Efficient and Thread-Safe Objects for Dynamically-Typed Languages

Benoit Daloze
Johannes Kepler University Linz, Austria
benoit.daloze@jku.at

Stefan Marr
Johannes Kepler University Linz, Austria
stefan.marr@jku.at

Daniele Bonetta
Oracle Labs, Austria
daniele.bonetta@oracle.com

Hanspeter Mössenböck
Johannes Kepler University Linz, Austria
moessenboeck@ssw.jku.at

Abstract

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 multithreaded environments.

This paper defines safety requirements for dynamic object models in multithreaded 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 shared between threads.

Basing our work on JRuby+Truffle, 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 field updates require synchronization. Thus, it can be a foundation for safe and efficient multithreaded VMs for a wide range of dynamic languages.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and objects, Dynamic storage management; D.3.4 [*Programming Languages*]: Processors—Run-time environments, Optimization

Keywords Dynamically-sized objects, Dynamic languages, Concurrency, Virtual Machine, Java, Truffle, Graal, Ruby

OOPSLA'16, November 2–4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4444-9/16/11...
http://dx.doi.org/10.1145/2983990.2984001

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 [22], V8 [11], Nashorn [24]), for Python (PyPy [3], Jython [14]), as well as for Ruby (JRuby [25], Rubinius [28]). 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 [5] or similar approaches [35]. By using maps, 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 [12].

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 multithreaded 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 paper, 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 3). These guarantees are of high practical value, because such problems can happen inside existing language runtimes despite application programs being seemingly data race free. 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, this paper contributes:

- a safety definition for adaptable object representations for multithreaded 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 on objects that are accessible 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 JRuby+Truffle, a state-of-the-art language runtime using the Truffle object storage model [35].

2. Background

This section discusses SELF's maps [5] and Truffle's object storage model [35, 36] to provide sufficient background on the object representation approaches used in dynamic language implementations.

2.1 SELF Maps

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.

The SELF programming language was the first to solve these issues by introducing *maps* [5]. 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 a complex and 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 [5] and has inspired many language implementations (e.g., PyPy [2], and Google's V8 [31]).

2.2 The Truffle Object Storage Model

Truffle [36] 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-optimized machine code by the Graal compiler [34], which uses partial evaluation. Language implementations using Truffle include JavaScript, R, Smalltalk, and Ruby [26].

The Truffle object storage model [35] 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 represents an arbitrary number of fields by combining a small number of fixed field locations with extension arrays for additional fields.

Figure 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

```
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 1. Sketch of Truffle's object storage model and a method reading a field.

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. 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.

3. 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.

3.1 State of the Art

Both SELF's *maps* and the Truffle object storage model 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 [14] and JRuby [25]. Since, for instance, JavaScript does not provide shared memory concurrency, V8's hidden classes [31] as well as Nashorn's `PropertyMap` [16] and SpiderMonkey's object model [23] do not provide safety guarantees for shared-memory multithreading. While Nashorn is not designed for multithreaded 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 [16]. 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.

3.2 Lost Field Definitions

The first safety problem is that concurrent field definitions for an object can lead to only one of the fields being added. This is illustrated in Figure 2.

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. 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 data race free, 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.

3.3 Lost Field Updates

Another race condition arises when the storage allocated to a given object needs to grow to accommodate new fields. Gen-

```
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)
}
```

Figure 2. The definition of field *b* can be lost when there are concurrent field definitions. The comments indicate a problematic interleaving of implementation-level operations performed by the object model.

```
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"
}
```

Figure 3. The update to field *a* can be lost if growing the storage is done concurrently with the field update. The comments indicate a problematic interleaving of implementation-level operations performed by the object model.

erally, 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). 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 Figure 3.

```
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)
}
```

Figure 4. 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.

3.4 Out-Of-Thin-Air Values

In some unsafe object representations, it is possible to observe out-of-thin-air values [17], 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 another field 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 Figure 4. 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. 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 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 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 on single-threaded execution and 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. [15, sec. 6.5] show that reads are $28\times$ more frequent than writes on shared objects in concurrent DaCapo benchmarks. From a performance perspective, multithreaded 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 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.

4.1 Read-side Synchronization

As described in Section 3.4, reading fields from an object that is concurrently updated from another 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 values of some other field, i.e., read out-of-thin-air values. For a shape transition, the object shape and one extension array would need to be updated atomically (cf. Section 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 some update might still be under way in another 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.

4.2 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 6.2.

4.3 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).

For such shape transitions, the object storage needs to be be updated, and one of the extension arrays potentially needs to be replaced, too. 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 details how we distinguish between thread-local and shared objects.

5. Local and Shared Objects

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

5.1 Distinguishing Local and Shared Objects

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 *local* and *shared* objects different shapes, 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 semantics to perform them 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 and 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 from all threads. In Java, the initial set would also include objects stored in static fields, and 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 7.4. The conceptual distinction between local and shared objects based on reachability was first introduced by Domani et al. and is detailed in Section 8.1.

Note that tracking sharing based on reachability overapproximates 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, detailed 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, multithreaded 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.

5.2 Write Barrier

To track all shared objects, the 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 from other threads. Not only the assigned object needs to be marked as shared, but all objects that are reachable from it as well 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 illustrated in Figure 5.

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.

The write barrier decides what to do based on the type of the assigned value as well as its shape if the value is an object. If the value has a primitive type, then it does not have fields and 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. For optimal performance, the write barrier specializes itself optimistically on the type and shape of the value for a given assignment. The type and shape of a value are expected to be stable at a certain assignment in the source code, as the value is assigned to a specific field, and application code reading from that field typically has specific expectations on its type. 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.

5.3 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 Figure 5. Such a traversal has a considerable run-time overhead. To minimize this overhead,

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

Figure 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.

we use the information provided in shapes to optimize the traversals of the graph. That is, we look at the fields of the assigned object and specialize on the shapes of the objects contained in those fields. We apply this optimization transitively, with a depth limit to avoid traversing large object graphs. Larger graphs are shared without this optimization.

This optimization is done by building a caching 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 caching structure is in this case a tree of 3 Share nodes, one for the rectangle and two for the two points, illustrated in the middle of Figure 6.

The caching structure is part of the program execution. The corresponding nodes are in fact Truffle AST nodes that implement the sharing. 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.

The Truffle AST can then be compiled, exposing the structure of the object graph to the compiler. Figure 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 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* [12, 19], 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.

6. Discussion

The design choices of Sections 4 and 5 have tradeoffs, which are discussed in this section.

6.1 Sharing Large Object Graphs

As discussed in Section 5.3, 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.

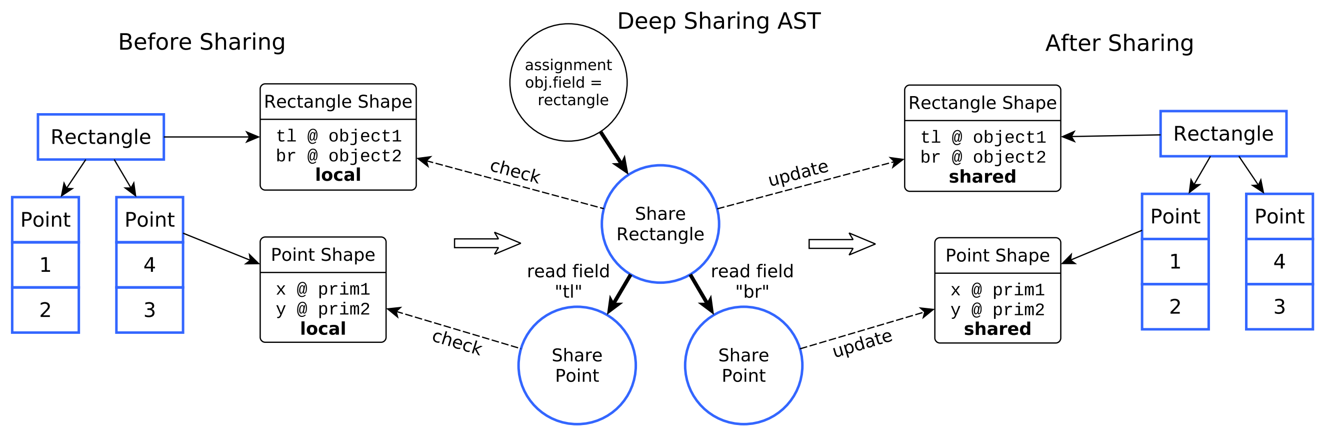

Figure 6. Deep Sharing of a Rectangle and two Points. The AST mirrors the structure of the Rectangle and its Points. The Share nodes check if an object matches the expected shape and update it to a shared variant.

```
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 */ }
}
```

Figure 7. Specialized sharing function for a Rectangle and its two Points. This code illustrates what the Deep Sharing nodes in Figure 6 perform when they are compiled down to machine code.

This technique is similar to what is done by Domani et al. [9] and Clifford et al. [6]. Domani et al. use such a technique to allocate objects directly in the global heap as described in further detail in Section 8.1. Clifford et al. use *mementos* to gather allocation site feedback and 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.

6.2 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 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 [7]. 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.

6.3 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 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.

6.4 Language-Independent Object Model

The Truffle object model is a language-independent runtime component and is currently used by Truffle-based language runtimes, including JRuby+Truffle. 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 [21].

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.

6.5 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 from 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.

6.6 Lazy Sharing of the Initial Set of Shared Objects

As an optional optimization, tracking of shared objects only needs to be done in truly multithreaded 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 the 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.

6.7 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 [7], could be used when class layouts need to be changed. This synchronization would coordinate all threads to perform the change for all objects as well as updating the class layout to include the new field. This approach has however 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.

7. Evaluation

We evaluate the proposed safe object model based on JRuby+Truffle [30], 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 JRuby+Truffle 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.

7.1 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[1] and revision `0bd7fa2d` of JRuby+Truffle[2].

[1] `https://github.com/graalvm/graal-core/commit/2075f904`

[2] `https://github.com/jruby/jruby/commit/0bd7fa2d`

Since Truffle and Graal are designed for implementing languages for server applications, this evaluation focuses on peak performance to assess the impact of the 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.

7.2 Baseline Performance of JRuby+Truffle

Since this work is based on JRuby+Truffle, we first demonstrate that its performance is competitive with custom-built dynamic language VMs. 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 [20]. This includes objects, closures, arrays, method dispatch, and basic operations. The benchmark set includes classic VM benchmarks such as DeltaBlue and Richards [33], 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 8 depicts the results normalized to Java 1.8.0_66 as a box plot. The box plot is overlaid with 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 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 JRuby+Truffle 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 JRuby+Truffle can compete with custom dynamic language VMs such as V8.

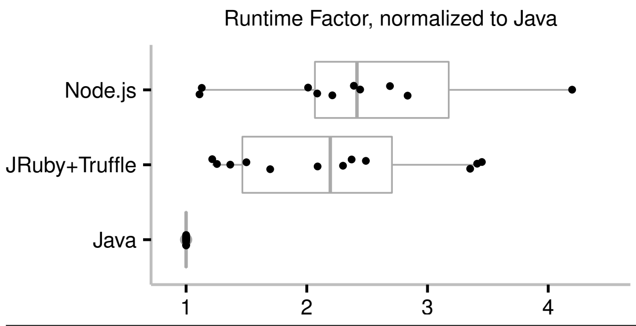

Figure 8. Comparing the performance of Java 1.8.0_66, JRuby+Truffle, and Node.js 5.4 based on twelve benchmarks that use a set of common language features between all three languages. Lower is better.

7.3 Impact on Sequential Performance

To verify that our design for 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* JRuby+Truffle, the version with our *safe* and optimized object model, and a configuration where all objects are *shared*, which means all object writes are synchronized. This *all shared* configuration approximates the worst-case impact of the object model, but does not represent common application behavior. We added this configuration also to estimate the overhead of state-of-the-art synchronization strategies as used by the JRuby object model.[3]

As illustrated in Figure 9, there is no significant difference between the *unsafe* and *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.

7.4 Initial Set of Shared Objects

In order to understand the potential of distinguishing between local and shared objects, we analyze the initial set of shared objects to determine which objects are globally reachable when starting an application. This set corresponds to the initial set for all sequential benchmarks in the *safe* configuration. The sequential benchmarks add a few classes and constants when loading but only in the order of a dozen objects.

Figure 10 lists the number of objects per class in this initial set. In total, 2,347 objects are globally reachable from the

[3] Note that we distinguish consistently between JRuby and JRuby+Truffle, since only the latter is using the optimized Truffle object model.

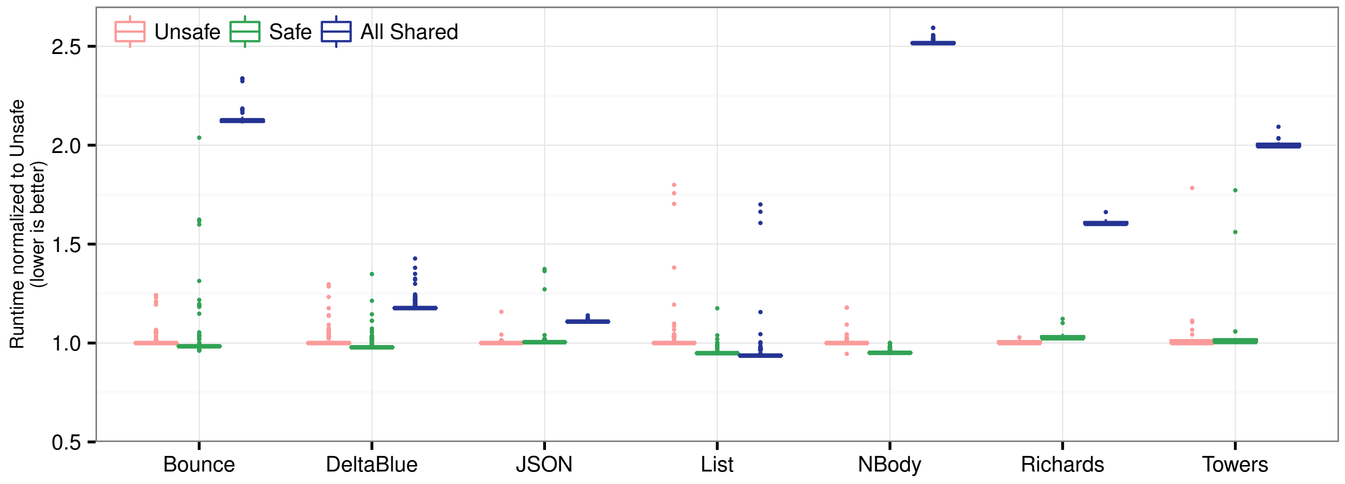

Figure 9. Impact on sequential performance, comparing the *unsafe* and *safe* object model. *All Shared* does not distinguish local and shared objects, and approximates state-of-the-art object models synchronizing on all object writes. Lower is better.

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 ($i = \sqrt{-1}$)

Figure 10. The initial set of 2,347 shared objects when starting an application under JRuby+Truffle with the safe object model.

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

Figure 11. Micro-benchmark for the worst-case field write overhead. Increasing integer values are assigned to a shared object field.

top-level constants and the global variables. These objects are created by the JRuby+Truffle 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.

7.5 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. Figure 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 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.

```
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
```

Figure 12. Benchmark for the deep sharing optimization. Rectangles are created and assigned to a shared object field.

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 10.

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 7.3. For multithreaded applications, performance is more similar to the parallel benchmarks discussed in Section 7.7.

7.6 Impact of Deep Sharing

To assess the impact of the deep sharing optimization, we created a micro-benchmark similar to the example in Figure 6. The benchmark, illustrated in Figure 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 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. Figure 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 is essentially free. The compiler sees the structure of the checks at compilation time and can fold all shape checks, field reads, and shape changes. It allocates the objects only right before the assignment to the shared object field, which makes it possible to construct the objects directly with the right shapes.

7.7 Parallel Actor Benchmarks

To assess the overhead of the safe object model on parallel code, we ported three of the parallel Savina actor benchmarks [13] from Scala to Ruby. The benchmarks focus on parallel computations that are coordinated via messages. They do not trigger the safety issues discussed in Section 3 by

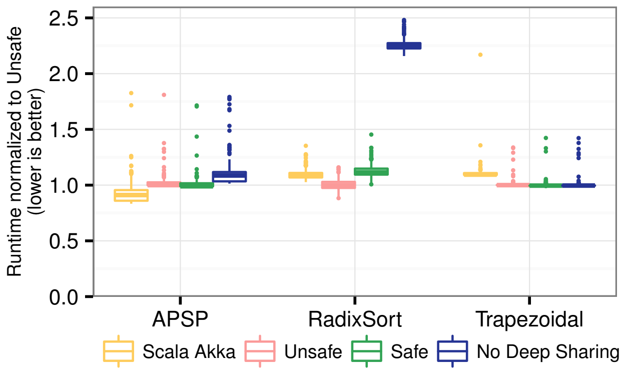

Figure 13. Impact on Parallel Actor Benchmarks, comparing the Scala implementation using Akka, the *unsafe* object model and the *safe* object model. *No Deep Sharing* disables the Deep Sharing optimization. Lower is better.

avoiding concurrent access to the same object. 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 [10]. *RadixSort* sorts integers using the radix sort algorithm.

We use a simple Ruby actor library that employs one thread per actor, each actor having its own mailbox. Additionally, we scale the benchmarks so that each thread has its own CPU core to avoid contention. Figure 13 shows how our library compares to the widely used and highly optimized Scala Akka actor framework. On APSP, Akka is 9% faster than JRuby+Truffle with the unsafe object model. On Radix-Sort it is 9% slower and on Trapezoidal it is 10% slower than the unsafe object model.[4] From these results we 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.

Nevertheless, Figure 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.

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

To characterize the behavior of the benchmarks in more detail, Figure 15 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 an embarrassingly parallel workload and requires 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 13 and the compiled code show 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 7.6).

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.

7.8 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 JRuby+Truffle, we evaluate the warmup behavior of the different object models on all the considered benchmarks.

Figure 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 JRuby+Truffle. Overall, the warmup of the safe object model is very similar to the unsafe one 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 7.7). The performance noise in RadixSort 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.

7.9 Memory Efficiency

Memory efficiency is a crucial aspect of object representations. To assess the impact of our approach on memory, 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 JRuby+Truffle 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 Figure 15. All benchmarks except *Startup* and *Sidekiq* only count user-defined objects allocated per iteration in order to depend less on language-specific representations of builtin types. In those benchmarks, builtin 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 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

[5] computed using the `sloccount` utility for the `lib` folders of Sidekiq and its used dependencies.

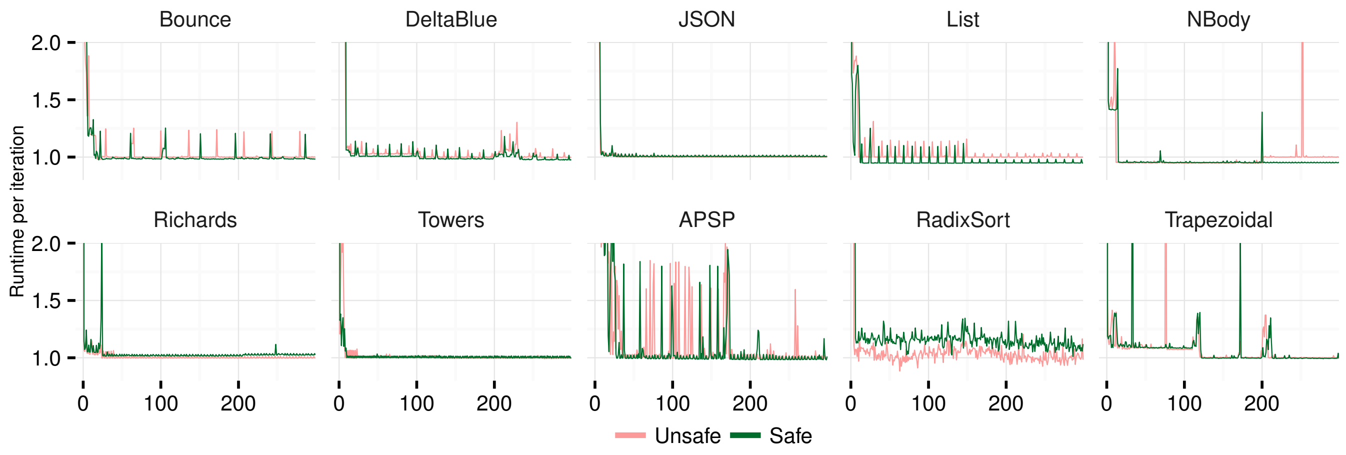

Figure 14. Runtime per iteration for the first 300 iterations, normalized to the median runtime for *unsafe*, illustratring the warmup of the *unsafe* and *safe* object models. Lower is better.

Benchmark	Sharing	Shapes	Objects	Frequent Objects	Extra Loc.
Startup	35.5%	45/61	2,354/6,640	1,919 String, 1,742 Symbol	0
Bounce	0%	0/2	0/151,500	150,000 Ball	0
DeltaBlue	0%	0/8	0/180,048	108,032 Vector, 36,003 Variable	0
Json	0%	0/7	0/189,900	67,550 JString, 43,350 Vector, 35,600 JNumber, . . .	0
List	0%	0/1	0/23,250	23,250 Element	0
NBody	0%	0/2	0/6	5 Body, 1 NBodySystem	0
Richards	0%	0/8	0/1,350	400 Packet, 300 TaskControlBlock, 300 TaskState	0
Towers	0%	0/1	0/8,400	8,400 TowersDisk	0
APSP	100%	4/4	1,456/1,456	724 Matrix, 724 ResultMessage	0
RadixSort	100%	6/6	400,014/400,014	400,004 ValueMessage	0
Trapezoidal	100%	4/4	14/14	5 WorkMessage, 4 WorkerActor, 4 ResultMessage	0
Sidekiq	2.4%	125/208	25,198/1,050,537	575,734 String, 150,602 Proc, 123,469 Array	3

Figure 15. User-defined objects allocated per iteration for each benchmark. Startup and Sidekiq count all objects including builtin 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.

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 4.2).

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 6.2, 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.

8. Related Work

Most related work is on language runtimes and synchronization techniques. This section provides an overview and discusses how these techniques relate to ours.

8.1 Memory Management and Garbage Collection

For garbage collectors, Domani et al. [9] 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 ours in Section 5.2. Their goal is to allow thread-local synchronization-free garbage collection. One of their optimizations is to allocate objects directly in the global heap, described in Section 6.1. They find this optimization crucial in their approach, as it reduces by multiple factors the sweeping time. Indeed, global objects allocated in thread-local heaps partition the free spaces, and the number of free spaces proportionally increases the sweeping time. 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.

8.2 Tracking Reachability and Precise Sharing

As part of an effort to define concurrency metrics for the DaCapo benchmarks, Kalibera et al. [15] 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 lessens 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.

8.3 Object Representations

As discussed in Section 3.1, 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 [14]. Thus, all object accesses are synchronized and safe with respect to our definition. However, as in the sequential case for which SELF's maps [5] were designed, using a hash table cannot compete in terms of performance.

JRuby's object model [25] 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 [35], 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. Similar to our approach, a field read is done without synchronization. However, JRuby synchronizes field writes for all objects regardless of sharing, to avoid losing concurrent updates and definitions.

By distinguishing 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.

8.4 Minimizing Synchronization and Avoiding it at Run Time

Other related work has been done with the goal of minimizing the overhead of synchronization primitives. Biased locking [32] is arguably the most popular technique in this domain. Biased locking relies on the assumption that locks are rarely contended, because even though a specific object might be accessed in parallel, often objects are only used temporarily 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 [8]), and is often combined with JIT compilation [29]. Our technique shares the goal of avoiding unnecessary operations for thead-local objects. 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. Note that the synchronization used for shared object writes in our model uses biased locks in HotSpot.

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 the LarkTM STM [37], 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, 18] 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 are not *shared*.

9. Conclusion

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 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 [5]. 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 evaluate our approach based on JRuby+Truffle, 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 multithreaded environments. By being language-independent, the model applies to a wide range of dynamic languages. Therefore, the techniques presented in this paper enable objects of dynamic languages to be used for shared-memory parallel computations while remaining safe and efficient.

10. Future Work

The results of this paper open up a number of new avenues for future research.

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 4.2). 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.

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`, which scales better for write-contended workloads.

With the safe object model presented in this paper, the next step is to widen the scope of problems considered. Built-in collections such as arrays, maps, and sets of many dynamic programming languages have similar safety issues (cf. Section 3) leading to, e.g., out-of-bounds exceptions or lost updates when they are dynamically resized. The problem is further exacerbated by sequential approaches to optimize their representation such as collection storage strategies [4, 27]. While these strategies improve the sequential performance significantly, it still needs to be investigated how they can be made safe for multithreaded environments.

Assuming a language implementation with safe objects and safe built-in collections, it would further be interesting to determine which useful guarantees are still missing compared to classic implementations using a *global interpreter lock*. We hope that such improvements could bring dynamic languages closer to a point where they can provide simple and safe parallel programming models to their users.

Acknowledgments

We gratefully acknowledge the support of the Virtual Machine Research Group at Oracle Labs, the Institute for System Software at JKU Linz and everyone else who has contributed to Graal and Truffle. We would like to thank Kevin Menard, Chris Seaton and Andreas Wöß for their careful reviews. Stefan Marr was funded by a grant of the Austrian Science Fund (FWF), project number I2491-N31. Oracle, Java, and HotSpot are trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

References

[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. 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. doi: 10.1145/1133981.1133985.

[2] C. F. Bolz and L. Tratt. The Impact of Meta-Tracing on VM Design and Implementation. *Science of Computer Programming*, pages 408–421, Feb. 2015. doi: 10.1016/j.scico.2013.02.001.

[3] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. 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. doi: 10.1145/1565824.1565827.

[4] C. F. Bolz, L. Diekmann, and L. 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. doi: 10.1145/2509136.2509531.

[5] C. Chambers, D. Ungar, and E. 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, pages 49–70. ACM, October 1989. doi: 10.1145/74878.74884.

[6] D. Clifford, H. Payer, M. Stanton, and B. 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. doi: 10.1145/2754169.2754181.

[7] B. Daloze, C. Seaton, D. Bonetta, and H. 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, 2015.

[8] D. 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.

[9] T. Domani, G. Goldshtein, E. K. Kolodner, E. Lewis, E. Petrank, and D. 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. doi: 10.1145/512429.512439.

[10] R. W. Floyd. Algorithm 97: Shortest Path. *Commun. ACM*, 5 (6):345, Jun. 1962. doi: 10.1145/367766.368168.

[11] Google. V8 – Google's JavaScript engine, 2015. `https://developers.google.com/v8/`.

[12] U. Hölzle, C. Chambers, and D. 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. doi: 10.1007/BFb0057013.

[13] S. M. Imam and V. 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. doi: 10.1145/2687357.2687368.

[14] J. Juneau, J. Baker, F. Wierzbicki, L. Soto, and V. Ng. *The Definitive Guide to Jython: Python for the Java platform*. Apress, 2010. doi: 10.1007/978-1-4302-2528-7.

[15] T. Kalibera, M. Mole, R. Jones, and J. 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. doi: 10.1145/2384616.2384641.

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

[17] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *Proceedings of the 32nd Symposium on Principles of Programming Languages*, POPL '05, pages 378–391. ACM, 2005. doi: 10.1145/1040305.1040336.

[18] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the Overhead of Nonblocking Software Transactional Memory. Technical report, Department of Computer Science, University of Rochester, 2006.

[19] S. Marr, C. Seaton, and S. 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. doi: 10.1145/2737924.2737963.

[20] S. Marr, B. Daloze, and H. Mössenböck. Cross-Language Compiler Benchmarking—Are We Fast Yet? In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS 2016. ACM, 2016. doi: 10.1145/2989225.2989232.

[21] F. Morandat, B. Hill, L. Osvald, and J. 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. doi: 10.1007/978-3-642-31057-7_6.

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

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

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

[25] C. Nutter, T. Enebo, O. Bini, N. Sieger, et al. JRuby, 2016. `http://jruby.org/`.

[26] Oracle Labs. GraalVM: New JIT Compiler and Polyglot Runtime for the JVM, 2016. `http://www.oracle.com/technetwork/oracle-labs/program-languages/overview/index.html`.

[27] T. Pape, T. Felgentreff, R. Hirschfeld, A. Gulenko, and C. F. Bolz. Language-independent Storage Strategies for Tracing-JIT-based Virtual Machines. In *Proceedings of the 11th Symposium on Dynamic Languages*, DLS 2015, pages 104–113. ACM, 2015. doi: 10.1145/2816707.2816716.

[28] E. Phoenix, B. Shirai, R. Davis, D. Bussink, et al. Rubinius, 2016. `http://rubini.us/`.

[29] K. Russell and D. 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. doi: 10.1145/1167473.1167496.

[30] C. Seaton, B. Daloze, K. Menard, P. Chalupa, T. Würthinger, et al. JRuby+Truffle – a High-Performance Truffle Backend

for JRuby, 2016. `https://github.com/jruby/jruby/wiki/Truffle`.

[31] V8 authors. V8 Design Elements - Hidden Classes, 2012. `https://developers.google.com/v8/design#prop_access`.

[32] N. Vasudevan, K. S. Namjoshi, and S. 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. doi: 10.1145/1854273.1854287.

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

[34] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. 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. doi: 10.1145/2509578.2509581.

[35] A. Wöß, C. Wirth, D. Bonetta, C. Seaton, C. Humer, and H. 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. doi: 10.1145/2647508.2647517.

[36] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-Optimizing AST Interpreters. In *Proceedings of the 8th Dynamic Languages Symposium*, DLS'12, pages 73–82, October 2012. doi: 10.1145/2384577.2384587.

[37] M. Zhang, J. Huang, M. Cao, and M. 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. doi: 10.1145/2688500.2688510.