo thread-safety for their object models because of performance concerns, we strived for a synchronization strategy that provides safety and efficiency as equally-important goals. We designed a strategy that provides safety without requiring any synchronization when reading fields. This is an important property to avoid impeding compiler optimizations such as moving loop-invariant reads out of loops or eliminating redundant reads. Updates to objects or their structure, however, need synchronization. To avoid incurring overhead in single-threaded execution or for objects that are only accessible by a single thread, we use a synchronization strategy that is only applied to objects that are shared between threads. More precisely, our technique is capable of:

- reading object fields without any performance overhead, regardless of the object being shared or not,
- enforcing synchronization on the internal object data structures when an object is accessible by concurrent threads, i.e., when one thread performs a field update on a shared object.

This design is based on the intuition that objects that are shared between threads are more likely to be read than written. As a motivating example, Kalibera et al. [42, sec. 6.5] show that reads are 28× more frequent than writes on shared objects in concurrent DaCapo benchmarks. From a performance perspective, multi-threaded algorithms typically avoid working with shared mutable state when possible, because it introduces the potential for race conditions and contention, i.e., sequential bottlenecks. Moreover, manipulating shared mutable state safely requires synchronization and therefore already has a significant overhead. Thus, designing a synchronization strategy that has no cost for non-shared objects and for objects that are only read is likely to give good results for common applications.

The remainder of this section presents the main elements and requirements for this synchronization strategy.

4.3.1 Read-side Synchronization

As described in Section 4.2.4, reading a field of an object that is concurrently updated by some other thread is unsafe in existing object models because the other thread might cause a shape transition and as a result, the reading thread might see some other field's value, i.e., it might read an out-of-thin-air value. For a shape transition, the object's shape and one of its extension arrays would need to be updated atomically (cf. Section 2.2.2 and the `DynamicObject` class). This would, however, require synchronization on each object read and write access. Without synchronization, a read from a shared object can see a shape that is newer than what the object storage represents, or see an object storage that is newer than what the shape describes. This can happen because a field update might still be under way in some other thread, the compiler moved operations, or the CPU reordered memory accesses. The result would be that an undefined, i.e., out-of-thin-air, value might be read that does not correspond to the field that was expected. Synchronizing on every access operation, however, is very expensive, and needs to be avoided to preserve performance. Instead, we adjust the object storage to remove the need for read-side synchronization, as described in the next section.

4.3.2 Separate Locations for Pairs of Field and Type

In the presence of possible inconsistencies between object storage and shape, we need to avoid reading the wrong storage location (as it would produce out-of-thin-air values). We make this possible without synchronization by changing how the object model uses storage locations in the object storage. Since out-of-thin-air values are caused by reusing storage locations, we change the object model to use separate locations for each pair of *object field* and *type*.

By ensuring that storage locations are only used for a single pair of field and type, it is guaranteed that a read can only see values related to that field, and cannot misinterpret it as the wrong type. If a stale shape is used, the field description might not yet be present in the shape, and we will perform the semantics of an absent field, which is also acceptable with our safety definition.

If the shape has already been updated, but the storage update is not yet visible, the operation could possibly access a field that is defined in the new shape, but whose storage location does not exist in the old storage. To account for this case, we furthermore require that the object storage only grows, and is allocated precisely for a specific shape, so that the object storage has a capacity fitting exactly the number of storage locations. With this design, an access to such a non-existing storage location results in an out-of-bounds error, which can be handled efficiently to provide the semantics of an absent field.

Since we require that the storage only grows and storage locations are not reused, we cannot just remove the corresponding storage location when removing a field from an object. Instead, we must keep that storage location and migrate to a shape where the field is marked as "removed".

As a consequence of this design, it is guaranteed that any object storage location is assigned to a single field only, i.e., there is no reuse of storage locations even though fields can be removed or they might require a different storage location because of type changes. This ensures, for instance, that values are never interpreted with an inconsistent type or for the wrong field. Therefore, any data race between updating an object's shape and its extension arrays cannot cause out-of-thin-air values at the reader side.

While this strategy prevents out-of-thin-air values, it can increase the memory footprint of objects. This issue and its solutions are discussed in Section 4.5.2.

4.3.3 Ordering Access to the Shape and Storage

The above is enough to handle possible inconsistencies between the object storage and shape when only using extension arrays. However, it is not sufficient when using fixed pre-allocated storage locations (e.g., `object1` in Figure 2.1). If the shape has already been updated, but the storage update is not yet visible, it is possible to read from the pre-allocated storage location even though no value has been written to it yet. That would mean an object field read could return the default value of that field, which is unacceptable as it would be an out-of-thin-air value. In Java, the default value would be `null` for reference types and `0` (or `false`) for primitive types. For reference types, we could handle this by having a sentinel value to mark uninitialized fields, and when reading such a sentinel value, perform the semantics of an absent field. However, there is no way to differentiate user values from some sentinel value for primitives types, because all values are potentially user values.

To solve this, we order the access to the shape and the storage, by reading the shape with a load-acquire, and by writing the shape with a store-release. When reading a field, the shape should be read first, followed by accessing the storage. When writing a new field, the storage should be expanded first if needed, followed by setting the new value in the storage, and finally setting the shape. That way, the storage we read is always at least as recent as the shape, and this case of the shape being newer is removed entirely.

On the AMD64 architecture, normal loads and stores already have the semantics of acquire and release respectively, and therefore no extra memory barriers are needed for for the CPU. Barriers might still be needed to avoid that the compiler reorders these accesses.

4.3.4 Write-side Synchronization

Object writes need synchronization to prevent lost field definitions and lost updates, because writing to a field can cause a shape transition. For instance, when the value to be written is incompatible with the currently allocated storage location, the shape needs to be updated to describe the new location (cf. Section 2.2.2). For such shape transitions, the object storage needs to be updated, and one of the extension arrays potentially needs to be replaced, too.

Note that it is necessary to use synchronization for *all* write accesses, not just those changing the shape, as otherwise a concurrent field update and field definition could result in losing the update when only the field definition uses synchronization (cf. Section 4.2.3).

In order to keep the synchronization strategy simple, all forms of updates are synchronized by locking the corresponding object for the duration of the update. This sequentializes all updates to prevent lost field definitions and lost updates, achieving the desired safety.

To minimize the overhead of synchronization, the object model uses it only for objects that are shared between multiple threads. The next section explains how we distinguish between thread-local and shared objects.

4.4 Local and Shared Objects

This section introduces our approaches to distinguish local and shared objects and to make deep sharing efficient.

4.4.1 Distinguishing Local and Shared Objects

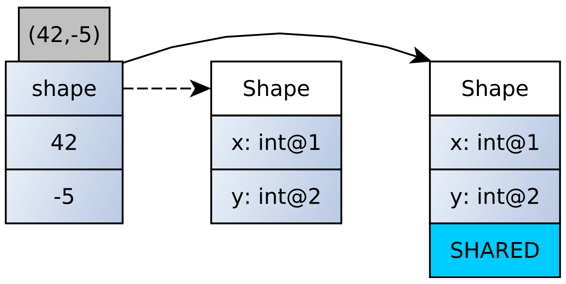


Figure 4.4: The shape of an object changes when it becomes shared between multiple threads. The figure illustrates a (x, y) coordinate object changing its shape when it becomes shared.

The synchronization overhead for object updates is only necessary for objects that are shared between multiple threads. If an object is local to a single thread, synchronization can be omitted. To perform synchronization only on objects that can be accessed by multiple threads, local objects need to be distinguished from shared ones. We do this by assigning different shapes to *local* and *shared* objects (illustrated in Figure 4.4), so that when an object becomes shared, its shape is changed to a *shared* variant which indicates that the object it represents is shared between multiple threads. Using different shapes allows us to reuse existing shape checks, which are already performed during object accesses, and to automatically choose the right update semantics without needing an additional test to know if an object is local or shared.

An object becomes shared the first time a reference to it is stored in a globally-reachable object. Globally-reachable objects are objects which can be reached from multiple threads. This includes an initial set of objects as well as all objects that over time become reachable from this initial set. The initial set is language-specific but typically includes global objects such as classes, constants, and generally data structures that are accessible by all threads. In Java, the initial set would also include objects stored in static fields. In Ruby, it would also include objects stored in global variables. A detailed overview of an initial set for Ruby is given later in Section 4.6.4. The conceptual distinction between local and shared objects based on reachability was first introduced by Domani et al. and is detailed in Section 6.2.1.

Note that tracking sharing based on reachability over-approximates the set of objects that are used concurrently by multiple threads. However, it avoids tracking all reads, which would be required to determine an exact set of shared objects. Shared objects also never become local again as this would require knowing when a thread stops referencing an object. Therefore, to maintain this set of globally-reachable objects during execution, we only need a write barrier on fields of already-*shared* objects, which is explained in the next section.

As a result, we can dynamically distinguish between local and shared objects, allowing us to restrict synchronization to objects that are accessible by multiple threads. The main assumption here is that for good performance, multi-threaded programs will minimize the mutation of shared state to avoid sequential bottlenecks, and thus, writing to shared objects is rare and synchronizing here is a good tradeoff between safety and performance.

4.4.2 Write Barrier

To track all shared objects, a write operation to a field of an *already shared* object needs to make sure that the assigned object is being *shared* before performing the assignment to the field, because this object suddenly becomes reachable by other threads. Not only the assigned object needs to be marked as shared, but also all objects that are reachable from it, since they become globally reachable once the assignment is performed. Therefore, *sharing* an object is a recursive operation. This is done by a write barrier as illustrated in Listing 4.5.

```java
void share(DynamicObject obj) {
  if (!isShared(obj.shape)) {
    obj.shape = sharedShape(obj.shape);
    // Share all reachable objects
    for (Location location : obj.shape.getObjectLocations()) {
      share(location.get(obj));
    }
  }
}

void writeBarrier(DynamicObject sharedObject,
                  Location cachedLocation,
                  Object value) {
  // (1) share the value if needed
  if (value instanceof DynamicObject) {
    share(value);
    Unsafe.storeFence(); // or just a StoreStore barrier
  }

  // (2) assign it in the shared object
  synchronized (sharedObject) {
    cachedLocation.set(sharedObject, value);
  }
}
```

Listing 4.5: Write Barrier for shared objects. Objects are marked as shared (1) before they are made reachable by other threads (2). The second step, i.e., the publication, is the assignment of the value to the object field of an already globally reachable object. In `share()`, the `getObjectLocations()` method returns all storage locations that contain references.

Note that sharing the object graph does not need synchronization as it is done while the object graph is still local and before it is made reachable by the assignment. However, a memory barrier is required between marking the object as shared and the actual assignment as shown in Listing 4.5, to prevent these two assignments to reorder. The reason is that the compiler could

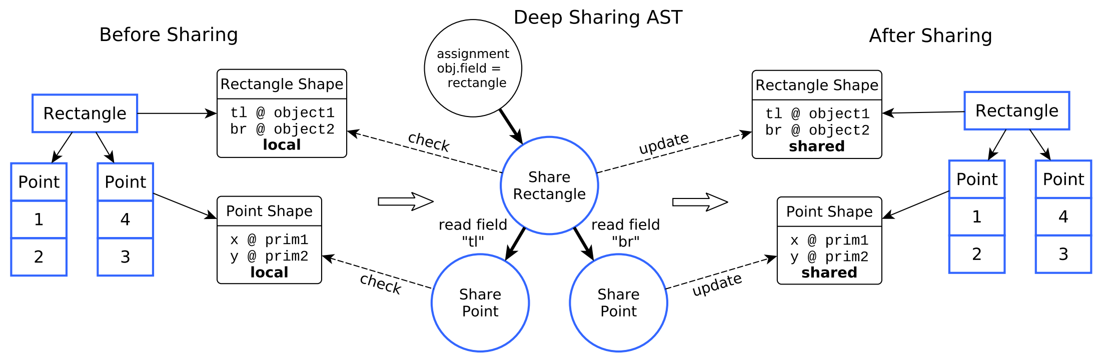


Figure 4.6: Deep Sharing of a Rectangle and two Points. The AST mirrors the structure of the Rectangle and its Points. The Share nodes check if an object matches the expected shape and update it to a shared variant.

otherwise move the write to `obj.shape` into the `synchronized` block[1] and then the compiler or hardware could reorder the write to `obj.shape` with the actual assignment since the two writes are to different fields. That could then lead to another thread observing the object with a local (non-shared) shape, which would be incorrect as the thread would not synchronize to access the object. This memory barrier also has the desirable effect to provide safe publication semantics [83], such that any thread reading from the fields of the object after it was shared will be guaranteed to observe values that are at least as recent as the ones that were set just before sharing the object. Other objects that are shared by the write barrier and are part of the object graph will also have safe publication semantics.

The write barrier decides what to do based on the type of the assigned value. If the value has a primitive type, it does not have fields and thus cannot reference other objects, so it does not need to be shared. If the value is an object, it needs to change its shape to a shared variant of it, unless the object is already shared.

When an object gets shared, the following actions are taken:

- the object shape is changed from the original non-shared shape to a shape marked as shared with the same fields and types.
- all objects reachable from the object being shared are also shared, recursively.

4.4.3 Deep Sharing

Sharing all reachable objects requires traversing the object graph of the assigned object and sharing it, as illustrated by the `share()` method in Listing 4.5. Such a traversal has a considerable run-time overhead. To minimize this overhead, we use the information provided in shapes to optimize the traversals of the object graph.

For optimal performance, the write barrier specializes itself optimistically on the shape of the value for a given assignment. We do so by creating an inline cache for the shape of the value.

[1]Under the Java Memory Model, plain memory accesses can be moved inside a `synchronized` block, but not outside. This is sometimes referred as the “roach motel” semantics of the Java Memory Model.

The shape of a value is expected to be stable at a certain assignment in the source code (i.e., only a few shapes are observed for most assignment sites), as the value is assigned to a specific field, and application code reading from that field typically has specific expectations on its type (e.g., it expects the read value to have a specific method).

The inline cache then contains information from the shape. Specifically, we cache the list of fields which can contain references and the offsets to read these fields efficiently. This way, we can check the shape once and read all fields by just reading from known offsets, instead of looking up the shape information each time, fetching the offsets, and then read the fields.

We also apply this optimization transitively, that is we not only look at the fields of the assigned value but also specialize on the shapes of the objects contained in those fields. We apply a depth limit to avoid traversing large object graphs. Objects beyond the depth limit are shared without specializing on their shape.

This optimization is done by building an inline cache structure that mirrors the structure of the object graph of the assigned object. For example, imagine that the object to be shared is a `Rectangle` described by two `Point` instances representing the top-left (`tl`) and bottom-right (`br`) corners. `Point` instances only contain numeric values and therefore do not need to propagate the sharing further. The inline cache structure is in our case a tree of 3 Share nodes, one for the rectangle and two for the two points, denoted by "Deep Sharing AST" in the middle of Figure 4.6.

With our system, the inline cache structure is part of the program execution. The corresponding nodes are in fact Truffle AST nodes that implement the sharing. Using AST nodes for building an inline cache is a common technique when using Truffle [51]. Sharing the Rectangle with this caching structure in place amounts to checking that the 3 object shapes match the *local* Rectangle and Point shapes, and updating them to their *shared* variants. This technique is 35× faster than using the generic recursive write barrier from Listing 4.5 for this example, as detailed in Section 4.6.6.

The Truffle AST can then be compiled, exposing the structure of the object graph to the compiler. Listing 4.7 represents Java code that is illustrative of the output of the partial evaluation phases of the Graal compiler applied to the AST in Figure 4.6, with AST node bodies inlined, loops unrolled, and code executed as far as it can be without run-time values. However, the real output of the compiler is machine code, not Java code as shown here.

In this way, the compiler can generate code to efficiently check for the structure of a specific object graph and without having to read primitive fields. This optimization therefore allows us to check efficiently if a small object graph matches a previously seen structure with just a few shape checks and field reads. Furthermore, it minimizes the overhead of sharing to just changing a few object shapes.

In more technical terms, the purpose of the optimization is to enable partial evaluation of the `share` method with respect to the structure of observed object graphs. That is, we want to compile an efficient version of the `share` method specialized for the object graph structure observed at a particular field assignment site. The technique also works in the case of circular object graphs, because the AST is built while sharing is performed, such that recursive references do not create a child node when an object is already shared. This optimization is similar to *dispatch chains* [38, 51], but instead of caching the result of a lookup or test, the full tree structure is built to capture the structure of an object graph.

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

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

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

Listing 4.7: Specialized sharing function for a Rectangle and its two Points. This code illustrates what the Deep Sharing nodes in Figure 4.6 perform when they are compiled down to machine code.

4.5 Discussion

The design choices of Sections 4.3 and 4.4 have tradeoffs, which are discussed in this section.

4.5.1 Sharing Large Object Graphs

As discussed in Section 4.4.3, the write barrier can be well optimized for small object graphs. However, a traversal is required to share large object graphs. The write barrier needs to ensure that all objects referenced transitively from the initial sharing root, which are not already shared, become shared. This can have a significant effect on performance by introducing overhead for sharing that is proportional to the size of the object graph.

One technique to address this is to eagerly *pre-share* objects that are likely to be shared eventually so to avoid the traversal of larger structures. This can be achieved by tracking whether objects become shared, based on their allocation site. To this end, some metadata can be associated with the object allocation site to keep track of how many objects are allocated and how many of them are shared later on. If the number of object allocations resulting in object sharing is high, objects created at the specific allocation site could be *pre-shared*, i.e., assigned a shared shape directly when allocated. This would avoid the large object graph traversal and instead only share single objects or smaller structures as they are added to the large object graph. Objects added to the graph will be pre-shared as well if most of the objects allocated at the same allocation sites are later added to a shared object graph.

This technique is similar to what is done by Domani et al. [23] and Clifford et al. [12]. Domani et al. use such a technique to allocate objects directly in the global heap as described in further detail in Section 6.2.1. Clifford et al. use *mementos* to gather allocation site feedback and to drive decisions such as pre-tenuring objects, the choice of collection representation, and initial collection storage size.

So far, we have not experimented with this approach, because the benchmark set we use does not present the need for this optimization.

4.5.2 Optimizing Memory Usage of Objects

To ensure safe concurrent reads without synchronization, our approach requires that storage locations are not reused by different pairs of field and type. This may potentially lead to unused space in objects in which field types changed or fields were removed. For objects where field types changed, only one storage location per field can be left unused, because at most one type transition per field can happen (from a primitive type to `Object`, as field types can only transition to a more general type). For objects where fields were removed, previously allocated storage locations will not be reused. We consider this a form of internal fragmentation of the object storage. Although fragmentation can result in increased memory consumption for pathological cases with many field removals, we do not consider this aspect a practical limitation of our object model, as removing fields from an object is considered an operation that happens rarely, and type transitions are limited by the number of fields.

One solution to this problem would be to trigger a cleanup phase for objects and their shapes. Such a cleanup would compact objects and their corresponding shapes by removing unused storage locations. This could be realized either based on a threshold for the degree of fragmentation observed for shapes, or it could be done periodically. The main requirement for safety would be to ensure that no threads can observe the updates of shapes and objects, which can be realized using guest-language safepoints [15]. Since such a cleanup phase would have a performance cost, it could also be integrated with the garbage collector, which could minimize the cost by combining the cleanup with a major collection.

4.5.3 Correctness When Interacting with External Code

A language implementation using our safe object model likely interacts with existing code such as Java libraries or native code via Java's native interface. To ensure correctness for objects handed to such external code, we mark these objects as shared even if no other threads are involved. This is necessary because it is not guaranteed that the external code does not use threads itself. Furthermore, we expect external code to treat objects as opaque handles and to use the correct accessor functions of the object model, which do the necessary synchronization. External code that does not use these accessor functions is considered inherently unsafe and outside the scope of this work.

4.5.4 Language-Independent Object Model

The Truffle object model is a language-independent runtime component and is currently used by Truffle-based language runtimes, including TruffleRuby. Our safe object model is fully-compatible with the Truffle object model API, and can be used as a drop-in replacement for any language runtime based on the Truffle framework. We consider this as an added value for our safe object model, as it implies that it can be used for a wide range of languages, including class-based languages such as Smalltalk, prototype-based languages such as JavaScript, or languages with more complex object models such as R [58].

The language independence of the safe object model has the benefit that a wide range of languages can use shared-memory concurrency. Even if it is arguable whether explicit shared-memory concurrency like in Java or Ruby is a desirable programming model, our safe object model can be used as the core underlying runtime mechanism to enable disciplined concurrency models that may support higher-level, safe, and concurrent access to shared objects.

4.5.5 Parallel Field Updates on Same Object are Limited

One conceptual limitation of the proposed object-granularity for synchronizing objects is that the safe object model does not allow multiple field updates for the same object at the same time (the update will be sequentialized by the object monitor). For instance, with Java's object representation it is possible to update separate fields by different threads in parallel. However, we believe that not supporting such parallel updates is not a major limitation. Even in Java, parallel updates to fields in the same object are highly problematic if the fields are on the same cache line, because the contention will degrade performance significantly. Thus, for performance it is generally advisable to avoid designs that rely on updating fields in the same object in parallel. Furthermore, it is unclear whether the performance cost of more fine-grained locking would be a good tradeoff to support this minor use case.

4.5.6 Lazy Sharing of the Initial Set of Shared Objects

As an optional optimization, tracking of shared objects only needs to be done in truly multi-threaded programs. Thus, shared shapes start to be assigned only when a second thread is created. Before the second thread is *started*, the initial set of globally-reachable objects and all objects reachable from it become shared.

This has the benefit of not requiring any synchronization for purely sequential programs. Furthermore, it can improve the startup behavior of applications, because the object graph of the initial set of shared objects is only traversed when a second thread is started. However, after implementation, we found that this optimization makes no significant performance benefit as confirmed in the evaluation. Our approach to distinguish between local and shared objects seems to be sufficient already.

4.5.7 Alternatives for Class-Based Languages

For languages with explicit classes that support dynamic adding or removing of fields, there are alternative designs for a safe object model. Assuming that the set of fields always stabilizes for all instances of a class, safety could be ensured without synchronization in the compiled code. With this assumption, it would be rare that the layout of classes changes, and instead of using a fine-grained synchronization as in our approach, synchronization could be done globally. Thus, instead of using a per-object lock for writes, a global synchronization point, such as a safepoint [15], could be used when class layouts need to be changed. This synchronization would coordinate all threads to perform the change for all objects of that class as well as updating the class layout to include the new field. However, this approach has other tradeoffs such as less fine-grained type specializations (per class instead of per instance), potential memory overhead (all instances have all fields), and scalability and warmup concerns (layout changes need global synchronization). Whether the stability assumption holds is also unclear, and would most likely need a fallback mechanism for less stable classes. This approach was implemented in SOMns [49], a Truffle-based implementation of the Newspeak language.

4.6 Evaluation

We evaluate the proposed safe object model based on TruffleRuby [81], a Ruby implementation on top of the Truffle framework and the Graal just-in-time compiler.

We evaluate our safe object model by comparing it to the unsafe version of TruffleRuby over a range of benchmarks to analyze, namely, the performance for sequential code, the worst-case write overhead, the cost of sharing, the performance for parallel actor benchmarks, and the memory usage. To relate our results to existing systems, we use Java and Scala on the HotSpot JVM, as well as JavaScript on V8, to demonstrate that our implementation of Ruby has reached a competitive performance level and to draw conclusions about the peak performance impact of our approach.

4.6.1 Methodology

All benchmarks discussed in this section are executed on a machine with an Intel Xeon E5-2690 with 8 cores, 2 threads per core, at 2.90 GHz. The Java VM is configured to use up to 2GB of heap space. All results are based on revision `2075f904` of Graal[2] and revision `0bd7fa2d` of TruffleRuby[3].

Since Truffle and Graal are designed for implementing languages for server applications, this evaluation focuses on peak performance to assess the impact of our safe object model on the performance of long-running code. Consequently, each benchmark is executed for 1000 iterations within the same VM instance. We manually verified that all benchmarks are fully warmed-up after the 500th iteration. Discarding the warmup iterations provides us with data on the peak performance. Each benchmark is run in a configuration where each iteration takes at least 50ms, ensuring that timing calls are insignificant in the iteration time and the precision of the monotonic clock is sufficient for accurate comparisons. For visualizing the results, we use traditional box plots that indicate the median and show a box from the 1st to the 3rd quartile. The whiskers extend from the box to the farthest value that is within 1.5 times the interquartile range.

4.6.2 Baseline Performance of TruffleRuby

Since this work is based on TruffleRuby, we first demonstrate that its performance is competitive with custom-built dynamic language VMs. This is important because we want to highlight any overhead that our safe object model would introduce. On slower VMs, small overheads could go unnoticed because the VM is generally slow in the first place. For that purpose, we take a set of twelve benchmarks that have been implemented for Java, JavaScript, and Ruby to enable a comparison of a set of core features of object-oriented languages [52]. This includes objects, closures, arrays, method dispatch, and basic operations. The benchmark set includes classic VM benchmarks such as DeltaBlue and Richards [95], more modern use cases such as JSON parsing, and classic kernel benchmarks such as Bounce, List, Mandelbrot, NBody, Permute, Queens, Sieve, Storage, and Towers. The benchmarks are carefully translated to all languages with the goal to be as identical as possible, lexically and in their behavior, while still using the languages idiomatically. With this approach, we measure the effectiveness of the just-in-time compilers, the object representation, and the method dispatch support, which are the most important criteria for assessing the performance of an object representation. While these benchmarks are not representative for large applications, we believe that they allow us to compare the performance of a common *core* language between Java, JavaScript, and Ruby.

Figure 4.8 depicts the results normalized to Java 1.8.0_66 as a box plot. The dots show the average peak performance for each benchmark to detail their distribution for each language implementation. The plot leaves out the results for Ruby MRI 2.3, the standard Ruby implementation, which is at the median 42.5 times slower than Java, as well as JRuby 9.0.5.0, which is 20.2 times slower. Compared to that Node.js 5.4, a JavaScript runtime built on Google's V8

[2]`https://github.com/oracle/graal/commit/2075f904`

[3]`https://github.com/oracle/truffleruby/commit/0bd7fa2d`

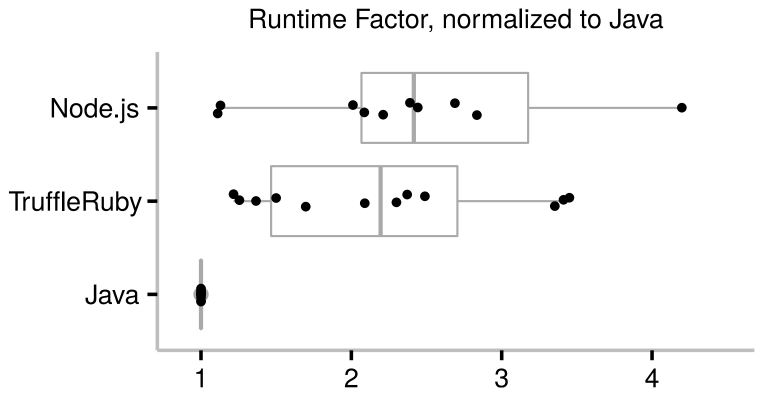


Figure 4.8: Comparing the peak performance of Java 1.8.0_66, TruffleRuby, and Node.js 5.4 based on twelve benchmarks that use a set of common language features between all three languages. Lower is better.

engine, is an order of magnitude faster. The benchmarks on Node.js are at the median a factor of 2.4 times slower than Java. The TruffleRuby implementation with the unsafe object model reaches the same level of performance and is at the median a factor of 2.2 times slower than the Java implementation. From these results, we conclude that TruffleRuby can compete with custom dynamic language VMs such as V8.

4.6.3 Impact on Sequential Performance

To verify that our design of a safe object model has no impact on the sequential performance, we use the benchmarks from the previous experiment. To focus on the object model's performance, we report only the results for the benchmarks that access object fields. We run them in three configurations, the original *unsafe* TruffleRuby, the version with our *safe* and optimized object model, and *all shared*, a configuration where all objects are considered shared and therefore all object field writes are synchronized. More precisely, this *all shared* configuration is implemented by always returning `true` for the `Shape#isShared()` predicate, which means new objects are already shared and dynamic sharing does not occur in that configuration. The *all shared* configuration approximates the worst-case impact of the object model, but does not represent common application behavior. We added this configuration primarily to estimate the overhead of state-of-the-art synchronization strategies as used by the JRuby object model.

As illustrated in Figure 4.9, there is no significant difference between the *unsafe* and the *safe* object model on these benchmarks. Specifically, the maximum difference between the medians is 5.1%, which is well within measurement error. However, the *all shared* configuration, synchronizing on all object writes similarly to the state of the art, incurs a large overhead and is 54% slower than *unsafe* using the geometric mean. NBody in the *all shared* configuration has the largest overhead and is 2.5 times slower, because there is a very large number of object writes as it is constantly updating objects in a N-body simulation.

4.6.4 Initial Set of Shared Objects

In order to gain more insights into what kind of objects are shared, we categorize by class the initial set of shared objects (cf. Section 4.4.1). This set is the same for all benchmarks in the

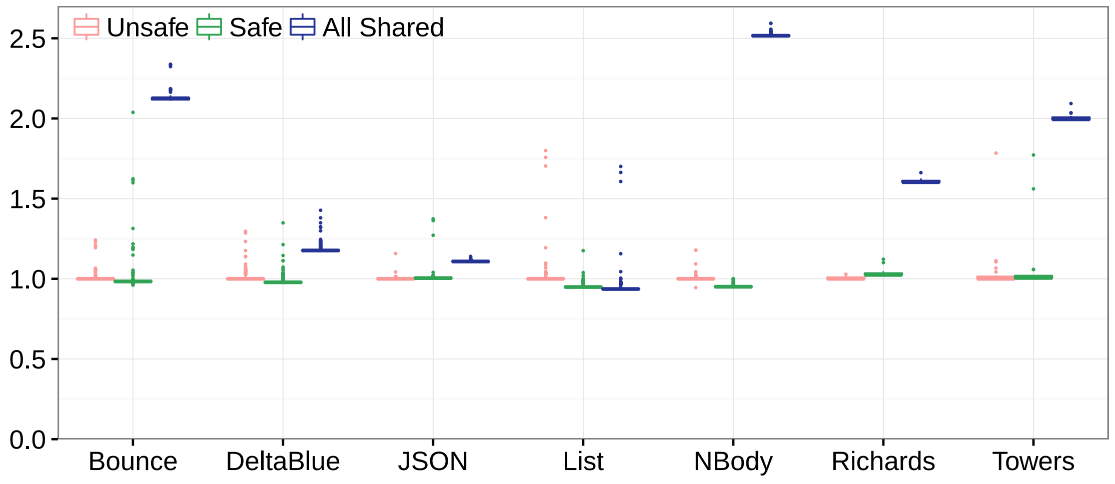


Figure 4.9: Impact on sequential performance, comparing the *unsafe* and *safe* object model. *All Shared* does not distinguish local and shared objects, and approximates state-of-the-art object models synchronizing on all object writes. Lower is better.

safe configuration because it is the set of shared objects at VM startup, before starting to load benchmark code. The sequential benchmarks add a few classes and constants when loading but only in the order of a dozen objects.

Listing 4.10 lists the number of objects per class in this initial set. In total, 2,347 objects are globally reachable from the top-level constants and the global variables. These objects are created by the TruffleRuby runtime and the core library written in Ruby. The large number of class objects is a result of Ruby's metaclasses, which more than doubles the number of classes defined in the system. We see many Strings, mostly from substitution tables and runtime configuration values exposed to the user. Ruby has a very extensive encoding support with 101 encodings defined by the system, alongside many encoding converters. Finally, we can observe some typical global state such as standard streams, the singleton *nil*, and the complex number *i*. Marking this initial set of objects as shared takes about 23 milliseconds.

4.6.5 Worst-Case Field Write Overhead

To assess the worst-case overhead of our approach, we measure the overhead of field writes to a shared object in a micro-benchmark. Listing 4.11 illustrates the benchmark, which is an extreme case, because the loop body only performs an integer addition, a comparison, and the field write. Since this benchmark is unsafe for multi-threaded applications, we also measure a more realistic variant of a counter, which first locks the object and then increments the field value. With this application-level synchronization, the benchmark gives correct results and is measured with four threads incrementing the counter.

The results for the simple field write show that the safe object model is up to 20 times slower than the unsafe version. For the more realistic counter with application-level synchronization, the safe object model is only 28% slower than the unsafe one.

In both cases, the slowdown is reflected by the complexity of the resulting machine code. The unsafe version of the simple field write benchmark only performs a shape check and an

685	Class
645	String
340	Symbol
190	Encoding::Transcoding
186	Array
109	Hash
101	Encoding
52	Module
...	...
3	IO (standard streams)
1	Thread
1	NilClass (`nil`)
1	Complex ($i = \sqrt{-1}$)

Listing 4.10: The initial set of 2,347 shared objects when starting an application under TruffleRuby with the safe object model.

```
def bench
  @count = 0
  i = 0
  while i < 100_000_000
    i += 1
    @count = i
  end
end
```

Listing 4.11: Micro-benchmark for the worst-case field write overhead. Increasing integer values are assigned to a shared object field.

unsynchronized write. The safe version performs a shape check in the loop body, then enters the monitor, checks the shape again in the monitor, writes the value to the field and exits the monitor. Because of the synchronization semantics of Java, the compiler cannot move performance-critical operations such as the shape check out of the loop.

The code generated for entering and exiting a monitor is large and contains lots of branches, which we believe is the main reason for the slowdown. One possible optimization would be to dynamically profile which path is taken, so that for instance only the biased locking path would be in the compiled code. Another alternative is to use a simpler lock implementation as we discuss in Section 8.4.

When considering the overhead for both micro-benchmarks, we must keep in mind that they do not correspond to application-level performance. The overhead for sequential code was already measured in Section 4.6.3. For multi-threaded applications, performance is more similar to the parallel benchmarks discussed in Section 4.6.7.

4.6.6 Impact of Deep Sharing

To assess the impact of the deep sharing optimization, we created a micro-benchmark similar to the example in Figure 4.6. The benchmark, illustrated in Listing 4.12, creates instances of `Rectangle` in a loop, each of them containing two instances of `Point` describing the top-left and bottom-right corners in two-dimensional cartesian coordinates. Each rectangle is then assigned

```
def bench
  i = 0
  while i < 1_000_000
    @shared = Rectangle.new(
      Point.new(i-2, i-1),
      Point.new(i+1, i))
    i += 1
  end
end
```

Listing 4.12: Benchmark for the deep sharing optimization. Rectangles are created and assigned to a shared object field.

to a shared object field.

Deep sharing is a crucial optimization here and improves performance by 35×. Without the optimization, the rectangle is shared using a generic routine (cf. Listing 4.5) that checks every field of the object, collects object references, and then performs the same analysis recursively on the referenced objects until all reachable objects are shared (in this case the Rectangle and the two Points). With our deep sharing optimization on the AST level, sharing the rectangle has no performance cost. The Graal compiler optimizes all shape checks, field reads, and shape changes of Listing 4.7 to basically nothing since it also allocates the 3 objects in the same compilation unit, mostly due to the partial escape analysis optimization [87]. The compiler moves the allocations of the 3 new objects right before the assignment to `@shared`, and is able to construct the objects directly with the shared shapes.

4.6.7 Parallel Actor Benchmarks

To assess the overhead of the safe object model on parallel code, we ported three of the parallel Savina actor benchmarks [39] from Scala to Ruby. The benchmarks focus on parallel computations that are coordinated via messages. In these benchmarks, message objects are passed by reference, not copied, and therefore rely on shared memory. We chose these benchmarks rather than other parallel benchmarks because collections were not yet thread-safe (which is introduced in Chapter 5) and so we need to limit ourselves to benchmarks not concurrently mutating shared collections. These actor benchmarks do not trigger the safety issues discussed in Section 4.2 by avoiding mutation of objects reachable by multiple threads. Therefore, they execute correctly also on the unsafe implementation.

Trapezoidal approximates the integral of a mathematical function using the trapezoidal rule. The *APSP* (All-Pairs Shortest Path) benchmark computes the shortest path between each pair of nodes in a graph using a distributed variant of the Floyd-Warshall algorithm [24]. *RadixSort* sorts integers using the radix sort algorithm.

We use a simple Ruby actor library where each actor has its own mailbox and each actor creates a thread in which it processes messages. Additionally, we scale the benchmarks so that each thread has its own CPU core to avoid contention. Figure 4.13 shows how our library compares to the widely used and highly optimized Scala Akka actor framework. On APSP, Akka is 9% faster than TruffleRuby with the unsafe object model. On RadixSort it is 9% slower and on Trapezoidal it is 10% slower than the unsafe object model.[4] From these results we

[4]The Scala version of the benchmark was changed from using `Math.pow(x,3)` to `x*x*x`, because TruffleRuby optimizes the power operator better, which would have distorted the comparison.

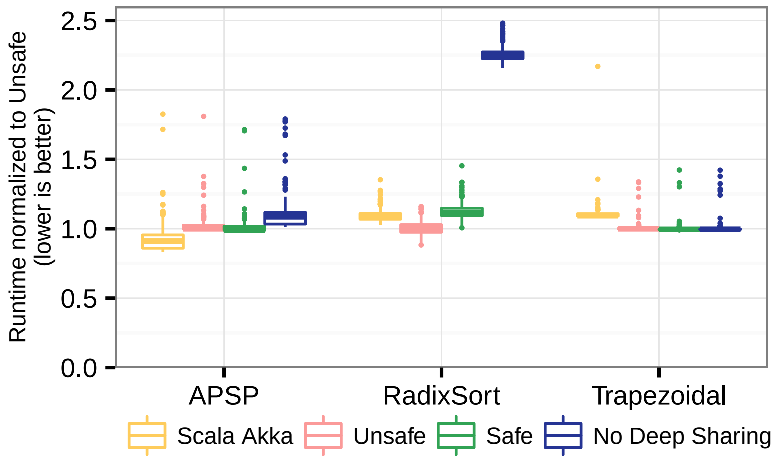


Figure 4.13: Impact on Parallel Actor Benchmarks, comparing the Scala implementation using Akka, the *unsafe* object model and the *safe* object model. *No Deep Sharing* disables the Deep Sharing optimization. Lower is better.

conclude that our Ruby actors reach a similar level of performance and are sufficient to measure the impact of the safe object model.

For the comparison of the safe and unsafe object models, our actor library design implies that almost all objects are shared. Since actors are run in different threads, the object representing an actor must be shared as it is referenced from at least two threads, the one that created the actor and the thread running the actor message loop. Consequently, all messages sent between actors must be shared as well, because they are passed from one actor to another, and therefore from one thread to another. Note that the original Scala benchmarks using Akka also share all message instances, by passing messages by reference.

Nevertheless, Figure 4.13 shows that the safe object model has an overhead of at most 12% compared to the unsafe version. The geometric mean between all three benchmarks shows that the safe object model is only 3% slower.

To characterize the behavior of the benchmarks in more detail, Table 4.1 lists the number of user-defined objects allocated per benchmark iteration. The ratio of sharing is 100% for the parallel benchmarks, confirming that all user-defined objects created per iteration end up shared. *Trapezoidal* is a so-called *embarrassingly parallel* workload, which means it requires very little communication. *APSP* sends over 700 matrices of 90x90 elements per iteration, represented as one-dimensional integer arrays, to communicate the intermediate states between the different actors. *RadixSort* sends many messages as it sorts 100,000 integers, and each of them is sent in a different message to the next actor, forming the radix selection chain. Overall, more than 400,000 messages are sent per iteration resulting in over 2.6 million messages per second. It is the only benchmark with an overhead. Compared to the unsafe version, the overhead is about 12%.

Figure 4.13 shows the importance of the structural deep sharing optimization because it greatly reduces the cost of sharing message objects. When the message instance is created in the same compiled method that sends the message, the compiler can completely remove the overhead of sharing, which is the case for this benchmark (cf. Section 4.6.6).

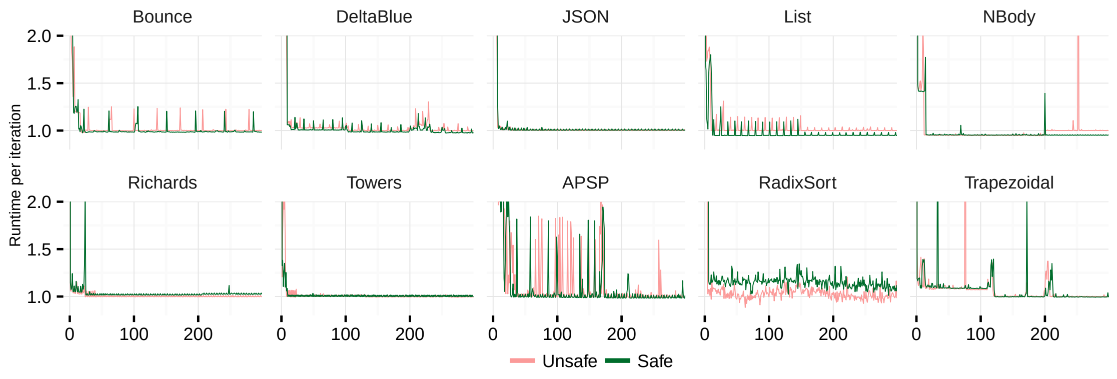


Figure 4.14: Runtime per iteration for the first 300 iterations, normalized to the median runtime for *unsafe*, illustrating the warmup of the *unsafe* and *safe* object models. Lower is better.

Thus the overhead for *RadixSort* is neither due to message sending nor sharing but due to the very frequent and small workload each actor performs when receiving a message, which consists of updating at least two fields per message received. Since actor objects are shared, these field updates require synchronization.

We consider *RadixSort* a small benchmark that performs a very high portion of write operations on shared objects compared to the other operations it performs. Thus, for the overhead of the safe object model on parallel applications, we expect the 12% overhead for *RadixSort* to be at the upper end of the range.

4.6.8 Warmup Performance

One important aspect of a modern language runtime is *warmup* performance. That is, how the performance of the language runtime evolves after initialization and during just-in-time compilation. To assess the impact of our approach on the warmup behavior of TruffleRuby, we evaluate the warmup behavior of the different object models on all the considered benchmarks.

Figure 4.14 depicts the evolution of the run time per iteration, for the first 300 iterations of each benchmark. Run times are normalized to the median run time of the unsafe object model to allow a better comparison.

As the picture clearly shows, our technique does not affect the warmup performance of TruffleRuby. Overall, the warmup of the safe object model is very similar to the warmup of the unsafe object model as the curves mostly overlap. The only two benchmarks where a warmup performance difference can be highlighted are APSP and RadixSort. APSP takes the same long warmup time to stabilize its performance for both the safe and unsafe object models. Such a long warmup is different from other benchmarks we have considered, but is not affected by our safe object model. RadixSort shows, as expected, slightly lower performance for the safe object model, since the peak performance is also affected (cf. Section 4.6.7). The performance noise in RadixSort (both for warmup and peak performance) seems mostly caused by the garbage collector, since the benchmark has a very high allocation rate. Nevertheless, the warmup behaviors of the two benchmarks do not differ. Overall, we conclude that the safe object model does not have a negative impact on warmup performance for the given benchmarks.

Table 4.1: User-defined objects allocated per iteration for each benchmark. Startup and Sidekiq count all objects including built-in types, allocated over the whole program execution. Class objects are not counted in this figure. Sharing ratios are expressed as number of *shared / total*. Frequent objects are those who represent more than 10% of the total.

Benchmark	Frequent Objects (> 10% of the total)
Startup	1,919 String, 1,742 Symbol
Bounce	150,000 Ball
DeltaBlue	108,032 Vector, 36,003 Variable
Json	67,550 JString, 43,350 Vector, 35,600 JNumber, ...
List	23,250 Element
NBody	5 Body, 1 NBodySystem
Richards	400 Packet, 300 TaskControlBlock, 300 TaskState
Towers	8,400 TowersDisk
APSP	724 Matrix, 724 ResultMessage
RadixSort	400,004 ValueMessage
Trapezoidal	5 WorkMessage, 4 WorkerActor, 4 ResultMessage
Sidekiq	575,734 String, 150,602 Proc, 123,469 Array

Benchmark	Sharing	Shapes	Objects	Extra Loc.
Startup	35.5%	45/61	2,354/6,640	0
Bounce	0%	0/2	0/151,500	0
DeltaBlue	0%	0/8	0/180,048	0
Json	0%	0/7	0/189,900	0
List	0%	0/1	0/23,250	0
NBody	0%	0/2	0/6	0
Richards	0%	0/8	0/1,350	0
Towers	0%	0/1	0/8,400	0
APSP	100%	4/4	1,456/1,456	0
RadixSort	100%	6/6	400,014/400,014	0
Trapezoidal	100%	4/4	14/14	0
Sidekiq	2.4%	125/208	25,198/1,050,537	3

4.6.9 Memory Efficiency

Memory efficiency is a crucial aspect of object representations. To assess the impact of our approach on memory consumption, we measured the memory overhead of our thread-safe objects compared with unsafe ones, using the benchmarks from the previous sections as well as a new synthetic benchmark designed to evaluate our approach in the worst case scenario. We also include numbers for a larger application, *Sidekiq*, a background job processing library using a Redis queue to estimate the memory efficiency on bigger programs.

For this evaluation, we instrumented TruffleRuby to track all object allocations, measure the number of shared objects and the used storage locations for each object. The results of our evaluation are depicted in Table 4.1. All benchmarks except *Startup* and *Sidekiq* only count user-defined objects allocated per iteration in order to depend less on language-specific representations of built-in types. In those benchmarks, objects of built-in types are only allocated by the implementation (such as a `Proc` representing a closure) or are always associated with a user-defined object (such as an `Array` backing up a `Matrix`).

For the benchmarks from the previous sections, using safe objects rather than unsafe ones introduces no additional memory cost. As the last column *Extra Loc.* shows, no extra storage locations are used by the objects. This is because all the considered benchmarks are *well-typed*. In this context, *well-typed* means a field which was assigned a primitive value is never later assigned a reference value or a primitive value of a different type. As a consequence of the well-typed property, no type transitions occur during execution, and safe objects do not require extra memory overhead.

From our experience, Ruby applications seem to be dominated by well-typed objects, considering the definition of well-typed above. As a larger example program, we use Sidekiq, a library of about 14,000 lines of Ruby code including its dependencies,[5] which creates one million objects, with more than 200 different shapes. Out of the million objects, only three are not *well-typed*. Specifically, two `Task` objects reading from the Redis connection are not well-typed: one of their fields is initialized with *false*, indicating that no data is available from Redis yet. When some data is received, the field value is replaced with a String object with the incoming data, which causes a type transition from a boolean location to a reference location. The other object which is not *well-typed* is an instance of JSON *State*, which is configured to raise an error if the object to serialize is too deep or circular. In this case, a field of the object (called `@max_nesting`) is first initialized with *false*, and then later reassigned to an integer: although both values are of primitive type, the type transition requires a migration from a primitive storage location to a reference one, in order to avoid too many shape changes as well as to ensure type correctness.

When an application is not dominated by well-typed objects, our approach could incur a memory overhead that is up to 2 times as large as the memory used by the baseline implementation for each non-well-typed object. This overhead is caused by the need to keep extra primitive locations to allow unsynchronized reads at all times (cf. Section 4.3.2).

To measure the impact of such a *worst-case* scenario, we designed a micro-benchmark that creates an object with 10 fields, which are initialized in a first phase with primitive values such as integers. In a second phase, reference values are assigned to all these fields. This effectively forces the object model to allocate space for reference locations, and in the case of the safe object model also requires to keep the old primitive locations. The memory overhead for the micro-benchmark is therefore the number of extra primitive locations. This means the safe object model must keep 10 reference locations and 10 primitives locations instead of just 10 reference locations.

In practical terms, this translates in our benchmark environment to a total size of 104 bytes for one such object with the baseline and 176 bytes for the safe object model. We consider this a reasonable trade-off as the ratio of not well-typed objects seems to be very low. As discussed in Section 4.5.2, this overhead could also be reduced dynamically in case an application has many such extra storage locations, e.g., by compacting shapes as part of garbage collection.

4.7 Summary

We presented a novel way to efficiently handle access to dynamically-typed objects while guaranteeing safety when objects are shared between multiple threads. Our safe object model prevents

[5] computed using the `sloccount` utility for the `lib` folders of Sidekiq and libraries it depends on.

lost field definitions and lost updates, as well as reading out-of-thin-air values, which are common problems for object models derived from SELF's maps [11]. Furthermore, we minimize the need for synchronization to avoid the corresponding overhead.

Generally, object models for dynamic languages provide an efficient run-time representation for objects to implement field accesses as direct memory accesses even though languages support dynamically adding and removing fields.

Our approach guarantees safety by enforcing that different field/type pairs use separate storage locations, as well as synchronizing field updates only for objects that are reachable by multiple threads. Object reachability is tracked efficiently as part of the object representation, and is optimized by using knowledge about the run-time object graph structure to minimize the operations necessary for marking objects as shared between multiple threads.

We evaluated our approach based on TruffleRuby, a Ruby implementation using the Truffle framework and the Graal just-in-time compiler, which reaches the same level of performance as V8. The evaluation of the sequential performance shows that our approach incurs zero overhead on peak performance for local objects. Parallel actor benchmarks show that the average overhead on benchmarks that write to shared objects is as low as 3%. From these results we conclude that the proposed object model enables an efficient and safe object representation for dynamic languages in multi-threaded environments. By being language-independent, the model applies to a wide range of dynamic languages. Therefore, the techniques presented in this chapter enable objects of dynamic languages to be used for shared-memory parallel computations while remaining safe and efficient.

Chapter 5

Thread-Safe Collections

In this chapter, we present our approach to automatically synchronize operations on built-in collections. We apply our efficient synchronization based on reachability to collections to maintain single-threaded performance. We then design multiple levels of synchronization for shared collections to adapt to the usage pattern of each collection instance.

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 chapter, we propose an approach for efficient and concurrent collections by gradually increasing synchronization levels according to the dynamic needs of each collection instance. With our approach, 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.

5.1 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 were rather inefficient, 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 [5, 7, 11, 12, 16, 20, 98, 99]. A major opportunity to increase performance is parallelism, allowing programmers to implement parallel algorithms and supporting their execution in the runtime.

In Chapter 4, we made significant progress by ensuring that objects can be safely and efficiently represented in multi-threaded runtimes. However, one major problem remains unsolved: how can we represent collections safely and efficiently that use optimizations such as storage strategies [7]? To give just one example, in today's dynamic language implementations, when

```
# append integers to a global array
array = Array.new()
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
# or
1   # incorrect result, appends were lost
```

Listing 5.1: A Ruby example illustrating the multi-threaded semantics for collections in dynamic language implementations. State of the art implementations either support concurrent modification of collections but then sequentialize every operation on the collection, or support parallel access to the collection but not concurrent modifications.

multiple threads append to an array, the result is either that all appends are successful, that one append gets an error because of a race condition in the underlying implementation, or that appends are lost (cf. Listing 5.1). In this work, we present a new safe, efficient, and scalable approach that can be applied to synchronize a wide range of collections.

As indicated with Listing 5.1, 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 [41], which degrades performance and scalability. PyPy-STM [55] emulates the GIL semantics and enables scaling but incurs a significant overhead on single-threaded performance. JRuby [65] and Rubinius [72] 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 thesis, *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 [13]. 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 our approach for efficient synchronization based on reachability, 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 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 synchro-

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

nization 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 layout changes are not executing. When a layout change occurs, accesses to the collection have to wait until the layout change finishes.

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 [81], 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 the 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 chapter is a proposal for a set of benchmarks that can be used to measure the performance of parallel dynamic language implementations. We ported known benchmarks and we wrote concurrent programs that make use of collections in parallel. We also created micro-benchmarks to assess specific behaviors of our 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 presented in this chapter are:

- A method of gradual synchronization, migrating collections between multiple levels of synchronization. The method adapts to 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. The main principles of this design are applicable also 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 [3], and various Ruby workloads,
- an evaluation of the scalability and performance of our solution using the benchmark suite.

5.2 Background

This section reviews some prior work on which we rely in the context of collections. Section 2.1.1 provides some context for this work by discussing how concurrency is used in Ruby. Section 2.3 explains Storage Strategies, on which we base our approach for collections. Section 2.4.1 presents a state-of-the-art lock, the Layout lock, which we later extend in this chapter.

5.2.1 Reachability and Thread-Safe Objects

Chapter 4 introduced a thread-safe object storage model for dynamic languages, enabling efficient synchronization of accesses to objects that support specialized storage for primitive values [99] and dynamic addition and removal of fields.

For efficiency, we keep track of 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 assigned to a field of 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, we use write barriers that specialize themselves and minimize the overhead.

Using shapes to distinguish between shared and local objects enables us 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, using separate shapes for local and shared objects could double the maximum number of shapes in a system, and could thereby increase the polymorphism of method calls. However, in practice, we did not observe such negative effects.

Local objects do not need any synchronization, which means that 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. Our approach in Chapter 4 uses a monitor per shared object, so that 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.

5.3 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 chapter and then discuss the requirements to be fulfilled.

Desired Semantic Intuition Table 5.1 gives seven 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.

Table 5.1: Semantics of GIL-based and unsafe implementations showing possible outcomes, and our goals.

	Example		GIL	Goal	Unsafe
1	Initial: `array = [0, 0]`		`[1, 2]`	`[1, 2]`	`[1, 2]`
	`array[0] = 1`	`array[1] = 2`			
	Result: `print array`				
2	Initial: `array = [0, 0]`		`["s", 2]`	`["s", 2]`	`["s", 2]`
	`array[0] = "s"`	`array[1] = 2`			`["s", 0]`
	Result: `print array`				
3	Initial: `array = []`		`[1, 2]`	`[1, 2]`	`[1, 2]`
	`array << 1`	`array << 2`	`[2, 1]`	`[2, 1]`	`[2, 1]`
	Result: `print array`				`[1]/[2]`
					exception
4	Initial: `hash = {}`		`{a:1, b:2}`	`{a:1, b:2}`	`{a:1, b:2}`
	`hash[:a] = 1`	`hash[:b] = 2`	`{b:2, a:1}`	`{b:2, a:1}`	`{b:2, a:1}`
	Result: `print hash`				`{a:1}/{b:2}`
					`{a:2}/{b:1}`
					exception
5	Initial: `a = [0, 0]; result = -1`		`1`	`1`	`1`
	`a[0] = 1`	`wait() until a[1] == 2`		`0`	`0`
	`a[1] = 2`	`result = a[0]`			
	Result: `print result`				
6	`key = Object.new; h = {key => 0}`		`2`	`2`	`2`
	`h[key] += 1`	`h[key] += 1`	`1`	`1`	`1`
	Result: `print h[key]`				
7	Initial: `array = [1]`		`1`	`1`	`1`
	`array = [2]`	`print array[0]`	`2`	`2`	`2`
					`0`

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 2.3.2). 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 append concurrently to an `Array`, similar to Listing 5.1. 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 even 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 lose updates or even 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 5.4.5 uses such a representation). The two threads can race and can 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 indices. 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 compiler or the CPU may reorder the assignments `a[0] = 1` and `a[1] = 2` without explicit synchronization. We consider this an application-level race and explicit synchronization or high-level mechanisms for communication need to be used to ensure correctness. For instance, a *promise* or a *queue* could be used to safely communicate a value between two threads. 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.[2]

Example 7 shares a newly-created collection and reads its contents from another thread. Unsafe has the additional outcome `0`, because it does not guarantee *safe publication* [83] and as a result might read the (allocated but not yet initialized) array element before it is set 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[]`), but in C with `malloc()`, for example, any value could be observed, since the allocated memory is not initialized. 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.

Requirements As discussed in Section 5.1, 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 5.1, 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 [32, p. 24], such that each operation transforms the data structure from one consistent state to another.

[2] 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.

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, such as reading from and writing to an `Array` index, should be atomic. With atomic we mean that operations are indivisible and have no observable intermediate state.

Iteration operations such as `Array#each` and `Hash#each_pair` should be weakly consistent [70], 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 (for instance, `array = [1, 2]; array.map { |e| array[1] = 3 if e == 1; e }` returns `[1, 3]`) 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`.

Section 5.5 addresses these requirements. While our analysis focuses on Ruby, the same issues are present in Python and similar dynamic languages.

5.4 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 [14].

5.4.1 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 appropriate synchronization strategies based on usage scenarios for shared collections;
- local and shared collections are distinguished by their 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 as well as Section 5.5 explain how concurrent strategies are realized based on these key ideas.

5.4.2 Tracking Reachability for Collections

Parallel algorithms aiming for good performance typically use local data structures for some part of their work, and only work with shared data structures when necessary [35]. Based on this observation, we adopt the idea of tracking reachability of objects from Chapter 4 (cf. Section 5.2.1) and apply it to keep track of whether collections are shared between multiple threads or whether they are 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 that we track the 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 Listing 5.2). 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 here, since the array contains only primitives like integers, which cannot reference objects.

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

Listing 5.2: Sharing a collection and its elements.

5.4.3 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 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.

5.4.4 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 like 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., to Python with its `list` and to JavaScript with its `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 5.3. 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.

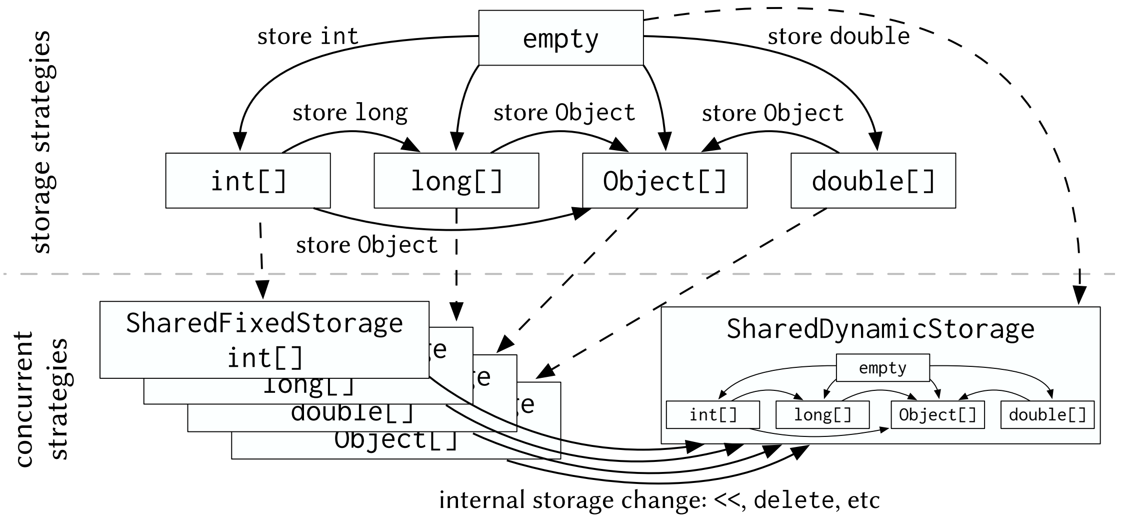


Figure 5.3: Strategies for arrays in TruffleRuby. Concurrent strategies are combined with a storage strategy. Plain arrows show storage transitions and dashed arrows represent transitions when sharing an array.

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 storage strategies to enable gradual synchronization. Concurrent strategies contain a nested storage strategy to optimize the data representation (see Figure 5.3). As detailed in Section 5.4.3, 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 5.5.2.

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 5.5.3. Section 5.8.2 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 instead to the `SharedDynamicStorage` strategy. In case elements are added to an empty array, its storage strategy needs to change and the `SharedDynamicStorage` strategy is required. If no elements are added, the additional synchronization on empty arrays has minimal impact since most operations cause out-of-bounds behavior (there are no elements to access) or are no-ops. The transitions to concurrent strategies are depicted in Figure 5.3 with dashed arrows. The figure shows that `SharedDynamicStorage` adapts its storage strategy dynamically, while the storage strategy is fixed for `SharedFixed-Storage`.

5.4.5 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,[3] 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

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

to apply for instance escape analysis [87]. The `Buckets` strategy is a traditional hash 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 5.5.4). 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.

5.4.6 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.

5.5 Implementation

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 5.3.

5.5.1 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 [74, Section 11] specifies that each read and write operation on arrays is itself atomic, except for `long[]` and `double[]` arrays. In practice, atomicity 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 5.3. 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 5.1. Data races when updating array elements (e.g., two threads concurrently incrementing an element, see example 6) cannot be addressed by the array implementation, thus 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`.

5.5.2 Migration to SharedDynamicStorage

The `SharedFixedStorage` strategy speculates that the size and the storage strategy of an `Array` do not change (cf. Section 5.4.4). If these assumptions are violated, the `Array` migrates to the `SharedDynamicStorage` strategy. We expect these violations to be rare, as detailed in Section 5.7.1.

When an `Array` uses `SharedFixedStorage`, it is already shared and its 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* [15] (cf. Chapter 3), 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 [15]. 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 do not check for guest-language safepoints *during* their execution, to guarantee that the `Array` strategy does not 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 change the strategy of the `Array`, 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 2.3.2), the strategy must be checked again after each call to Ruby code, to check if it has changed. The Ruby callback could for instance have stored an object into an `Array` that had an `int[]` storage strategy, leading to a `Object[]` storage strategy. Since these strategy checks are already performed in single-threaded code, *concurrent strategies* do not add extra strategy checks.

5.5.3 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 5.3. 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 `SharedFixed-Storage` and check the strategy again after each callback (cf. Section 5.5.2). 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 5.5.1.

5.5.4 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 [88], 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 [33].

We now detail the atomicity of the four `Hash` operations: lookups, updates, adding an entry and removing an entry. Hash lookups and updates for an existing key respectively read and write the volatile `value` field of the entry, so lookups and updates 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.

Idiomatic Concurrent `Hash` Operations Concurrent collections often have extra operations to provide atomicity guarantees for concurrent use cases. A prominent example for dictionaries is the *put-if-absent* operation. We use it as an example to discuss how to achieve natural semantics for dynamic languages, without introducing a new method in Ruby's `Hash` class. The main idea is to extend an existing single-threaded pattern and make it safe for concurrent use. Ruby's `Hash` class has a constructor which accepts a block of code to specify the default value:

```
hash = Hash.new { |h,key| h[key] = [] }
```

This block is invoked whenever a lookup `hash[key]` does not find `key` in the dictionary. Then, the block inserts a new `Array` at `key`. Thus, we can simply use `hash[key] << v` to accumulate all values `v` under the same `key` and do not need to check if there is already a value at `key`, as it is automatically created.

This feature becomes particularly interesting when `hash` is accessed concurrently. One option would be to not change `Hash` at all. Then the block might execute multiple times concurrently and override the previous value at `key`, losing the elements added in the `Array` before the override. Another option would be to synchronize the entire block as the `computeIfAbsent()` method does for Java's `ConcurrentHashMap`. Then the block is executed at most once but might deadlock if the default value block accesses another `Hash` and vice versa. We chose a different solution by allowing multiple executions of the default block, but transform the `Hash` *put* operation (`h[key] = []`) with a *put-if-absent* operation, therefore never overriding when there is already a value for `key`. The replacement of *put* with *put-if-absent* is achieved by passing a transparent proxy to the default block instead of the real `Hash` instance, intercepting a *put(key, value)* operation and replacing it with *put-if-absent(key, value)*. This provides intuitive semantics extending single-thread guarantees. Our solution enables using idiomatic code and is not subject to deadlocks.

5.5.5 Thread Safety Guarantees

The goal of concurrent strategies is to enable language implementations to provide thread safety guarantees for built-in collections (cf. Section 5.3). 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 their implementation approach.

We avoid such errors by using safepoints to atomically switch between `SharedFixedStorage` and `SharedDynamicStorage` (cf. Section 5.5.2), and by synchronizing operations on `Array` and `Hash` with the Lightweight Layout Lock as part of the `SharedDynamicStorage` and `ConcurrentBuckets` strategies. We ensure that operations only see a consistent state by using *layout changes* for any operation that could cause inconsistencies in collections. For instance, the reported size of an `Array` (`array.size` in Ruby) being bigger than the size of its storage is an inconsistency prevented by *layout changes*.

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.

Furthermore, our implementation provides safe publication [83] semantics for shared collections. In other words, it is guaranteed that other threads will observe values at least as recent as set just before sharing the collection, as illustrated in Example 7 of Table 5.1. This is guaranteed by the process of sharing a collection, which — similar to thread-safe objects — involves having a memory barrier between marking the collection as shared and the actual assignment making the collection reachable from other threads, as detailed in Section 4.4.2.

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 5.8.2).

5.6 Lightweight Layout Lock

This section describes the novel *Lightweight Layout Lock*. The Lightweight Layout Lock improves over the Layout Lock (cf. Section 2.4.1) and we designed it to support our primary contribution of gradual synchronization with concurrent strategies (Section 5.4). 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 5.8.

5.6.1 Structure and Design

First, we describe the overall structure of the lock, illustrated in Figure 5.4, and its design from a high-level view.

The Lightweight Layout Lock uses a *base lock* that supports *shared* versus *exclusive* locking. That is, the base lock supports locking in two modes: *shared*, where multiple threads can acquire the lock in *shared* mode at the same time, and *exclusive* which grants exclusive access to the thread which acquires the lock. This is also known as a readers-writer lock, but we do not use that term to avoid confusion with the Lightweight Layout Lock access modes.

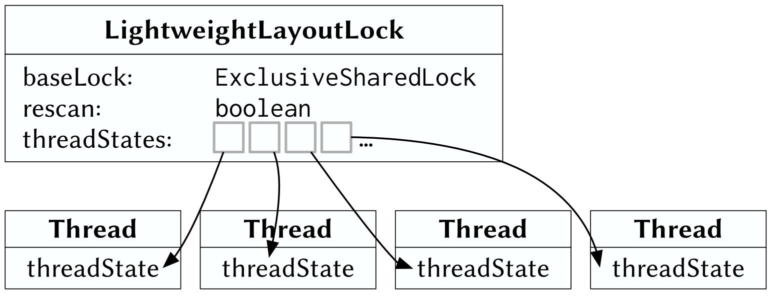


Figure 5.4: The Lightweight Layout Lock is composed of a base lock that supports shared versus exclusive accesses, an optimization flag that indicates whether the lock needs to change thread states, and a list of references to the thread states of registered threads.

The Lightweight Layout Lock 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 3 types of accesses: *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. There are 3 different thread states:

`INACTIVE`, also just 0: the thread is performing a read access, or does not use the lock.

`WRITING`, abbreviated `WR`: the thread is performing a write access.

`LAYOUT CHANGE`, abbreviated `LC`: the thread is performing a layout change.

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 was any read or write access between layout change operations, and is used to facilitate the layout change fast path described below.

Table 5.2: Example of actions and thread state changes in the Lightweight Layout Lock.

t_0	T_1 [INACTIVE]	T_2 [INACTIVE]	T_3 [INACTIVE]

t_1	`startRead()`
	`finishRead()`
	`return true`
		`startWrite()`
		[WRITING]
			`startWrite()`
			[WRITING]

t_2		`finishWrite()`
		[INACTIVE]
		`startRead()`	

t_3	`startLayoutChange()` [LAYOUT CHANGE] :	[LAYOUT CHANGE]	[WRITING+PENDING]
t_4	: : :		`finishWrite()` [INACTIVE+PENDING] = [LAYOUT CHANGE]
t_5	`rescan = false`		
t_6		`finishRead()`	
t_7	`finishLayoutChange()`	:	
t_8		[INACTIVE], `rescan = true` `return false`	

5.6.2 Example

We first describe an example to illustrate interactions with the Lightweight Layout Lock. Table 5.2 shows a scenario with three threads T_1 to T_3 using the lock. The table uses the notation [*state*] to show a thread's state. We assume that by time t_0, all three threads have already been registered with the lock, with their states set to `INACTIVE`, and the `rescan` flag was set to `true`.

At time t_1, thread T_1 performs a read operation followed by a successful `finishRead` action, since there was no concurrent layout change. Threads T_2 and T_3 start a write operation, and therefore change their respective states to `WRITING`. Then, at time t_2, thread T_2 finishes its write operation, and therefore restores its state to `INACTIVE`. Thread T_2 then starts a read operation. Thread T_3's write operation, however, has not finished, and its state remains `WRITING`.

At time t_3, thread T_1 starts a layout change, with the actions of `startLayoutChange` highlighted in blue. To do so, it first acquires the lock's `baseLock` in *exclusive* mode. Since the `rescan` flag is set to true (at t_0), it uses the lock's `threadStates` list to reach the registered thread states (cf. Figure 5.4) and attempts to change them all to `LAYOUT CHANGE`. It can change its own state, and T_2's to `LAYOUT CHANGE` immediately. However, thread T_3 has not finished writing yet. Therefore, `startLayoutChange` adds the value `PENDING` to T_3's state, to prevent T_3 from starting a subsequent write operation, and waits until T_3's state becomes `LAYOUT CHANGE`. At time t_4, thread T_3 finally issues a `finishWrite` operation, subtracting the `WRITING` value from its state, and therefore changing it to `INACTIVE+PENDING`. That value is actually equal to `LAYOUT CHANGE`. Thus, at time t_5, all `threadStates` are set to `LAYOUT CHANGE`. T_1's `startLayoutChange` operation then sets the `rescan` flag to false, and the actual layout change operation can proceed.

At time t_6, thread T_2 is finishing its read operation and calls `finishRead`. However, since its state is not `INACTIVE`, it uses its *slow path* and must wait for the layout change to complete. At time t_7, thread T_1 finishes the layout change operation. At time t_8, thread T_2's `finishRead` can then proceed by changing its state to `INACTIVE`, setting the `rescan` flag, and returning `false`, to indicate the read operation failed due to a concurrent layout change. After that, thread T_1

can then restart its read operation.

5.6.3 Implementation

We now describe the implementation of the lock along with the code shown in Listings 5.5 and 5.6. An example usage of the API for an array to which elements can be appended is shown in Listing 5.7 and might help to understand how the different methods are used.

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 5.5), 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`. A memory barrier is needed in `finishRead()` to prevent data reads to float below the read of the `threadState`, similar to Java StampedLock's `validate()`.

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. Therefore, `finishWrite()` changes the `threadState` from `WR` to `0` or from `WR+LC` to `LC`.

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 another layout change operation, without any intervening read or write operation since then. Thus, all the `threadState`s are still set to `LC`, and the layout change operation may proceed immediately.

Slow Path In their slow paths, read and write operations reset their thread's `threadState` to recover from layout changes. They do so by calling the `slowSet()` method in Listing 5.5, 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 `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[4], if needed, 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

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

```
class LightweightLayoutLock {
  final static int WR = 1, LC = 2;
  ExclusiveSharedLock baseLock;
  boolean rescan = false;
  List<AtomicInteger> threadStates;

  /* Read operation */
  void startRead(AtomicInteger threadState) { }

  boolean finishRead(AtomicInteger threadState) {
    Unsafe.loadFence(); // Prevent data reads to reorder with threadState.get()
    if (threadState.get() == 0) { // INACTIVE
      return true; // no LC so the read data is valid
    } else {
      slowSet(threadState, 0); // reset to INACTIVE after LC done
      return false; // the read operation must be retried
    }
  }

  /* Write operation */
  void startWrite(AtomicInteger threadState) {
    if (!threadState.compareAndSet(0, WR)) {
      // LC happened or pending, wait for LC to complete
      slowSet(threadState, WR);
    }
  }

  void finishWrite(AtomicInteger threadState) {
    // subtract since LC might be pending
    threadState.addAndGet(-WR);
  }

  /* Common slow path for read and write operations */
  void slowSet(AtomicInteger threadState, int n) {
    baseLock.lockShared(); // wait for LC to complete
    rescan = true; // subsequent LC must scan thread states
    threadState.set(n);
    baseLock.unlockShared();
  }
```

Listing 5.5: A Lightweight Layout Lock implementation: read and write operations.

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 write operations, thus avoiding starvation for layout changes.

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 5.6. During thread registration, the `baseLock` is held in exclusive mode, to prevent a race between layout changes, 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

```
  /* Layout change operation */
  void startLayoutChange() {
    baseLock.lockExclusive(); // wait for any slowSet() or other LC to complete
    if (rescan) { // slow path, when slowSet() was executed since last LC
      for (AtomicInteger threadState : threadStates) {
        if (threadState.get() != LC) {
          threadState.addAndGet(LC); // atomically add LC: 0->LC or WR->WR+LC

          // wait until that thread's write operation completes, if any
          while (threadState.get() != LC) { }
        }
      }
      // No need to rescan for the next startLayoutChange, unless slowSet() executed
      rescan = false;
    }
  }

  void finishLayoutChange() {
    // Do not reset threadStates to optimize consecutive layout changes
    baseLock.unlockExclusive();
  }

  /* Thread registration and unregistration */
  void registerThread(AtomicInteger threadState) {
    threadState.set(LC);  // Initial value
    baseLock.lockExclusive();
    threadStates.add(threadState);
    baseLock.unlockExclusive();
  }

  void unregisterThread(AtomicInteger threadState) {
    baseLock.lockExclusive();
    threadStates.remove(threadState);
    baseLock.unlockExclusive();
  }
}
```

Listing 5.6: A Lightweight Layout Lock implementation: layout changes and thread registration.

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

5.6.4 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.

```
class AppendableArray<E> {
  final LightweightLayoutLock lock = new LightweightLayoutLock();
  // store and size are only reassigned under a layout change operation
  E[] store = (E[]) new Object[8];
  int size = 0;

  E get(int index) {
    AtomicInteger ts = getThreadState();
    E result;
    do {
      // store might be reassigned concurrently, make sure we check the length and
      // read an element from the same instance to avoid IndexOutOfBoundsException
      E[] snapshot = store;
      if (index < size && index < snapshot.length) {
        result = snapshot[index];
      } else {
        result = null;
      }
    } while (!lock.finishRead(ts));
    // When finishRead() returns true, then store and size did not
    // change concurrently since the previous finishRead()
    return result;
  }

  void set(int index, E element) {
    AtomicInteger ts = getThreadState();
    lock.startWrite(ts); // Shared access but no concurrent layout change
    // Therefore store and size cannot change within startWrite()/finishWrite()
    try {
      if (index < size) {
        store[index] = element;
      } else {
        throw new IndexOutOfBoundsException();
      }
    } finally {
      lock.finishWrite(ts);
    }
  }

  void append(E element) {
    lock.startLayoutChange(); // Exclusive access to the data structure
    try {
      if (store.length <= size) {
        store = Arrays.copyOf(store, store.length * 2);
      }
      store[size] = element;
      size++;
    } finally {
      lock.finishLayoutChange();
    }
  }

  void remove(int index) {
    lock.startLayoutChange();
    ...
  }
}
```

Listing 5.7: Example usage of the Lightweight Layout Lock API for an array to which elements can be appended and removed. The `get()` method is the most complicated one because it might read inconsistent `store` and `size` and in that case it should retry.

5.6.5 Reasoning about Correctness

In this section, we sketch the correctness of the Lightweight Layout Lock protocol by describing invariants that are preserved by the lock. Similarly to the Layout Lock [13], the Lightweight Layout Lock must satisfy the following properties:

- layout change operations are mutually exclusive,
- write operations are not concurrent with a layout change operation, and
- optimistic read operations detect a concurrent layout change operation and fail, allowing for recovery.

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.

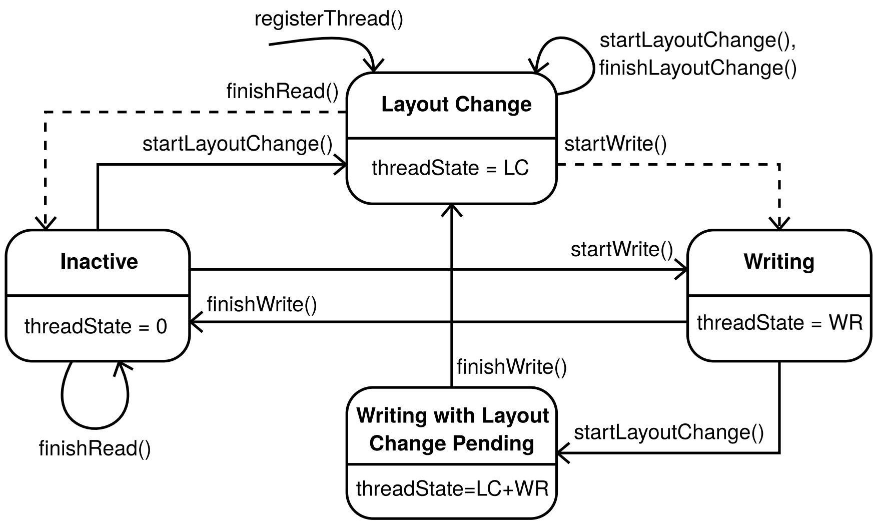


Figure 5.8: This diagram shows all the state transitions of a `threadState`, as they are triggered by the different API calls of the Lightweight Layout Lock. The dashed arrows indicate the reader and writer slow paths (i.e. a call to `slowSet()` in Listing 5.5). There are no arrows for `startRead()` since it does nothing and therefore neither accesses nor modifies the `threadState`.

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 [36].

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*, symmetric 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 Figure 5.8. 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 5.5, 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 5.5) 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 64), thus thread registration preserves this invariant.

When a reader calls `finishRead()` and detects (at line 12) that the `threadState` is not 0 (not *Inactive*), it calls `slowSet()` (at line 15) 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 24) 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 43) 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. Figure 5.8). 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 thread calls `finishWrite()` and clears the `WR` bit.

After all `threadState`s 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 `threadState`s and tries to set them to `LC` (*Layout Change*). However, since the CAS of `startWrite` succeeded for `threadState`, then the comparison at line 45 fails for that same `threadState` (either before or after the CAS), thus the value `LC` is atomically added to the `threadState` at line 46 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's 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 42), and releases the lock only after the layout change operation completed (at line 59), 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 via `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 16).

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.

5.7 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.

5.7.1 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 5.5.2) to perform the migration atomically. Safepoints stop all threads during their operation. Thus, if safepoints 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 5.4.4, 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 preallocated and then have elements added or removed, or have elements of an incompatible type stored in them. This violates their fixed storage assumption. For such cases, it is possible to extend our heuristic using allocation site feedback such as *mementos* [12], 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.

5.7.2 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 5.8.

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. [13] was implemented both in Java and C++. We therefore believe that the Lightweight Layout Lock could also be ported to other memory models than 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 a safepoint mechanism. We rely specifically on the synchronization semantics of safepoints for migrating between concurrent strategies. In systems without a safepoint mechanism, adding a safepoint mechanism would possibly add run time overhead [46]. We also assume that accessing array elements is atomic (cf. Section 5.5.1).

5.7.3 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 (i.e., the code might not do what is expected). For instance, when implementing a classic parallel algorithm (cf. Section 5.8.6), one would not expect the need to switch between different concurrent strategies for an array. More generally, races on the layouts of data structures can indicate performance bugs. For these kind of issues, we imagine tools similar to the work of Gong et al. [27] and St-Amour and Guo [86], which could use the information on safepoints and strategy changes to guide performance optimizations.

5.8 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 also 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 [3], a benchmark from PyPy-STM [55], 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 [13], 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 that every benchmark iteration takes at least a second. For micro-benchmarks, each iteration measures the throughput during 5 seconds. We use weak scaling 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.

5.8.1 Baseline Single-Threaded Performance

We use the *Are We Fast Yet* benchmark suite [52], 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 5.9 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 5.9, 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 implementations such as TruffleRuby, PyPy, and V8. Marr et al. [52] showed that TruffleRuby and V8 reach about the same performance on their benchmarks, illustrating that the TruffleRuby single-threaded performance is among the best.

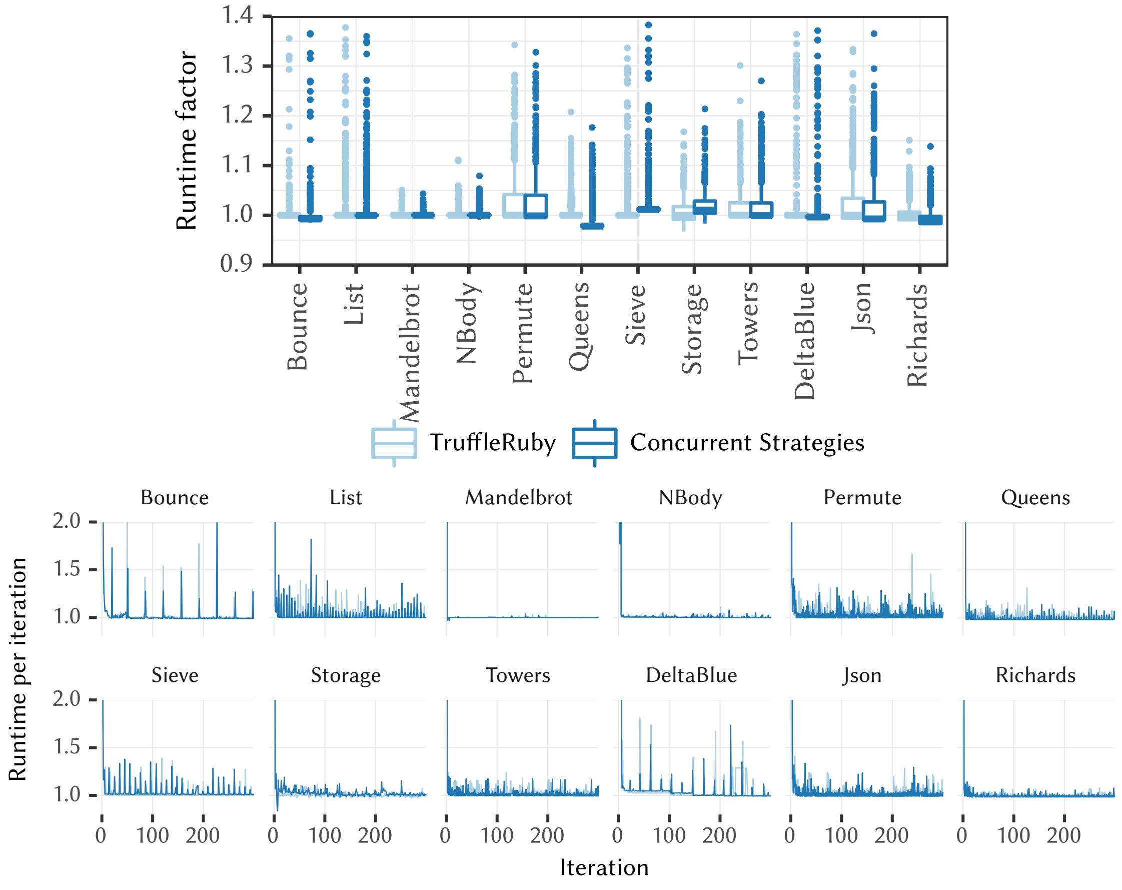
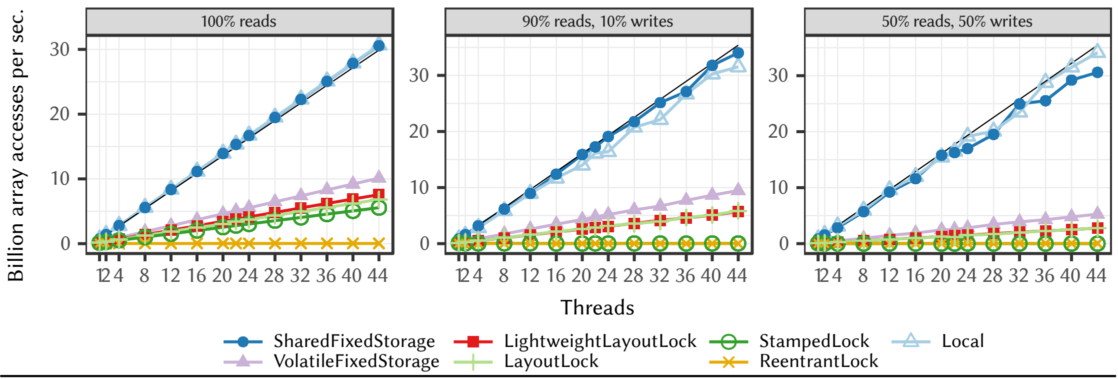


Figure 5.9: Performance of `Concurrent Strategies` on 12 single-threaded benchmarks. Results are normalized per benchmark to the median of unmodified `TruffleRuby`. The box plot indicates the median and quartiles of the run time factor. On average, performance is identical within measurement errors and 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`.

5.8.2 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 simple read and write accesses that do not cause layout changes to compare `SharedFixedStorage` with `SharedDynamicStorage`, for which we evaluate different types of locks. These benchmarks sum the array elements to avoid optimizing out the reads. As Figure 5.10 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 5.5.5) has



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 5.10: 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. The table shows 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.

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 5.10 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.

The plot in Figure 5.10 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 a table as well. 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 `SharedDynamicStorage` configurations.

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 5.11 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.

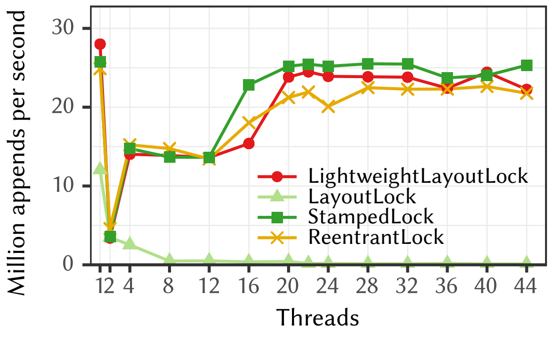


Figure 5.11: Throughput of concurrent array appends. The `LayoutLock` performs poorly due to frequent layout changes.

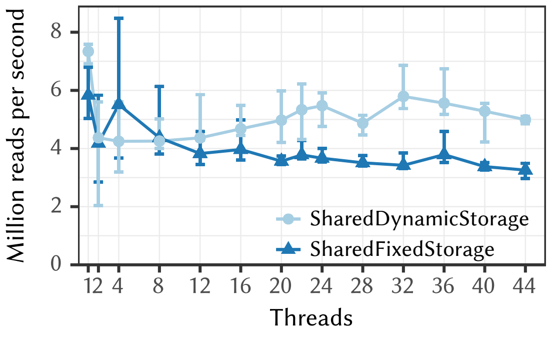


Figure 5.12: Throughput of migrating 1000 `SharedFixedStorage` arrays and then reading from them, compared to starting with `SharedDynamicStorage`.

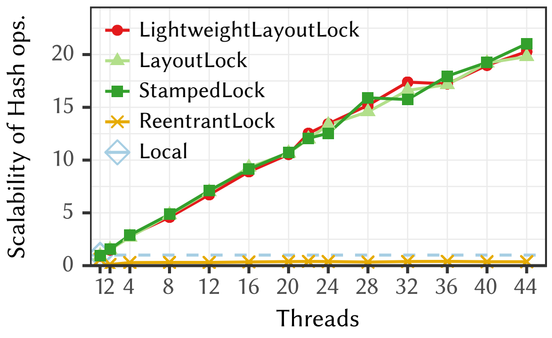


Figure 5.13: Scalability of the Ruby `Hash` with 80% contains, 10% put, and 10% delete operations over 65,536 keys, relative to 1-thread performance of `Local`.

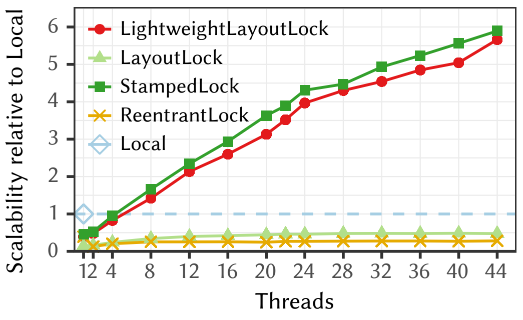


Figure 5.14: Scalability of `group_by` benchmark inserting 40 millions random values into 65,536 bins. The dashed line shows the `Local` (single-threaded) baseline.

5.8.3 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 5.7.1), we use a throughput benchmark that causes the migration of 1000 new 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 `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 5.12 shows that `SharedFixedStorage` with layout changes 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.

5.8.4 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 `Hash`. Figure 5.13 shows that the different locks scale similarly except for Reentrant-Lock, which does not support parallel operations. `StampedLock` scales well here because each `Hash` operation starts with a lookup, which uses `StampedLock`'s optimistic read, similar to layout locks. However, the locks 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 (which cannot easily be seen on the figure). For comparison with array accesses in absolute terms, the Lightweight Layout Lock reaches 603 million `Hash` operations per second on 44 threads.

5.8.5 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 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 numerical values into 65,536 bins, categorizing by their value. `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 5.14 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.

5.8.6 Classic Parallel Benchmarks

We evaluate the scalability of our approach on 12 classic benchmarks and compare it 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 our benchmark results with those of less dynamic languages such as Java and Fortran because these languages are known to be highly scalable for parallel applications. The chosen benchmarks therefore 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. [7] is that performance-sensitive

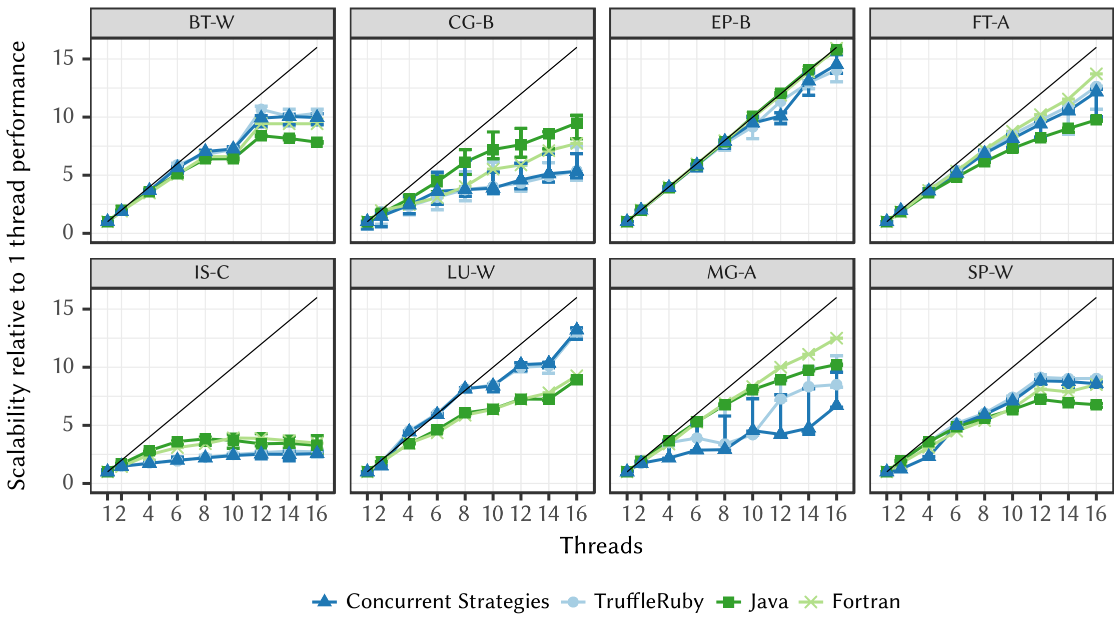


Figure 5.15: Scalability relative to 1-thread performance per language on the NAS Parallel Benchmarks. The straight black lines denote ideal scalability based on 1-thread performance. All implementations show similar scalability.

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.

NAS Parallel Benchmarks We use the NASA Advanced Supercomputing Parallel Benchmarks 3.0 (NPB) [3] 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 [64]. 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 the EP benchmark 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 rather than absolute throughput.

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

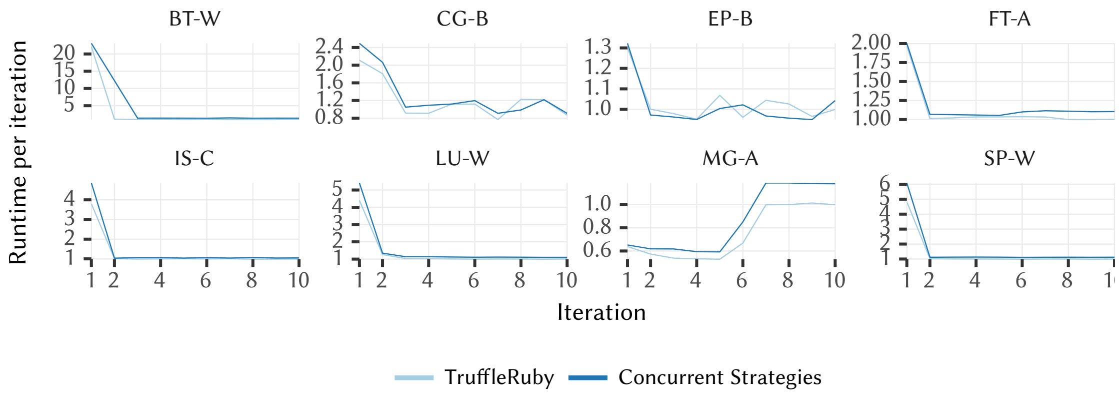


Figure 5.16: Runtime per iteration with 8 threads relative to the median run time for `TruffleRuby`, illustrating the warmup of `TruffleRuby` and `Concurrent Strategies` on the NAS Parallel Benchmarks. Warmup graphs for different number of threads are similar. Lower is better.

Figure 5.16 shows the warmup behavior with 8 threads. On the MG benchmark, 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, the run time of the first iteration 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 in Figure 5.15. Overall, we conclude that our approach provides a safe and scalable programming model for classic parallel algorithms.

PyPy-STM Mandelbrot We ported the parallel Mandelbrot benchmark from PyPy-STM [55] 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 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 5.17, 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 [40], 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 5.18, both `SharedFixedStorage`

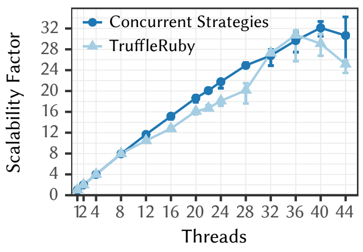


Figure 5.17: Scalability of the parallel Mandelbrot benchmark, relative to 1-thread performance of TruffleRuby.

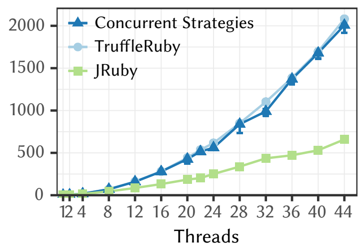


Figure 5.18: Scalability of the threaded-reverse benchmark, relative to 1-thread performance *of each implementation.*

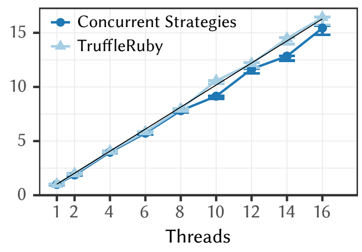


Figure 5.19: Scalability of the Sidekiq Sum benchmark with Concurrent Strategies, relative to 1-thread performance of TruffleRuby.

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 than JRuby while also providing thread safety.

5.8.7 Realistic Ruby Benchmarks

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 an HTTP server.

Sidekiq We run a simplified version of the popular Ruby background processing library, Sidekiq [71], 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 scala-

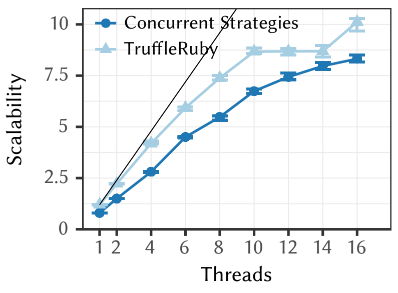


Figure 5.20: Scalability of the Sidekiq PDF Invoice benchmark with Concurrent Strategies, relative to 1-thread performance of TruffleRuby.

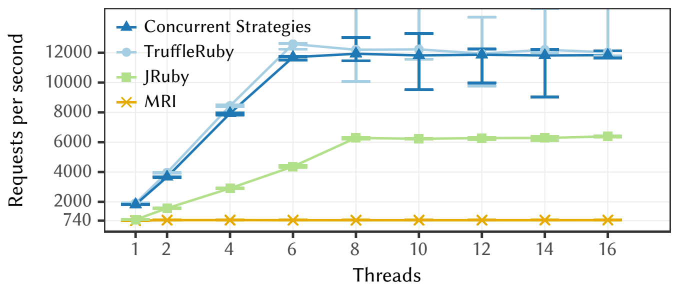


Figure 5.21: Throughput of the WEBrick benchmark for "Hello World" requests. No implementation scales beyond 8 threads as WEBrick creates a new thread for each request. Error bars indicate the first and third quartiles as the throughput has many outliers.

bility 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) [9]. Figure 5.19 shows that the sum variant scales perfectly and Figure 5.20 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 [10] 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 which is run only once and therefore removing the `Mutex` has no impact on peak performance.

WEBrick As a third benchmark, we measure the throughput of WEBrick, the HTTP server in the standard library, simply delivering "Hello World". Figure 5.21 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.

5.8.8 Evaluation 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. The Lightweight Layout Lock improves significantly over StampedLock and ReentrantLock by enabling scalability for concurrent writes (cf. Figure 5.10). As a result, we use it for the `SharedDynamicStorage` strategy and the `ConcurrentBuckets` strategy.

5.9 Summary

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 modifications causing for instance lost updates or low-level exceptions. In this chapter, 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 that are 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 as well.

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 assumption no longer holds, the array is migrated to a more general strategy, which can handle arbitrary changes to the array safely. This strategy uses the novel Lightweight Layout Lock, a refined version of the Layout Lock [13], 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 collection types, 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 look at the cost for the safety provided by our approach by comparing results of all non-micro benchmarks on 1 thread. We observe that the safety provided by our approach comes with an overhead of 5% (geometric mean), ranging from 33% slower up to 1% faster. 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, 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.

Chapter 6

Related Work

In this chapter, we describe how our research relates to prior work. Most of the related work is on synchronization techniques and language runtimes. This section provides an overview and discusses how these techniques relate to ours.

6.1 Global Synchronization with Safepoints

We start by presenting work related to Virtual Machine safepoints and how they relate to our guest-language safepoints presented in Chapter 3.

6.1.1 Safepoints

Host VM safepoints are a common technique found in many VMs. The major JVMs implement safepoints using the optimized techniques we described in Section 3.4. Microsoft's CLR uses a volatile flag. Simpler VMs such as the reference implementation of Ruby and the Rubinius implementation also use the flag implementation technique. Rubinius has multiple flags for separate applications such as GC, passing exceptions to threads, and debugging, which means that each safepoint can consist of multiple load instructions, and as the value of the flag is being explicitly checked it also includes a branch instruction.

In Section 3.1 we said that threading libraries do not generally provide a mechanism to interrupt a running thread which could be used instead of requiring each thread to poll. In fact, in some systems such as FreeBSD and HP-UNIX the `pthread_suspend` and `pthread_continue` calls are available, which will pause and restart a running thread. The Boehm-Demers-Weiser conservative GC [4] uses these calls when available. If they are not available, the GC sends signals to the threads instead. However this only addresses one application of safepoints — pausing threads so that they do not modify the heap while collection phases run. For instance, pausing threads via these calls or signals does not allow to run an action in the paused threads. Guest-language safepoint enable this possibility and therefore have more applications.

6.1.2 Deoptimization

Deoptimization is the process of transferring one or multiple threads running compiled code to the interpreter. This is typically done by manipulating native call stacks to convert the current compiled frame into an interpreter frame, adjust the stack pointer and continue running the application from there. Deoptimizing to the interpreter is useful for operations that are not handled in compiled code, such as debugger breakpoints.

Deoptimization is closely associated with development of the Smalltalk language, where it was described as a technique for implementing a debugger [37]. This kind of deoptimization also required VM safepoints, which were originally called "interrupt points" [21], as those were the locations where the debugger could interrupt the running program. The emphasis was on making the program at that point able to be inspected by having debug information available.

A key part of our technique for guest-language safepoints is that we use deoptimization to jump from code that does not actively poll for guest-language safepoints to code that does. Guest-language safepoints reuse the VM's deoptimization mechanism to avoid any overhead when running in compiled code. There are no guest-language safepoint polls in compiled code. When a guest-language safepoint is triggered, all threads are deoptimized. Guest-language safepoints polls are always checked in the interpreter, and all threads eventually runs into such a check and enter the guest-language safepoint.

6.1.3 SwitchPoint

`SwitchPoint` is a mechanism for global speculation (e.g., for speculating that a given class has no subclasses, even though that might change when loading more classes). The `SwitchPoint` class is part of the `java.lang.invoke` package, which contains various dynamic language support classes, that are part of JSR 292 [76]. The implementation of `SwitchPoint` uses VM safepoints and adds no overhead to peak performance. In that regard, it is very similar to Truffle's `Assumption` class and can be used as an alternative implementation to reuse the VM safepoint functionality for global speculations. In contrast to our guest-language safepoints API, it provides no way to run arbitrary code in other threads. As far as we know, current use cases of `SwitchPoint` are limited to global speculations, which is a small subset of the applications that we presented in Chapter 3 and uses safepoints as a cheap check but not as an opportunity to run code in other threads.

One example use case for a global speculation is an "almost constant value" [25], such as speculating that a `static` field does not change its value and being able to consider it as a constant during compilation. If the value is changed, deoptimization is triggered to remove all compiled methods depending on that speculation. JRuby [65], an implementation of Ruby on top of the JVM and a heavy user JSR 292, uses `SwitchPoint` for speculating that Ruby constants are not redefined and for speculating that Ruby methods are not redefined. Invalidation of a `SwitchPoint` always triggers deoptimization.

6.2 Synchronization Based on Reachability

In this section, we discuss work related to reachability when combined with synchronization. To the best of our knowledge, distinguishing objects based on reachability was first introduced by Domani et al. [23] in the context of garbage collection. Kalibera et al. [42] compared reachability and precise tracking of concurrent object accesses. Finally, we see that our techniques for synchronization based on reachability already inspired new research, such as Ungar et al. [92].

6.2.1 Memory Management and Garbage Collection

For garbage collectors, Domani et al. [23] introduced a distinction between local and global objects based on reachability that is identical to our distinction of local and shared objects. The distinction is not made with the shape but with a bit per object, stored in a separate bitmap. They also describe the corresponding write barrier to dynamically monitor when objects become global and share their transitive closure, much like we do in Section 4.4.2. Their goal is to allow thread-local synchronization-free garbage collection. One of their optimizations is to allocate objects directly in the global heap when they know the object is likely to escape the current thread, as described in Section 4.5.1. They find this optimization crucial in their approach, as it reduces the sweeping time significantly. Global objects cannot be moved during thread-local garbage collection as the thread-local GC cannot know all references from other heaps. Global objects allocated in a thread-local heap partition the heap into many free spaces, and the number of free spaces proportionally increases the sweeping time in their GC. It is therefore essential to limit the number of global objects in thread-local heaps. We do not have a related concept to this unfortunate side-effect in our model.

6.2.2 Tracking Reachability and Precise Sharing

As part of an effort to define concurrency metrics for the DaCapo benchmarks, Kalibera et al. [42] compare two techniques to track object sharing between threads. The first one tracks reachability as in our model and the second *precise* technique identifies objects as shared only if they are accessed by multiple threads through their lifetime. They find a significant gap between the ratio of shared objects (number and total size) reported by the two techniques on these benchmarks. The gap shrinks when looking at the proportion of reads and writes on shared objects, but remains as high as 19%. In our evaluation, we demonstrate that our overhead remains limited to at most 12% even on benchmarks where almost all objects are shared. Therefore, tracking reachability seems a good trade-off of performance versus precision.

6.2.3 Reachability Used for Efficient Reference Counting

Ungar et al. [92] experimented with improving the performance of reference counting in the Swift programming language. Due to Swift supporting shared-memory parallelism, reference counting must appear atomic for correctness. In the current Swift runtime, atomic addition and subtraction instructions are used for reference counting, leading to a significant overhead. Ungar et al. [92] leveraged our technique of tracking reachability and distinguishing between *local* and *shared* objects for the purpose of optimizing reference counting. By applying our technique in this context, local objects can use the traditional fast addition and subtraction instructions for reference counting and only shared objects need to use more expensive atomic instructions. They report savings up to 50% of the execution time on the Computer Language Benchmarks Game and 54% on Richards.

6.3 Synchronization Techniques

Other related work also had the goal of minimizing the overhead of synchronization primitives, which are often considered a potential performance bottleneck. Here we review the most relevant research.

6.3.1 Biased Locking

Biased locking [94] is arguably the most popular technique in this domain. Biased locking relies on the assumption that locks are rarely contended, because even though specific objects might be accessed in parallel, most of the time, they are only used by a single thread. When this assumption holds, lock acquisition is never attempted and a more efficient lock-less operation is performed instead. VM-level run-time checks and synchronizations ensure that once a lock becomes contended the correct behavior is enforced. The technique is implemented in several VMs (e.g., in Oracle's HotSpot JVM [22]), and is often combined with JIT compilation [77]. Our techniques share the goal of avoiding unnecessary operations for thread-local objects and collections. The strategy for biased locks is even more optimistic than ours by using a full lock only when it is actually accessed by multiple threads. However, it also needs to keep track of the thread owning the lock and check at *each access* if the owner is the same as the current thread. Our technique does not impose any overhead for object reads.

Furthermore, the biased locking technique described by Russell and Detlefs [77] uses VM safepoints to revoke the bias from a specific lock. This happens when a thread tries to access a lock biased towards another thread. The biased lock is then replaced with some other locking implementation which can handle multiple owners over time more efficiently, i.e., without needing safepoints. Russell and Detlefs also use safepoints for bulk re-biasing and revocation to ensure a consistent view of the object headers and their lock states. These bulk operations operate on all objects of a given type to amortize the cost of individual revocation. Our approach for efficient synchronization of object field accesses does not use safepoints, and therefore avoids that cost.

Biased locking is not always easy to apply to guest languages because they might provide unstructured locking (`lock`/`unlock` instead of only `synchronized` blocks) or they might require the `lock()` operation to be interruptible (required for guest-language safepoints). Guest-language safepoints could be used to implement biased locks at the guest language level and their expression would be rather elegant with our API.

Note that the synchronization used for shared object field writes in our thread-safe object model uses biased locks in HotSpot.

6.3.2 Biased Reader-Writer Locks

LarkTM [102], a Software Transactional Memory implementation uses VM safepoints for coordinating the resolution of conflicting lock operations with their biased reader-writer locks [8]. They partition the threads into two categories: threads executing a blocking operation (waiting, I/O, running native code, etc) and all others threads. An *explicit* protocol is used to handle the threads that are *not* executing a blocking operation, in which case the responding thread resolves the conflict. In the *implicit* protocol, used for the blocked threads, the requesting thread does the operation on behalf of the blocked thread, which must wait for that operation to complete before returning from its blocking operation. This approach only works if the code does not depend on the thread in which it is run (call stack, thread-local state, etc) and needs careful memory barriers to ensure visibility of the changes. Our guest-language safepoints API avoids these limitations by providing the ability to run code in any participating thread. We also discussed how to avoid blocked threads such that an *implicit* protocol is not needed in Section 3.4.4.

6.3.3 Software Transactional Memory

In addition to locking and similar explicit synchronization primitives, the distinction between objects that require synchronization and others that do not has been applied to other synchronization techniques, too. As an example, several implementations of software transactional memory (STM) reduce the overhead of the STM runtime by avoiding or minimizing unnecessary operations. One notable example is LarkTM [102], which assumes every object to be read-only until a transaction attempts to modify it. Similarly, there are examples of STMs that have been integrated with language runtimes and JIT compilers [1, 48] to apply common techniques such as escape analysis to reduce the overhead of STM read and write barriers. Our approach is integrated with the language runtime at a similar level of abstraction, but unlike common STM algorithms it does not impose any run-time overhead (e.g., due to logging) as long as objects and collections are not *shared*.

6.3.4 Layout Lock

As detailed in Section 5.6, the Lightweight Layout Lock is an extension of Cohen et al. [13]'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 needs 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 5.11). 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.

6.3.5 VM-level Synchronization Techniques

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

Odaira et al. [66] 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 than our approach (at best a 5.5x speedup with 12 threads) as well as an overhead for single-threaded performance ($\geq 25\%$ slower than the GIL for each benchmark).

Meier et al. [55, 56] 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 (geometric average: 30% slower) and scalability.

STM and HTM systems, like the ones discussed above, add overhead to all shared memory accesses, since they do not distinguish between accesses to the data structures of the interpreter 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.

6.4 Thread Safety of Data Representations

Representing data efficiently in dynamic languages is an active area of research. However, many approaches ignore thread safety when accessing the data representations, and rely on the user to synchronize all accesses externally. We detail the relevant approaches here.

6.4.1 Object Representations

As discussed in Section 4.2.1, as far as we know, only a few dynamic language runtimes use object representations that are safe for use with multithreading.

Jython's object model implements Python objects based on Java's `ConcurrentHashMap` class [41]. Thus, all object accesses are synchronized and safe with respect to our definition. However, as in the sequential case for which SELF's maps [11] were designed, using a hash table cannot compete in terms of performance.

JRuby's object model [65] uses an array of references to store the field values and thus uses an approach similar to SELF's maps. Compared to the Truffle object model [99], it does not support specializing on field types, e.g., for integers or floats. For adding fields to an object, the array needs to be replaced with a larger one, which requires synchronization to guarantee safety and avoid losing concurrent updates and definitions. Similar to our approach, a field read is done without synchronization. However, JRuby synchronizes field writes for all objects regardless of whether they are shared or not.

By distinguishing between local and shared objects, our approach avoids any overhead on sequential applications. In combination with the type-specialization of the Truffle object model, it also provides additional performance benefits.

6.4.2 Adaptive Data Representations and Collections

To optimize the data representations of collections, Bolz et al. [7] 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. [19] 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, Chapter 5 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. [63] 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 are composed of an immutable data structure and an additional mutable variable that points to the current version of the data structure. That way, the data structure can be updated by mutating the variable and yet allow unsynchronized reads. 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. Newton et al.'s solution is not applicable to dynamic programming languages due to the strict limitation of relying on purely functional data structures.

6.5 Shared-Memory Parallel Language Runtimes

As discussed earlier, most dynamic languages do not support shared-memory parallelism. The few that do rarely provide thread-safety guarantees, or when they do they severely restrict what data structures can be passed between multiple threads of execution. Here, we review proposals for, and existing implementations of, parallel shared-memory dynamic language runtimes.

6.5.1 Shared-Memory JavaScript

In JavaScript, only primitive data can be shared between parallel workers using a `SharedArrayBuffer` [60]. Pizlo [73] published a blog post with some ideas for efficient shared-memory including arbitrary objects. The approach 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 in the thread that created them. On access, such objects and collections check that the current thread is the creator thread. If the check fails, heavier synchronization is used such as a per-object lock. In contrast to that, our notion of *local* implies no overhead, but applies to fewer objects. Pizlo proposes either using exclusive per-array locking or *compare-and-swap* for array resizes. However, he does not explain 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 fall back to a less efficient representation.

6.5.2 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 constructor argument named `synchronized` can be used to create a synchronized hash table.[1] LispWorks synchronizes by default, paying a performance cost that can be avoided by passing the `single-thread` flag to the constructor.[2] With our techniques, this manual approach could be avoided, sources for errors could be removed, and performance for single threaded execution could be improved.

6.5.3 Racket

Racket is a dynamically-typed language from the Lisp family. The reference implementation tried two approaches to add parallelism: *futures* and *places*. Swaine et al. [89] introduce futures as a way to add parallelism in an existing language runtime incrementally with minimal efforts. They partition primitive operations in 3 categories: *safe*, *unsafe* and *synchronized*. Only *safe* operations can be run in parallel, and typically do not touch the language implementation's

[1]SBCL User Manual: `http://www.sbcl.org/manual/#Hash-Table-Extensions`

[2]Documentation of `make-hash-table`: `http://www.lispworks.com/documentation/lw60/LW/html/lw-608.htm`

internal state. *Unsafe* operations are delegated to and executed by the initial thread for safety and *synchronized* operations use a global lock. Swaine et al. [89] argues that *fast-path* operations generally correspond to *safe* operations, but a later paper [90] shows that this is not so intuitive and introduces a visualization tool to understand the bottlenecks to achieve parallelism in this model. In general, even though the model is simple and easy to implement, it seems difficult to achieve good performance, as only the low-level operations run in parallel. As such, the parallel programs look a lot less idiomatic than the corresponding sequential programs in Racket. Collection operations seem to be considered as *unsafe* or *synchronized* and therefore are serialized.

Racket's *places* [91] proposes an actor-like concurrency model for Racket, but also allows sharing mutable fixed-size arrays of primitive values (bytes, integers, floating-point numbers). Except for these shared primitive arrays, places are fully isolated and have their own heap. Thus, garbage collection can run for each place independently. The authors report it took approximately two graduate-student years to implement *places*. This is more than for the *futures* but is still reasonable. They also evaluated their implementation with the NAS Parallel Benchmarks [3] and reported scalability results similar to our approach. They had to modify the benchmarks to make them work in a more actor-like programming style, sharing only primitive arrays and not any other object. Places do not allow sharing mutable data structures containing objects and therefore avoid the concurrent mutation problem. We believe that this was done by design to reduce the implementation effort (they experienced many races and bugs when trying full shared-memory parallelism), to avoid synchronization overhead and to enable place-local garbage collection. The only data structure they allow to share are these fixed-size primitive arrays, similar to `SharedFixedStorage` but only for primitive values. Similarly to us, they do not need synchronization for these arrays as they rely on the underlying processor memory model and atomic reads and writes.

Chapter 7

Conclusion

7.1 A New Era of Parallel Dynamic Languages

In this thesis we worked on a solution to provide efficient parallelism for dynamically-typed languages. We noticed that the state of the art implementations of dynamic languages using shared-memory parallelism are either inefficient or unsafe. Implementations which are unsafe seem to do so because they consider that synchronization is too expensive for performance. Implementations which are safe pay a significant cost in performance due to the synchronization being applied in all cases, even for single-threaded programs. As an example, the standard implementations of Ruby and Python use a Global Interpreter Lock which provides safety but prevents the execution of user code in parallel in the same process entirely. A key problem for safety and performance is to synchronize access to core data structures, that is the built-in objects and the built-in collections provided by the language implementation.

To address this we invented multiple techniques to correctly synchronize access to core data structures of dynamic languages, maintain single-threaded efficiency and scale when accessing core data structures in parallel.

First, we lifted a key mechanism for global virtual machine synchronization called *safepoints* to make it available for guest languages. This allows us, with zero overhead on peak performance until it is triggered, to synchronize all threads together and make them perform an arbitrary action and then resume their execution. Beyond being a powerful synchronization mechanism notably for core data structures, this mechanism can also be used for thread cancellation, running a signal handler on an existing thread, enumerating all objects of the heap, examining the call stacks, debugging programs and even deterministic parallel execution.

Second, we looked at built-in objects, the foundation of object-oriented programming languages and how to synchronize them correctly and efficiently. Fields can be added or removed at any time for objects of dynamic languages. The single-threaded solution to represent such objects efficiently is the long-known *maps* technique from Self [11]. State of the art representations for such objects supporting concurrent access are much less efficient though. Jython for instance resorts to using a `ConcurrentHashMap` for this, which significantly hinders performance. JRuby synchronizes on every object write, leading up to an estimated 2.5x slowdown for the NBody single-threaded benchmark. To address this, we came up with the idea to synchronize based on reachability. Objects which are reachable from only one thread do not need synchronization: they cannot be accessed concurrently. Therefore, we only synchronize on objects which are reachable from multiple threads, which we call *shared* objects. Our design maintains the efficient single-threaded representation for objects, and enables parallel reads for fields of the same object. Only shared object field writes need to be synchronized in our approach.

To know which objects are shared and which are not, we use a write barrier when writing to a shared object. We optimize this write barrier by using an inline cache structure mirroring the object graph structure of the value being written, which makes the write barrier 35 times faster in our micro benchmark by converting a recursive generic process into a series of simple checks and writes specialized for the specific object graph structure of the value. Our evaluation shows that single-threaded benchmarks incur zero overhead on peak performance for non-shared objects. Parallel actor benchmarks show an average overhead of 3% when writing to shared object fields very frequently, a fairly low overhead considering our object model provides safety.

Third, we looked at built-in collections, that is collections provided by the language implementation. Synchronizing concurrent collections has been the subject of much research, but most of it is very hard to apply to dynamic language collections, which are so versatile and have so many operations. Moreover, most concurrent collections are significantly slower than single-threaded collections. Here again we applied our idea of synchronization based on reachability, this time to collections. Collections which are reachable from only one thread do not need synchronization and can use the most efficient single-threaded representation. Collections which are reachable from multiple threads use synchronization and possibly a different representation to perform safely and support parallel access.

We added another refinement here by supporting two representations for concurrent arrays. One which performs as fast as single-threaded for arrays of which the size and the storage do not change and another to support all operations and synchronize with a lock. Existing state-of-the-art locks proved inefficient for our usage, as we wanted both parallel read and write accesses, as well as supporting large modification of the data structure such as removing an element from the middle of an array. Therefore we worked together with Arie Tal, which invented the Lightweight Layout Lock to satisfy these requirements. Our experiments show that our thread-safe collections have no overhead on single-threaded benchmarks, scale linearly for array and dictionary accesses, achieve the same scalability as Fortran and Java for classic parallel algorithms and scale better than other Ruby implementations on Ruby workloads. When comparing the results of all parallel macro benchmarks using a single worker thread, we observe an average overhead of only 5% for our thread-safe collections compared to unsafe collections.

Together, these techniques enable efficient and safe dynamically-typed language implementations. We demonstrated our performance claims using TruffleRuby, the fastest Ruby implementations on classic benchmarks [52]. We believe our techniques are portable to other dynamic languages and expect to use them for other GraalVM languages such as Python and JavaScript.

We hope this research enables widespread use of parallelism in shared-memory dynamically-typed languages, in order to use multi-core processors more efficiently.

Chapter 8

Future Work

This thesis opens up a number of new avenues for future research.

8.1 Optimizing Guest-Language Safepoints

Our current implementation of the guest-language safepoints API provides the functionality that we need to implement existing Ruby functionality and other useful new Ruby features. Our API imposes zero overhead on peak performance and insignificant compilation overhead, which means that it causes no overhead for programs which do not actually use a guest-language safepoint. Latency is reasonable, but could probably be lowered.

However, our current implementation uses deoptimization of all application threads which is both time consuming and also impairs peak performance after the safepoint is finished until the compiler recompiles the methods. Future work will focus on mitigating these costs so that safepoints can be used for high performance operations such as inter-thread message passing. This can be achieved by moving the API down into the Truffle framework and by exposing a mechanism from the Virtual Machine to deoptimize threads so that they go back to the interpreter and poll for guest-language safepoints without invalidating compiled code. When this is done we aim to measure the costs for various use cases of the safepoint API (e.g., the time needed to attach or detach a debugger), rather than just the costs of the API primitives themselves.

Currently, the VM safepoints that we use are global safepoints pausing all host-language threads in the system. We intend to explore optimizing the case of pausing just a single thread, which is common in many applications of guest-language safepoints and could significantly reduce the latency. It would also improve global throughput as other threads would not be interrupted.

When our safepoint API has been moved down into the Truffle framework, another interesting area of research will be cross-language safepoints. Threads running code written in multiple languages will then be able to cause each other to enter safepoints by using the common Truffle safepoint API. We are planning to use this technique to allow safepoints to work within Ruby C extensions by applying the same API both within the Ruby interpreter and our C interpreter.

Specifically in the implementation of TruffleRuby we see safepoints as potentially the key primitive to provide an alternative to locking VM data structures. Data structures that are frequently read but infrequently modified can be written to in a safepoint, meaning that no lock is required for readers.

8.2 Alternative to Avoid Out-Of-Bounds Checks

Currently, the object model relies on out-of-bounds checks to handle the race between updating an object's shape and one of its extension arrays (cf. Section 4.3.2). To further improve performance, such out-of-bounds checks could be removed. To this end, it needs to be ensured that the shape is never newer than the extension arrays to avoid out-of-bounds accesses. This could be ensured by encoding the dependency between these elements with memory barriers and compiler intrinsics, ideally without restricting optimizations on the corresponding performance-sensitive memory operations.

8.3 Handling Deeply-Immutable Objects

An object is immutable when its field values can no longer be changed. This might be enforced at the language level or by the runtime. An object is deeply-immutable if it is itself immutable and any object it refers to is also immutable. Such a deeply-immutable object does not need to be protected when shared to other threads. In our object model, it means that the *write barrier* could check if the object is deeply-immutable and in that case would not need to change the shape of the object or any object it references.

8.4 Synchronizing on Write Accesses

Another performance-related aspect is the use of Java object monitors for synchronizing shared object writes. Java monitors are optimized in the HotSpot VM by implementing biased locking, which minimizes the overhead when a shared object is only used by a single thread. However, there might be better-performing lock implementations for our object model. Particularly, when lock contention is high, Java's monitors are suboptimal. Instead, some custom lock could be used, for instance based on Java 8's `StampedLock` or based on the `LayoutLock` [13], which could scale better for workloads performing object field writes concurrently on the same object.

8.5 Further Thread-Safety Guarantees

Combining thread-safe objects and thread-safe collection implementations 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.

8.6 Removing Redundant Synchronization

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 that might result in unnecessary overhead.

8.7 Further Concurrent Strategies

It would also be worthwhile to investigate further strategies in addition to the proposed ones. While collections in dynamic language 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. However, since this potentially leads to endless strategy changes and unstable performance, it requires careful investigation.

8.8 Granularity of Synchronization: Transaction Scope

Worthington [96] wrote a blog post about the implicit *transaction scope* necessary to correctly synchronize a shared-memory program in the context of Perl 6. In his blog post, he shows that synchronizing arrays and dictionaries is not enough to make a parallel wordcount program thread-safe. Instead, the *transaction scope*, i.e., the scope to which synchronization should be applied, needs to be carefully considered. Future work could explore how to infer such a transaction scope automatically and extend synchronization to that scope. That would have the potential to automatically and correctly synchronize user code. However, this seems a very challenging problem in the presence of metaprogramming available in dynamic languages.

8.9 Using Reachability for Lock Objects

If locks are exposed in the language as objects, then reachability is already tracked for them with our approach, just like for any other object. This is for instance the case of `Mutex` in Ruby. If the lock is known to be Local, then it can only be acquired and released from its allocating thread. Synchronization would only be performed once the lock becomes Shared. Such a lock would also need to track whether it is acquired or not, even when Local, in case it becomes Shared while the lock is acquired. In such a case, it is also necessary for the current thread to actually lock the underlying synchronization primitive, such that another thread wanting to acquire the lock would have to wait until the synchronization primitive is unlocked. This technique is similar to biased locking [77], but does not require to keep a reference to the allocating thread, and enables cheaper checks based on the identity of the Shape instead of reading the biased thread out of the object header and comparing it to the current thread. Furthermore, this technique does not require safepoints. Safepoints are needed for biased locking when a thread tries to acquire a lock biased against another thread, to atomically change the object header, and make the other thread lock the underlying synchronization primitive. Reachability for lock objects avoids the need for safepoints at the expense of using a synchronization primitive as soon as the lock is accessible by multiple threads, instead of delaying it to the first acquire operation by another thread as done by biased locking. Reachability for lock objects could also be used in combination with biased locking, with reachability as the first layer for cheaper checks and biased locking as a second layer when the lock becomes Shared.

8.10 Exposing Reachability for User-Defined Data Structures

So far, this thesis used reachability to optimize synchronization on objects and built-in collections. Reachability in the form of a predicate to know whether an object is *local* or *shared* (reachable by multiple threads) could be exposed to the data structure library developer. Such a predicate would allow more efficient synchronization for these data structures, which could take advantage of knowing whether they are reachable and accessed only by a single thread. Furthermore, we could introduce a callback triggered just before the data structure becomes *shared* between multiple threads (e.g., `on_share`). For instance, the user-defined data structure could start *local* and not use synchronization at all. Once the data structure would become *shared* between multiple threads, the hook could trigger and, for example, a lock could be added and attached as a field of the data structure. Each method of the data structure would then check if there is a lock attached and only synchronize if there is one. We did not explore this further as exposing reachability to the user has the potential of being misused by the user. However, this approach seems promising to optimize synchronization on user-defined data structures.

Bibliography

[1] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and Runtime Support for Efficient Software Transactional Memory. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 26–37. ACM, 2006.

[2] Apple. JavaScriptCore, the built-in JavaScript engine for WebKit, 2018. URL `https://trac.webkit.org/wiki/JavaScriptCore`.

[3] 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. The NAS Parallel Benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991.

[4] Hans-Juergen Boehm and Mark Weiser. Garbage Collection in an Uncooperative Environment. *Software: Practice and Experience*, 18(9):807–820, September 1988. ISSN 0038-0644.

[5] Carl Friedrich Bolz and Laurence Tratt. The Impact of Meta-Tracing on VM Design and Implementation. *Science of Computer Programming*, 2013. ISSN 0167-6423.

[6] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 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, pages 18–25. ACM, 2009.

[7] Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. 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, pages 167–182. ACM, 2013. ISBN 978-1-4503-2374-1.

[8] Michael D Bond, Milind Kulkarni, Man Cao, Minjia Zhang, Meisam Fathi Salmi, Swarnendu Biswas, Aritra Sengupta, and Jipeng Huang. OCTET: Capturing and Controlling Cross-Thread Dependences Efficiently. In *ACM SIGPLAN Notices*, volume 48, pages 693–712, 2013.

[9] Gregory Brown, Brad Ediger, Alexander Mankuta, et al. Prawn: Fast, Nimble PDF Generation For Ruby, 2017. URL `http://prawnpdf.org/`.

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

[11] Craig Chambers, David Ungar, and Elgin Lee. 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, pages 49–70. ACM, October 1989. ISBN 0-89791-333-7.

[12] Daniel Clifford, Hannes Payer, Michael Stanton, and Ben L. Titzer. Memento Mori: Dynamic Allocation-site-based Optimizations. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 105–117. ACM, 2015. ISBN 978-1-4503-3589-8.

[13] Nachshon Cohen, Arie Tal, and Erez Petrank. 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, pages 17–29. ACM, 2017. ISBN 978-1-4503-4493-7.

[14] Diego Costa, Artur Andrzejak, Janos Seboek, and David Lo. Empirical Study of Usage and Performance of Java Collections. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE'17, pages 389–400. ACM, 2017. ISBN 978-1-4503-4404-3.

[15] Benoit Daloze, Chris Seaton, Daniele Bonetta, and Hanspeter Mössenböck. Techniques and Applications for Guest-Language Safepoints. In *Proceedings of the 10th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS'15, pages 1–10, 2015.

[16] Benoit Daloze, Stefan Marr, Daniele Bonetta, and Hanspeter Mössenböck. Efficient and Thread-Safe Objects for Dynamically-Typed Languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages & Applications*, OOPSLA 2016, pages 642–659, 2016. ISBN 978-1-4503-4444-9.

[17] Benoit Daloze, Arie Tal, Stefan Marr, Hanspeter Mössenböck, and Erez Petrank. Parallelization of Dynamic Languages: Synchronizing Built-in Collections. In *Proceedings of the 2018 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages & Applications*, OOPSLA 2018, 2018.

[18] Jerry D'Antonio and Petr Chalupa. Concurrent Ruby — Modern concurrency tools for Ruby, 2017. URL `http://www.concurrent-ruby.com`.

[19] Mattias De Wael, Stefan Marr, Joeri De Koster, Jennifer B. Sartor, and Wolfgang De Meuter. 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, pages 61–75. ACM, October 2015. ISBN 978-1-4503-3688-8.

[20] Ulan Degenbaev, Jochen Eisinger, Manfred Ernst, Ross McIlroy, and Hannes Payer. Idle Time Garbage Collection Scheduling. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'16, pages 570–583. ACM, 2016. ISBN 978-1-4503-4261-2.

[21] L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 297–302. ACM, 1984. ISBN 0-89791-125-3.

[22] David Dice. Implementing Fast Java Monitors with Relaxed-locks. In *Proceedings of the 2001 Symposium on Java Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, page 13. USENIX Association, 2001.

[23] Tamar Domani, Gal Goldshtein, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. Thread-Local Heaps for Java. In *Proceedings of the 3rd International Symposium on Memory Management*, ISMM '02, pages 76–87. ACM, 2002. ISBN 1-58113-539-4.

[24] Robert W. Floyd. Algorithm 97: Shortest Path. *Commun. ACM*, 5(6):345, Jun. 1962.

[25] Rémi Forax. JSR 292 Goodness: Almost static final field., 2011. URL `https://community.oracle.com/blogs/forax/2011/12/17/jsr-292-goodness-almost-static-final-field`.

[26] GitHub. Contributors to TruffleRuby, 2018. URL `https://github.com/oracle/truffleruby/graphs/contributors?from=2013-01-01&to=2018-09-30`.

[27] Liang Gong, Michael Pradel, and Koushik Sen. JITProf: Pinpointing JIT-unfriendly JavaScript Code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 357–368. ACM, 2015. ISBN 978-1-4503-3675-8.

[28] Google. V8 — Google's JavaScript engine, 2017. URL `https://developers.google.com/v8/`.

[29] GraalVM developers. GraalVM — Run Programs Faster Anywhere, 2018. URL `http://www.graalvm.org/`.

[30] Matthias Grimmer, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. Dynamically Composing Languages in a Modular Way: Supporting C Extensions for Dynamic Languages. In *Proceedings of the 14th International Conference on Modularity*, MODULARITY 2015, pages 1–13. ACM, 2015. ISBN 978-1-4503-3249-1.

[31] David Heinemeier Hansson et al. 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, 2018. URL `http://rubyonrails.org/`.

[32] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010. ISBN 1608452352, 9781608452354.

[33] Timothy L. Harris. A Pragmatic Implementation of Non-Blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314. Springer, 2001. ISBN 3-540-42605-1.

[34] Stephan Herhut, Richard L. Hudson, Tatiana Shpeisman, and Jaswanth Sreeram. River Trail: A Path to Parallelism in JavaScript. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 729–744. ACM, 2013. ISBN 978-1-4503-2374-1.

[35] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2011.

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

[37] Urs Hölzle, Craig Chambers, and David Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI)*, pages 32–43, 1992.

[38] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *ECOOP '91: European Conference on Object-Oriented Programming*, volume 512 of *LNCS*, pages 21–38. Springer, 1991.

[39] Shams M. Imam and Vivek Sarkar. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, AGERE! '14, pages 67–80. ACM, 2014.

[40] JRuby. JRuby's threaded-reverse benchmark, 2008. URL `https://github.com/jruby/jruby/blob/a0a3e4bd22/test/bench/bench_threaded_reverse.rb`.

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

[42] Tomas Kalibera, Matthew Mole, Richard Jones, and Jan Vitek. A Black-box Approach to Understanding Concurrency in DaCapo. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 335–354. ACM, 2012.

[43] Faiz Khan, Vincent Foley-Bourgon, Sujay Kathrotia, Erick Lavoie, and Laurie Hendren. Using JavaScript and WebCL for Numerical Computations: A Comparative Study of Native and Web Technologies. In *Proc. of DLS*, pages 91–102, 2014.

[44] Hemant Kumar. Bug report on the RubyGems project: Thread issues with JRuby, 2013. URL `https://github.com/rubygems/rubygems/issues/597`.

[45] Jim Laskey. Nashorn Multithreading and MT-safety, 2013. URL `https://blogs.oracle.com/nashorn/entry/nashorn_multi_threading_and_mt`.

[46] Yi Lin, Kunshan Wang, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. Stop and Go: Understanding Yieldpoint Behavior. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 70–80. ACM, 2015. ISBN 978-1-4503-3589-8.

[47] Li Lu, Weixing Ji, and Michael L Scott. Dynamic Enforcement of Determinism in a Parallel Scripting Language. In *Proc. of PLDI*, page 53, 2014.

[48] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the Overhead of Nonblocking Software Transactional Memory. Technical report, Department of Computer Science, University of Rochester, 2006.

[49] Stefan Marr. SOMns — A Newspeak for Concurrency Research, 2017. URL `https://github.com/smarr/SOMns/commit/7e4a157a`.

[50] Stefan Marr and Benoit Daloze. 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, 2018.

[51] Stefan Marr, Chris Seaton, and Stéphane Ducasse. Zero-Overhead Metaprogramming: Reflection and Metaobject Protocols Fast and without Compromises. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 545–554. ACM, 2015.

[52] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. Cross-Language Compiler Benchmarking: Are We Fast Yet? In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS 2016, pages 120–131. ACM, 2016. ISBN 978-1-4503-4445-6.

[53] Nicholas D. Matsakis. Parallel Closures: A New Twist on an Old Idea. In *Proc. of USENIX HotPar*, 2012.

[54] Yukihiro Matsumoto, Koichi Sasada, Nobuyoshi Nakada, et al. MRI — Matz's Ruby Interpreter — The Ruby Programming Language, 2018. URL `https://www.ruby-lang.org/en/`.

[55] Remigius Meier, Armin Rigo, and Thomas R. Gross. Parallel Virtual Machines with RPython. In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS '16, pages 48–59. ACM, 2016. ISBN 978-1-4503-4445-6.

[56] Remigius Meier, Armin Rigo, and Thomas R. Gross. 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, 2018.

[57] Kevin Menard, Chris Seaton, and Benoit Daloze. Specializing Ropes for Ruby. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, ManLang'18. ACM, 2018. ISBN 978-1-4503-6424-9.

[58] Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. Evaluating the Design of the R Language: Objects and Functions for Data Analysis. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 104–131. Springer-Verlag, 2012.

[59] Mozilla. SpiderMonkey — Mozilla's JavaScript engine, 2015. URL `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`.

[60] Mozilla. SharedArrayBuffer Documentation, 2018. URL `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer`.

[61] Mozilla Developer Network. JS-THREADSAFE, 2015. URL `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/JSAPI_reference/JS_THREADSAFE`.

[62] Nashorn authors. Project Nashorn, 2016. URL `http://openjdk.java.net/projects/nashorn/`.

[63] Ryan R. Newton, Peter P. Fogg, and Ali Varamesh. Adaptive Lock-Free Maps: Purely-Functional to Scalable. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 218–229. ACM, 2015. ISBN 978-1-4503-3669-7.

[64] Takafumi Nose. Ruby version of NAS Parallel Benchmarks 3.0, 2013. URL `https://github.com/plus7/npb_ruby`.

[65] Charles Nutter, Thomas Enebo, Ola Bini, Nick Sieger, et al. JRuby — The Ruby Programming Language on the JVM, 2018. URL `http://jruby.org/`.

[66] Rei Odaira, Jose G. Castanos, and Hisanobu Tomari. 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, pages 131–142. ACM, 2014. ISBN 978-1-4503-2656-8.

[67] Oracle. OpenJDK, 2015. URL `http://openjdk.java.net/`.

[68] Oracle. Class SwitchPoint, 2015. URL `http://docs.oracle.com/javase/8/docs/api/java/lang/invoke/SwitchPoint.html`.

[69] Oracle Labs. GraalJS - A High-Performance JavaScript Engine Built on GraalVM, 2018. URL `https://github.com/graalvm/graaljs`.

[70] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice.* Addison-Wesley Professional, 2005. ISBN 0321349601.

[71] Mike Perham. Sidekiq — Simple, efficient background processing for Ruby, 2017. URL `https://sidekiq.org/`.

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

[73] Filip Pizlo. Concurrent JavaScript: It can work!, 2017. URL `https://webkit.org/blog/7846/concurrent-javascript-it-can-work/`.

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

[75] Rails Project. Threading and Code Execution in Rails, 2018. URL `http://edgeguides.rubyonrails.org/threading_and_code_execution.html`.

[76] John Rose et al. JSR 292: Supporting Dynamically Typed Languages on the Java Platform, 2011. URL `https://jcp.org/en/jsr/detail?id=292`.

[77] Kenneth Russell and David Detlefs. Eliminating Synchronization-related Atomic Operations with Biased Locking and Bulk Rebiasing. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 263–272. ACM, 2006.

[78] Chris Seaton. *Specialising Dynamic Techniques for Implementing the Ruby Programming Language*. PhD thesis, University of Manchester, 2015.

[79] Chris Seaton, Michael L Van De Vanter, and Michael Haupt. Debugging at Full Speed. In *Proc. of Dynamic Languages and Applications (DYLA)*, 2014.

[80] Chris Seaton, Benoit Daloze, Kevin Menard, Thomas Würthinger, et al. A TruffleRuby fork with the branches used for evaluation, 2015. URL `https://github.com/eregon/jruby/tree/safepoint`.

[81] Chris Seaton, Benoit Daloze, Kevin Menard, Petr Chalupa, Brandon Fish, and Duncan MacGregor. TruffleRuby — A High Performance Implementation of the Ruby Programming Language, 2018. URL `https://github.com/oracle/truffleruby`.

[82] Chris Seaton et al. Bench9000, 2014. URL `https://github.com/eregon/bench9000`.

[83] Aleksey Shipilëv. Safe Publication and Safe Initialization in Java, 2014. URL `https://shipilev.net/blog/2014/safe-public-construction/`.

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

[85] Brian Shirai. Bug report on the Bundler project: Unsychronized concurrent updates of Hash, 2018. URL `https://github.com/bundler/bundler/issues/6274`.

[86] Vincent St-Amour and Shu-yu Guo. Optimization Coaching for JavaScript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *LIPIcs*, pages 271–295. Schloss Dagstuhl — Leibniz-Zentrum fuer Informatik, 2015. ISBN 978-3-939897-86-6.

[87] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 165–174. ACM, 2014. ISBN 978-1-4503-2670-4.

[88] Håkan Sundell and Philippas Tsigas. Lock-Free and Practical Doubly Linked List-Based Deques Using Single-Word Compare-and-Swap. *Principles of Distributed Systems*, pages 240–255, 2005. ISSN 03029743.

[89] James Swaine, Kevin Tew, Peter Dinda, Robert Bruce Findler, and Matthew Flatt. 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, pages 583–597. ACM, 2010. ISBN 978-1-4503-0203-6.

[90] James Swaine, Burke Fetscher, Vincent St-Amour, Robert Bruce Findler, and Matthew Flatt. Seeing the Futures: Profiling Shared-memory Parallel Racket. In *Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '12, pages 73–82. ACM, 2012. ISBN 978-1-4503-1577-7.

[91] Kevin Tew, James Swaine, Matthew Flatt, Robert Bruce Findler, and Peter Dinda. Places: Adding Message-passing Parallelism to Racket. In *Proceedings of the 7th Symposium on Dynamic Languages*, DLS '11, pages 85–96. ACM, 2011. ISBN 978-1-4503-0939-4.

[92] David Ungar, David Grove, and Hubertus Franke. Dynamic Atomicity: Optimizing Swift Memory Management. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2017, pages 15–26. ACM, 2017. ISBN 978-1-4503-5526-1.

[93] V8 authors. V8 Design Elements - Hidden Classes, 2012. URL `https://github.com/v8/v8/wiki/DesignElements`.

[94] Nalini Vasudevan, Kedar S. Namjoshi, and Stephen A. Edwards. Simple and Fast Biased Locks. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 65–74. ACM, 2010.

[95] Mario Wolczko. Benchmarking Java with Richards and Deltablue, 2013. URL `http://www.wolczko.com/java_benchmarking.html`.

[96] Jonathan Worthington. Racing to writeness to wrongness leads, 2014. URL `https://6guts.wordpress.com/2014/04/17/racing-to-writeness-to-wrongness-leads/`.

[97] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward!'13, pages 187–204. ACM, 2013.

[98] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 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, pages 662–676. ACM, 2017. ISBN 978-1-4503-4988-8.

[99] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. 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, pages 133–144. ACM, 2014. ISBN 978-1-4503-2926-2.

[100] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-Optimizing AST Interpreters. In *Proceedings of the 8th Dynamic Languages Symposium*, DLS '12, pages 73–82, October 2012. ISBN 978-1-4503-1564-7.

[101] Guoqing Xu. CoCo: Sound and Adaptive Replacement of Java Collections. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 1–26. Springer-Verlag, 2013. ISBN 978-3-642-39037-1. URL `http://dx.doi.org/10.1007/978-3-642-39038-8_1`.

[102] Minjia Zhang, Jipeng Huang, Man Cao, and Michael D. Bond. Low-overhead Software Transactional Memory with Progress Guarantees and Strong Semantics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 97–108. ACM, 2015.

Statutory Declaration

I hereby declare that the thesis submitted is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.

This printed thesis is identical with the electronic version submitted.

Benoit Daloze