An Object Storage Model for the Truffle Language Implementation Framework

Andreas Wöß* Christian Wirth† Daniele Bonetta† Chris Seaton† Christian Humer*
Hanspeter Mössenböck*

*Institute for System Software, Johannes Kepler University Linz, Austria †Oracle Labs
{woess, christian.humer, moessenboeck}@ssw.jku.at {christian.wirth, daniele.bonetta, chris.seaton}@oracle.com

Abstract

Truffle is a Java-based framework for developing high-performance language runtimes. Language implementers aiming at developing new runtimes have to design all the runtime mechanisms for managing dynamically typed objects from scratch. This not only leads to potential code duplication, but also impacts the actual time needed to develop a fully-fledged runtime.

In this paper we address this issue by introducing a common object storage model (OSM) for Truffle that can be used by language implementers to develop new runtimes. The OSM is generic, language-agnostic, and portable, as it can be used to implement a great variety of dynamic languages. It is extensible, featuring built-in support for custom extension mechanisms. It is also high-performance, as it is designed to benefit from the optimizing compiler in the Truffle framework. Our initial evaluation indicates that the Truffle OSM can be used to implement high-performance language runtimes, with no performance overhead when compared to language-specific solutions.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Run-time environments, Optimization

General Terms Algorithms, Languages, Performance

Keywords Dynamic languages, virtual machine, language implementation, optimization, Java, JavaScript, Ruby, Truffle

1. Introduction

Truffle [25] is an open-source framework for the implementation of high-performance language runtimes using Java and the Java virtual machine (JVM). Using Truffle, a language runtime can be developed just by implementing an AST interpreter. More precisely, the AST interpreter *is* the language runtime. The AST (and the interpreter which is represented by the *execute* methods of the AST nodes) can be automatically optimized by Truffle and can finally be compiled into very efficient machine code. Thanks to this feature, Truffle can be used conveniently for developing the runtime support for dynamically typed languages. As of today, several Truffle-based implementations for dynamic languages exist, including JavaScript, Ruby, Python, Smalltalk, and R. All of the existing implementations offer very competitive performance when compared to other state-of-the-art implementations, and have the notable characteristics of being developed in pure Java (in contrast to native runtimes that are usually written in C/C++).

To further sustain and widen the adoption of Truffle as a common Java-based platform for language implementation, Truffle offers a number of shared APIs that language implementers can use to optimize the AST interpreter in order to produce even more optimized machine code. In order to obtain high performance, however, there has still been one core component that the Truffle platform did not offer to language implementers, and that had to be implemented manually. This core component is the object storage model, that is, the runtime support for implementing dynamic objects. Indeed, language implementers relying on the Truffle platform have to implement their own language-specific model for representing objects, and then have to optimize the language runtime accordingly in order to optimize the AST interpreter for the characteristics of a certain language's object model. Requiring language implementers to develop the object storage model of their new language *from scratch* is not only a waste of resources, but could also lead to questionable software engineering practices such as code duplication and non-modular design.

With the goal of solving the above limitation of the Truffle framework and with the aim of supporting language developers with a richer shared infrastructure, this paper introduces a new, language-independent, object storage model (OSM) for Truffle. The new object storage model represents a notable progress in the Truffle framework—as well as in the domain of similar frameworks—as it provides language implementers with a common shared component that can be used to obtain support for new language runtimes having the following properties:

- *Generality*. The Truffle OSM can be used by language implementers as the basis for developing the object model of their new or existing guest languages. Its design allows common mechanisms (such as dynamic object resizing and dynamic type dispatch) to be reused across language runtimes, providing developers with a set of built-in optimizations. Thanks to this property, the Truffle-based language runtimes of Ruby and JavaScript have been implemented sharing the same type specialization and inline caching mechanisms.
- *Extensibility*. The Truffle OSM is extensible. Languages requiring custom operations on object instances (e.g., on proxies [23]) can extend the OSM by simply adding the new feature to the specific language runtime. The extension mechanism

PPPJ '14, September 23–26 2014, Cracow, Poland.

http://dx.doi.org/10.1145/2647508.2647517

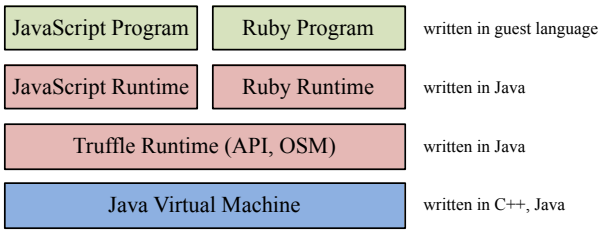


Figure 1: System structure of guest language implementations on top of the Truffle platform

is designed to take advantage of Truffle-based self-optimizing AST interpreters.

- *High performance*. Not only can the Truffle OSM be shared across multiple language runtimes, but it also offers them the same competitive performance characteristics.

The Truffle OSM is designed to be a core runtime component for any new language runtime based on the Truffle platform, offering automatic built-in support for runtime mechanisms such as type specialization and polymorphic inline caches.

This paper is structured as follows. In the next section we give background information on the Truffle framework. Section 3 introduces the design and the main characteristics of the Truffle OSM. Section 4 presents the implementation details, while Section 5 presents a performance evaluation. Section 6 presents related work, while Sections 7 and 8 conclude this paper.

2. Background

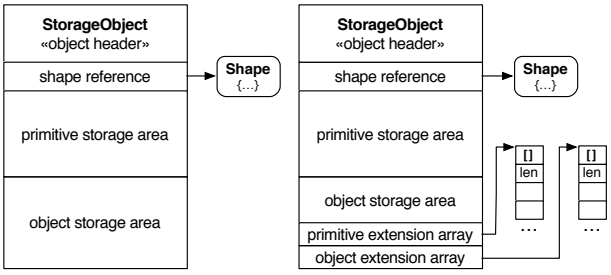
An *object model* defines the properties of objects in a specific programming language as well as the semantics of operations on them. Usually, definitions of a language's object model (e.g., the Java object model [10]) include a formalization of all the operations that can be performed on object instances along with the formal semantics for such operations with respect to other language aspects (e.g., the Java memory model [15]).

We use the term *object storage model* (OSM) to describe an abstract language-agnostic object model on top of which concrete language-specific object models can be built. An OSM defines how objects are represented in the *host language* (i.e., the language in which the OSM is implemented, e.g., Java) and provides basic concepts for emulating object models of *guest languages* (i.e., languages that are implemented on top of the OSM, e.g., Ruby). Note that we are concerned only about storage within memory.

Moreover, the OSM features built-in support for common optimization techniques that can be used to build efficient language implementations. A naive, but often inefficient, approach would be to represent a dynamically-typed object as a hash table.

2.1 The Truffle Framework Overview

Truffle [25] is a language implementation framework entirely written in Java. A language runtime implemented in Truffle (called the guest language runtime) is expressed as an abstract syntax tree (AST) interpreter implemented in Java (which is called the host language). Figure 1 shows an overview of the separate layers in the Truffle platform. As with standard AST interpreters, the AST is evaluated by recursively executing its nodes in post-order traversal. In contrast to standard AST interpreters, however, Truffle ASTs can *self-optimize*: based on profiling feedback gathered at run time, AST nodes can speculatively replace themselves with *specialized* variants that can eventually be compiled into highly optimized machine code [24]. For instance, a node can speculate on the type of arguments or on the size of data structures, ignoring unlikely and potentially costly corner cases (in terms of generated machine code). If the speculation turns out to be wrong later on, nodes can again rewrite themselves to more generic versions, providing all the necessary functionality for the cases observed.



(a) without extension arrays (b) with extension arrays

Figure 2: In-memory layout of a storage object

Truffle AST interpreters are standard Java applications, and can therefore be executed by any Java Virtual Machine (e.g., by the Oracle HotSpot VM). When executed on the Graal VM [18], however, the Truffle AST can be further optimized and compiled in order to produce even more efficient machine code. More precisely, the Graal just-in-time (JIT) compiler [5, 21] has special knowledge of the Truffle AST structure, and can optimize the execution speed of the guest language program by means of *partial evaluation* of the AST [25]. Partial evaluation means that when an AST reaches a stable state, the interpreter dispatch can be precomputed and the residual code transformed into a single compilation unit which Graal can compile into highly optimized machine code. An AST is considered *stable* when it did not change for a predefined amount of executions. In order to guarantee that all AST nodes can be compiled, every node must stabilize eventually. Therefore, the number of possible rewrites per node must be finite. When an assumption is invalidated during execution of the machine code, the machine code is *deoptimized* [13], i.e., the code is discarded and execution is continued in the interpreter. This allows the AST to re-specialize to new run-time feedback and later to be compiled to new machine code again.

3. The Truffle Object Storage Model

The Truffle OSM maps guest-language object instances to storage objects that store its data in the form of properties.

For example, consider the following JavaScript code fragment:

```
object = { day: 23, month: "September" };
```

The variable *object* is assigned a newly created Truffle object with two properties, *day* and *month*, that are assigned integer and string values, respectively.

3.1 Object Layout

Objects in our model consist of a variable number of properties, each with a name, a value, and a set of attributes. Furthermore, objects are self-describing and have a fixed number of operations that define their behavior. In the class-based object model of Java, object instances would only contain values and references as well as a pointer to a shared class descriptor that describes the format and the behavior of the object. In contrast, in a dynamic object model, metadata is mutable and associated with the object itself. In order not to waste space with duplicate metadata and to be able to optimize object accesses, we take an approach that roughly resembles the object model of SELF [2], but at the same time is implemented

on top of the Java object model with a fixed object size. An object consists of two separate parts: the *object storage*, containing per-instance data, and the *shape*, which provides a mapping of member names to locations in the object storage (similar to a Java class).

Every object created using this object storage model is an instance of a so-called *storage class*, a Java class that acts as a container for per-instance data. A storage class extends the common base class `StorageObject`. This class has a reference to a shape object that describes the current format and behavior of the object (note that the shape of an object can change at run time). Figure 2 shows how an object is structured in memory. Data is split into two separate storage areas, one for object references (managed by the garbage collector) and one for primitive values. Since the storage class has a fixed size, the object model can allocate any additional properties in an optional *extension array* which is resized as needed. There can be up to two extension arrays, one for objects and one for primitive values. The references to those arrays are stored in the object storage area.

A *shape* maps property names to storage locations and sets of attributes. The location specifies the exact memory location where the value of the property is stored, either relative to the storage object or absolute (like with static fields in Java). A shape fulfills the same task as a Java class of expressing the metadata of a storage object. However, in contrast to the immutable class pointer, an object's shape field can change over time. Shapes themselves and all their accessible data are immutable. Immutability allows us to compare shapes very efficiently using an identity comparison. Any change to the property collection results in a new shape. Finally, every shape has a mutable data structure called the *transition map* that is used to quickly find existing successor shapes that were derived from this one by property changes (see Section 3.5).

3.2 Operations

Any interaction with Truffle objects—rather than accessing data directly—goes through a set of *operations* defined by the language runtime. By default, OSM-managed objects offer the following operations for accessing properties. A guest language can add operations by combining existing ones or adding new operations. In the following we present the operations predefined in the OSM and give JavaScript code examples in which these are used.

- **get(object, key)**
  Gets the value of the property *key* or `null` if no such property exists. This operation is used when reading a named property of an object (e.g., `object.key`).
- **has(object, key)**
  Checks if the object has a property *key* (e.g., `'key' in object`).
- **set(object, key, value)**
  Sets the value of the property *key* or creates a new one with the name *key* and default attributes (e.g., `object.key = value`).
- **define(object, key, value, attributes)**
  Defines a new property of name *key* with the given value and attributes. If the property already exists, the operation fails.
- **delete(object, key)**
  Deletes a property if it exists (e.g., `delete object.key`).
- **getSize(object)**
  Gets the number of properties defined in the object.
- **getAttr(object, key)**
  Gets the attributes of the property *key*. If no such property exists the operation fails.
- **changeAttr(object, key, attributes)**
  Changes the attributes of the property *key*. If no such property exists the operation fails.

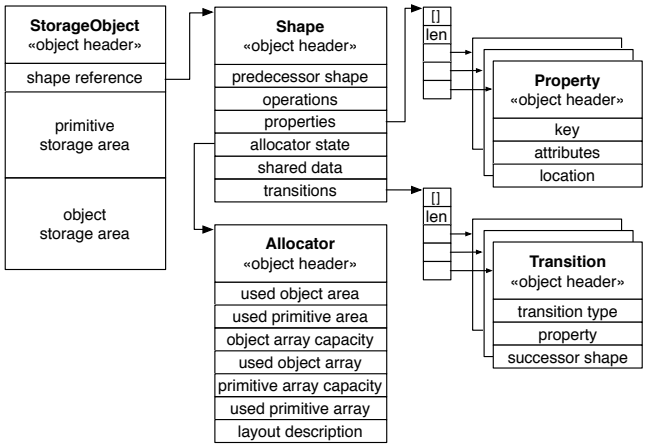


Figure 3: Components of the Truffle OSM

- **getKeys(object, filterCallback)**
  Gets a list of keys defined in the object, filtering on a given boolean predicate (e.g., `for (var key in object) {}`).
- **getValues(object, filterCallback)**
  Gets a list of values defined in the object, filtering on a given boolean predicate.

Language implementations can override the standard operations and can add custom operations. Overriding an operation also allows to change what happens if an operation fails; by default, an exception is thrown. Each operation is defined by a method that executes the operation without any argument specialization and by a corresponding Truffle node that specializes the operation based on the provided arguments so that repeated executions become faster. This is achieved using a polymorphic inline cache, as described in Section 4.4.

3.3 Components of the Truffle OSM

Figure 3 gives an overview of the main components of the Truffle OSM.

Storage Class: a Java class extending the class `StorageObject` (see Figure 6). Every storage class contains a number of storage fields automatically managed by the OSM as well as a field pointing to the object's metadata stored in the shape. The fields are used to store the data of the guest-language object.

Shape: a sharable description of the object's format and properties. The shape contains all the metadata of a Truffle OSM object.

Property: describes a property by its identifier, its location, and its attributes. Depending on what attributes are set, operations may behave differently. The OSM provides the predefined attribute *hidden*, which indicates that the property is injected by the language implementation and shall not be visible to guest-language programs. Others can be defined by the language implementation.

Allocator: The allocator is used to create storage locations for new members. It maintains information about the size of the extension arrays and which parts of the storage areas are in use.

Layout: every storage class is assigned a layout description upon its first use that provides information about its available fields, the allocation strategy, and enabled OSM features. The allocator uses this information to lay out storage locations in the available storage areas.

Location: defines the storage location of a property (see the next section). It can reference one or more in-object fields, one or more array indices in a storage extension array or can contain a constant value. Additionally, the location constrains the type of the storage location and whether it can only be set once (cf. final fields in Java).

Operations: a method table of operations applicable to the object. Language implementers can add or override these operations to contribute new features. Operations are described in Section 3.2.

Transition: when a property is added to an object or removed from it, the shape of the object changes, which is described by a transition. The result of a transition is a successor shape. Every shape has a *transition map* that links a shape to its successor shapes. Transitions are described in more detail in Section 3.5.

Shared Data: a special storage area in the shape that can be used by the language implementation to store additional metadata in the shape. This data is inherited by successor shapes and thus preserved across shape transitions. For instance, it can be used to share a class object between multiple shapes that belong to a common guest-language class.

3.4 Storage locations

We distinguish the following types of storage locations:

- **Object field location**: denotes an instance field of the storage object that is used to store an object reference. Additionally, the location holds a lower bound Java type of the referenced object, and whether it is guaranteed not to be `null`.
- **Primitive field location**: denotes one or more primitive instance fields of a Truffle object used to store a value of the primitive type attached to the location. If the value spans multiple fields, they must be consecutive in memory and properly aligned (see Section 4.3).
- **Object extension array location**: denotes an element of an `Object[]` array, loaded from an object field location. The type information is the same as with object field locations.
- **Primitive extension array location:** like a primitive field location but instead denotes a segment of an `int[]` array, loaded from an object field location. As with primitive field locations, the elements must be consecutive and properly aligned.
- **Static location**: an untyped location that itself contains a value. This type of locations can be used to store constant values directly in the shape.

Additionally, all locations can be equipped with Java *final* semantics as well as with *volatile* semantics.

3.5 Shape Transitions

The set of shapes is organized as a tree where nodes represent shapes and edges represent shape transitions. The transitions table is used to lookup successor shapes and to ensure that equally-structured objects are always assigned the same shape. The order in which properties are inserted is part of the structure. The root of the tree is the empty shape, i.e., the initial shape of an object without any properties. Using shape transitions, two objects with the same properties can eventually end up having the same shape.

Figure 4 shows some example JavaScript code that explains how shapes are dynamically created and assigned to objects. The corresponding shape tree is visualized in Figure 5. The empty shape, denoted by {}, always exists from the beginning. In the example, it is used to create a new, empty object. Then, the property `x` with an integer value is added to the object, triggering the creation of the shape {`x:int`}, i.e., a shape containing a property `x` with an integer location, and adding it as a successor shape to {} with a transition "x:int". Likewise, another property `y` is added, adding successor shape {`x:int, y:int`} to the graph. After that, another object is created, again with properties `x` and `y`, but this time with String values. Thus, due to different types, a new path is inserted with the shapes {`x:String`} and {`x:String, y:String`}. Finally, another such object is created, but with a String and an integer value, resulting in another shape {`x:String, y:int`}.

```
var a = {};
// a's shape is {}
a.x = 4;
// a's shape is {x:int}
a.y = 2;
// a's shape is {x:int, y: int}

var b = {x: "one", y: "two"};
// b's shape is {x:String, y:String}

var c = {x: "one", y: 2};
// c's shape is {x:String, y:int}
```

Figure 4: JavaScript example demonstrating the relation between objects and shapes

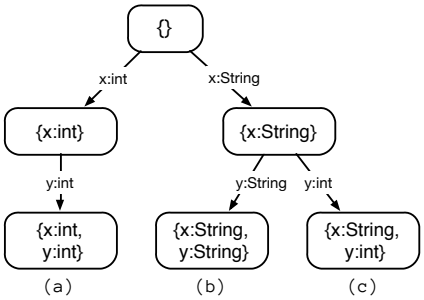


Figure 5: Shape tree for the JavaScript code in Figure 4

Possible transitions between shapes are:

- **Add a property:** change to a shape containing the new property; if necessary, grow an extension array.
- **Delete a property:** change to a shape without the removed property; if the deletion produces a gap in the storage, shift subsequent properties to fill the gap.
- **Change the attributes of a property:** change to a shape with an updated property description.
- **Change the storage location of a property:** change to a shape with a modified storage location of a property, thus changing the representation and the type of the property. This transition may also require moving existing properties to make space or to fill up empty space. For example, an `int` location may have to be widened to a `long` location or may be moved to an `Object` location. Subsequent storage locations need to be shifted to accommodate for the size change. Type transitions are only allowed to Java types that are higher up in the type hierarchy, i.e., when a shape has a property with the type `String`, it cannot make a type transition to `Boolean`, but it can make a transition to `Object`. This is to ensure type transitions are acyclic. It also allows migration of existing objects.

```
public class JSObject extends StorageObject {
  // private Shape shape; // inherited

  // implementation-defined number of fields
  // managed by object storage model
  @DynamicField private int pri1;
  @DynamicField private int pri2;
  @DynamicField private int pri3;
  @DynamicField private int pri4;
  @DynamicField private Object obj1;
  @DynamicField private Object obj2;
  @DynamicField private Object obj3;
  @DynamicField private Object obj4;

  public JSObject(Shape shape) {
    super(shape);
  }
}
```

Figure 6: Java class definition of an object storage class

- **Change the object's operations:** change to a shape with different operations. This can be used to change and instrument the behavior of an object. For example, it can be used to intercept accesses for debugging purposes, or to enable aspect-oriented programming features.

4. Implementation

This section explains how the Truffle OSM is implemented in Java.

4.1 Object Storage Class

As described in the previous section, every object of the guest language (e.g., JavaScript) is represented by a Truffle object which is an instance of an object storage class and serves as a container for the properties of the object obeying the semantics of the guest language.

All object storage classes are derived from the base class `StorageObject` that takes care of the object's relation to its shape. The language implementer defines the actual storage strategy of the guest language by extending this class. Figure 6 shows a storage class with four primitive fields and four object fields. Fields with the `@DynamicField` annotation are automatically used to store data. We distinguish two types of dynamic fields: object fields and primitive fields, declared as Java fields of type `Object` and `int`, respectively. The OSM does not allow other field types in order to ensure absolute freedom in the allocation of locations to fields.

The object fields define an object storage area with object references managed by the garbage collector, while the primitive fields form a primitive storage area. The latter can be used to store values of arbitrary primitive types, as described in Section 4.3. The size of these areas is fixed and cannot be changed after object creation. However, the object model can extend the two storage areas on-demand with `Object[]` and `int[]` arrays. Additional elements not fitting into the storage object's fields can be stored in those extension arrays. References of those arrays are stored as object fields. Until an extension array field is needed, its (object) field can be used as an ordinary storage field. The object model automatically takes care of moving conflicting entries out of the way. Overall, this ensures that an arbitrary number of elements can be stored in a storage object, providing fast access to fields and slightly slower access to elements stored in extension arrays.

For a language implementer, the most crucial decision is how many (primitive and object) fields should be provided. The more fields are provided, the more elements can be accessed in fast mode without further indirections via extension arrays. However, the larger the storage object is, the more heap space is consumed even if some guest-language objects do not use it. For instance, an object with no elements still consumes all the heap space necessary to allocate its storage object. The Truffle OSM does not make any assumption on the number of fields marked with `@DynamicField`. Therefore, language implementers can choose an arbitrary number. A good number of fields can be found by empirical measurements on a diverse set of benchmarks. How to define the ideal number of fields is out of the scope of this paper.

4.2 Shape

A shape object represents the layout of a Truffle object, i.e., it describes which guest-language elements are stored in which fields of the storage object. A storage object would be called an *Object* in the Java ecosystem, while a shape is equivalent to a *Class* in Java. Note, however, that this terminology is different in other languages. In JavaScript, for instance, there are no explicit classes, so a shape in the corresponding Truffle implementation just describes which properties an object currently has.

The shape infrastructure is automatically provided by Truffle without any need of the language implementer to contribute to it. Currently, a tree of nodes is created, where each node represents one property of the guest language. This concept, known as *hidden classes*, works well when the guest language program uses objects with a reasonably low number of fixed properties. A statically typed guest language (e.g., Java) always fulfills this requirement as it does not dynamically add properties to its objects during execution. Typical applications in dynamically typed languages also comply, as their dynamically added properties usually have fixed names. In rare cases, however, objects can have an unusually high number of different properties and thus require large amount of shapes. An example of this might be when a guest-language object is used as a hash table. In such cases, the OSM implementation can decide to store the shape information differently, e.g., as a Java `HashMap`.

4.3 Primitive Value Allocation Strategy

The Truffle OSM can handle objects of any size. This is achieved by storing primitive values into dynamic object fields (i.e., fields marked as `@DynamicField`), and by using the extension array when all the dynamic fields have been used[1].

If possible, primitive values are directly stored in the object without boxing or tagging, and can be accessed without any additional indirection. Boxing and tagging are avoided by storing primitives values in unboxed mode as long as their types stay monomorphic, i.e., by speculating on the primitive type of a value. When the extension array is used, boxing can be avoided similarly, and only one more indirection is needed.

All primitive fields and extension array elements are of a Java primitive type that does not necessarily match the types of the values stored in them. We combine consecutive slots to provide storage for larger types, e.g., `double` and `long` values. For values larger than 4 bytes, we enforce an 8-byte alignment, because misalignment of such values could lead to suboptimal performance depending on the processor architecture. When attempting to store an incompatible type in a primitive storage location, the primitive type speculation has failed, and the object has to be *reshaped*, that is, a new shape has to be assigned to it. From then on, the property is stored in boxed mode in a storage location of reference type.

[1] The extension array is stored in one of the dynamic fields like any other Java value

4.4 Polymorphic Inline Caches

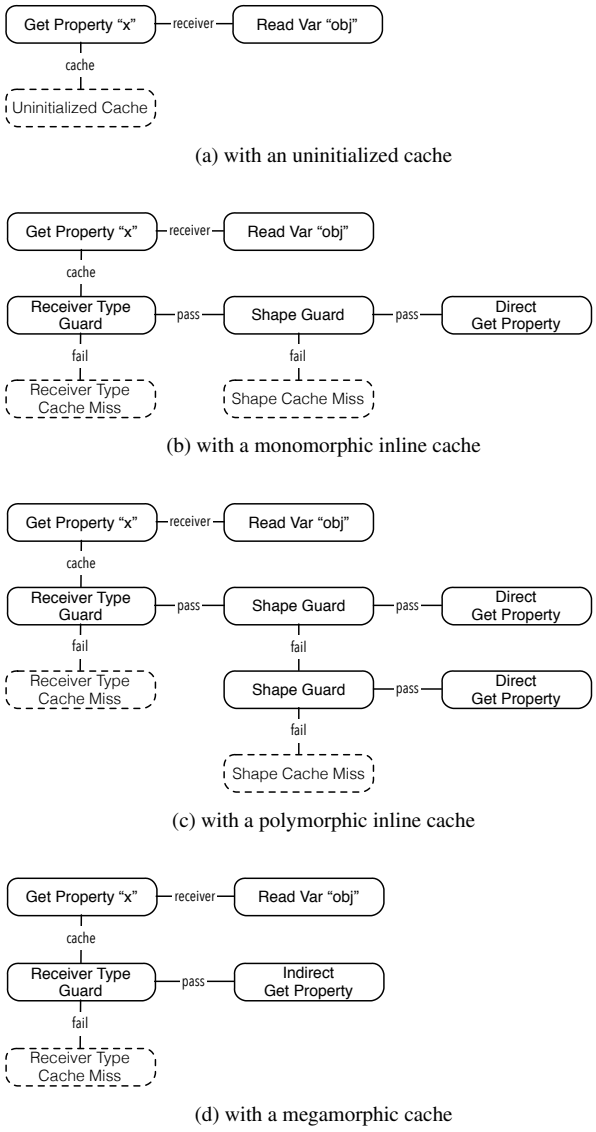
A polymorphic inline cache [12] is a common technique used to speedup the performance of dynamically typed languages. Thanks to inline caches, operations involving some form of dynamic binding (e.g., a dynamic property lookup) speculate on the operation being somehow stable (e.g., reading a dynamic property always having the same type), thus leading to a notable performance gain.

Inline caches in Truffle are implemented as nodes in the interpreter's AST. To form a polymorphic inline cache, cache entries are represented as a chain of nodes. Every cache node has one or more *guards* that check whether to proceed with the specialized operation (i.e., whether the speculation is valid) or to skip over to the next node in the chain. At the end of the node chain lies a special rewrite node that is never compiled. When reached from within compiled code, it immediately transfers control back to the interpreter and invalidates the code. This cache rewrite node adds a new entry to the cache, i.e., a new node to the chain.

For example, consider the following JavaScript expression: `obj.x`. The AST interpreter represents this as a GetProperty node with the constant argument `"x"` and two children: the receiver node which evaluates the left hand side expression, `obj`, and a cache node that performs the access to `x` using a polymorphic inline cache. The GetProperty node first evaluates the receiver and then invokes the cache node with the evaluated receiver object as an argument. Initially this cache is in the *uninitialized* state, as shown in Figure 7a. When executed, it queries the object's class and shape, and calls the *resolveGet* method pertaining to the *get* operation to create a specialized access node. This node is inserted to initialize the cache to serve as a *monomorphic* inline cache (Figure 7b). Two nested node chains check the Java class and the object's shape, and if both match, the property's location is directly accessed. If none of the current guarding checks in a chain succeeds, the cache miss handler inserts a new link into the chain (Figure 7c).

Inline caching has a very positive performance impact when the number of cache entries is small. Some access sites, however, may have a large number of different types/shapes. In this case, the polymorphic inline cache would grow excessively large, slowing down the dispatch. Furthermore, due to the fact that the AST encapsulating the inline caches is compiled as a whole, single, compilation unit, polymorphic access sites bloat the code and can lead to frequent code invalidations when the inline cache is unstable.

To address this issue a predefined limit on the number of cache entries is used. Whenever the chain exceeds this limit on a cache miss, the inline cache is considered *megamorphic* and is rewritten to an indirect dispatch node (Figure 7d). This node loads the respective operation (e.g., *get* or *set*) from the operation table in the object's shape and calls it.



(a) with an uninitialized cache

(b) with a monomorphic inline cache

(c) with a polymorphic inline cache

(d) with a megamorphic cache

Figure 7: AST interpreter tree of the JavaScript expression `obj.x`

4.5 Type Specialization

The Truffle OSM automatically specializes properties on their type. When a new property is added, it is assigned the most suitable type. For example, for a string value, an object location of type `String` is allocated. The implementation of this location is shown in Figure 8. It makes use of the compiler directives described in Section 4.9.1 to access the field and to cast the read value to its current type (which can change at run time due to type specialization) without a run-time check. On each set operation, we check whether the new value is still of the expected type. If not, we throw an exception that is handled by the set operation, in which case it generalizes the location's type to fit both the old and the new value.

For faster access, the set operation is handled by an inline cache node that performs a guarding check on the shape and accesses the location directly, thus omitting the costly lookup of the property. If the location's type check fails, we go from the compiled code back to the interpreter, generalize the type, and rewrite the inline cache node to the generalized location. Figure 9 shows the implementation of the cached set operation. If the shape does not match the expected shape of the cache entry, it probes the next cache entry, if any. At the end of every cache node chain that forms a polymorphic inline cache, there is a cache miss handler that is not compiled but transfers to the interpreter and rewrites the cache.

The cached set operation may also require a shape change, for instance to add a new property or change the type of an existing one. The cache implementation for this case is shown in Figure 10. We use the *setWithShape* method to set the value and to change the shape in a single atomic operation in order not to leave the object in an inconsistent state.

4.6 Inheritance

As a deliberate design decision, the Truffle OSM has no inherent concept of inheritance, owing to the divergence of inheritance models available. Instead, the inheritance semantics is to be modelled by language-specific extensions. Such extensions are implemented in terms of shape operations and inline cache nodes. The Truffle

```
class ObjectFieldLocation implements Location {
  private final long offset;
  private final Class<?> type;

  Object get(StorageObject obj, boolean condition) {
    return unsafeCast(unsafeGetObject(obj, offset,
        condition, this)), type, condition);
  }

  void set(StorageObject obj, Object value)
        throws IncompatibleTypeException {
    if (type.isInstance(value)) {
      unsafePutObject(obj, offset, value, this);
    } else {
      throw new IncompatibleTypeException();
    }
  }
}
```

Figure 8: Typed object field location implementation using Truffle compiler directives

```
void set(StorageObject object, Object value) {
  if (shapeGuard.check(object)) {
    try {
      cachedLocation.set(object, value);
    } catch (IncompatibleTypeException e) {
      transferToInterpreterAndInvalidate();
      generalizeType(value).set(object, value);
    }
  } else {
    nextCacheEntry.set(object, value);
  }
}
```

Figure 9: Inline cache node implementation for a setting the value of a cached existing property

```
void set(StorageObject object, Object value) {
  if (oldShapeGuard.check(object)) {
    try {
      cachedLocation.setWithShape(object,
          value, cachedNewShape);
    } catch (IncompatibleTypeException e) {
      transferToInterpreterAndInvalidate();
      generalizeType(value).set(object, value);
    }
  } else {
    nextCacheEntry.set(object, value);
  }
}
```

Figure 10: Inline cache node implementation for a setting the value of a property with a simultaneous shape change, for the purpose of adding a new property or changing its location

```
var result;
// global's shape is {result:<undefined>}
result = 42;
// global's shape is {result:int}
result = undefined;
// global's shape is {result:Object}
```

Figure 11: Delayed initialization of global variables in JavaScript

OSM offers *hidden properties* to store needed data in the shape or in an allocated object storage location.

4.7 Delayed Property Initialization

Some languages allow properties to be declared and even accessed before they are initialized. For example, this is the case for global variables in JavaScript, which are stored as properties of a special builtin object called global object. The JavaScript semantics for such properties prescribes that such properties exist with default value `undefined` until explicitly set. If every member were initialized with the default value, we would not be able to do specialization of primitive values effectively. To handle this class of cases, we create a placeholder location with the default value. As soon as its value is initialized with the actual value, we change the object's shape, replacing the location with a newly allocated one. Figure 11 shows how the Truffle JavaScript implementation handles global variable declarations. Initially the property *result* is assigned a static location with the value `undefined`. No space is reserved yet, as the desired representation is still unknown. When the variable is first assigned, we allocate a new location based on the value and perform a storage location transition. Setting an initialized variable to the default value (i.e., the value returned when reading an uninitialized variable, in our example the value `undefined`) conducts the usual type transition.

4.8 Behavior under Memory Pressure

The total number of shapes in the system is typically small, even for larger programs. However, pathological cases can be constructed that allocate an excessive number of shapes. With all the metadata associated with shapes, this can increase the pressure on the garbage collector and, in the worst case, eventually lead to out-of-memory errors. To reduce the memory pressure introduced by a large number of shapes, we take the following measures:

- The transition map uses `SoftReference`s for successor shapes, so that they can be released by the garbage collector under memory pressure.
- The properties collection in a shape is created lazily and only when actually used. It can be reconstructed (recursively) from the property collection of the predecessor shape by applying the changes of the transition that lead to this shape.
- The collections used for properties and transitions use lighter weight versions for small numbers of entries.

Moreover, the system handles extremely large objects by degrading to a fallback representation based on a hash table that does not require any shapes. This is particularly convenient to handle rare cases in which a program never stabilizes, for example in JavaScript, where objects can be sometimes used in an unusual way, with random entries added in a random order. This leads to a high number of shapes with little reuse. For such usage patterns, a simple hash table representation for properties is more efficient. Therefore, we provide a fallback object representation that stores all properties in a hash table.

4.9 AST Compilation

Our system is built on top of the Truffle language implementation framework and therefore requires a Graal-enabled version of the Java HotSpot VM to run at full speed [25]. When running on top of this JVM, Truffle interpreters are automatically partially evaluated and ASTs are compiled just-in-time to optimized machine code. Truffle can still be executed on any standard Java VM, but without partial evaluation. Partial evaluation takes the AST of a Truffle interpreter as the input and—acting under the assumption that it will not change—generates specialized compiled code without interpreter dispatch. Paths that change the state of the AST are excluded from compilation and cause a transfer back to the interpreter. The language implementer can also mark certain branches to be excluded from partial evaluation. This should be done for rarely taken branches in order to generate less and thus more optimized machine code.

4.9.1 Compiler Directives and API

The Truffle API offers a number of compiler directives to interact with the Graal JIT Compiler, if running on a Graal-enabled Java VM; otherwise these directives are simply ignored, leading to standard JVM behavior. Some directives are generally useful for Truffle users, such as the `transferToInterpreterAndInvalidate` directive, which instructs the compiler to replace subsequent code with a jump to a deoptimization routine transfering the execution back to the interpreter and invalidates the compiled code. It is used to exclude parts of the code from compilation when a speculation fails. These parts should only be executed in the interpreter.

The Truffle OSM uses additional compiler directives to optimize the compiled code:

`unsafeGet`*`Type`* loads a value or reference at the specified offset in the given Java object. *`Type`* is either `Object` or one of the primitive Java types. This directive enhances the `get`*`Type`* method of the `sun.misc.Unsafe` class with an optional location identity. Location identities can be used by the compiler for alias analysis. Two storage locations cannot alias if they have different location identities. If no location identity is provided, aliasing is treated as unknown. This method also requires a condition parameter that links the unsafe access to the condition under which it is valid (i.e., a shape comparison). It allows the Graal compiler to freely move the read anywhere below the condition in order to do more aggressive read optimizations. This parameter can always be `false` which means that the compiler cannot move the read.

`unsafePut`*`Type`* stores a value or reference at the specified offset in the given Java object. *`Type`* is either `Object` or one of the primitive Java types. This directive enhances the `put`*`Type`* method of the `sun.misc.Unsafe` class with an optional location identity.

`unsafeCast` casts a reference to a more specific type without a run-time check. This directive is used to inject type information to the compiler in order to remove unnecessary type checks where the object storage model already guarantees that the reference is of that type.

Furthermore, Truffle offers an API to register optimistic global *assumptions*. Technically, an assumption is a global boolean flag that is initially `true` and can be set to `false` a single time to invalidate the assumption. During partial evaluation, the state of the assumption is assumed stable and the assumption is registered as a dependency of the compiled code. If subsequently the assumption is invalidated, any dependent compiled code is automatically deoptimized. Consequently, assumption checks have no overhead in compiled code.

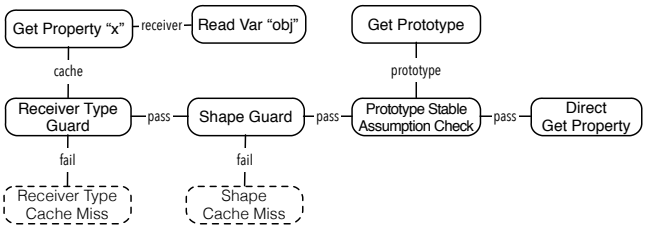


Figure 12: AST interpreter tree for the monomorphic cache of the JavaScript expression `obj.x` where the property is found in the prototype and the prototype shape is stable

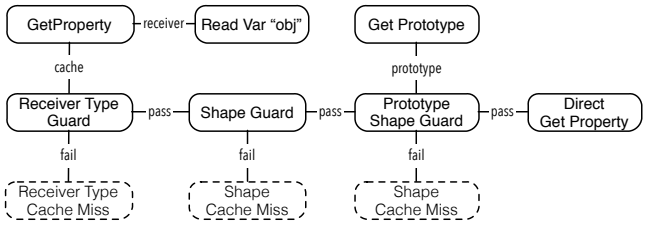


Figure 13: AST interpreter tree for the monomorphic cache of the JavaScript expression `obj.x` where the property is found in the prototype and the prototype shape is not stable

All these directives have implementations that are fully compatible with off-the-shelf Java virtual machines. While absence of the Graal-based Truffle runtime will lead to inferior performance, the code will stay fully functional.

5. Evaluation

We evaluate our approach focusing on two Truffle-based languages both relying on the Truffle OSM, namely JavaScript and Ruby. We select those two languages because of the maturity of their implementation. We measure the performance of the two engines and we compare against state-of-the-art engines, with the aim of showing that our Truffle-based approach achieves high peak performance. We evaluate our JavaScript performance using the Octane benchmark suite [8], and our Ruby performance using common micro-benchmarks. Since there is currently no Java implementation on top of Truffle to evaluate, we only discuss its requirements.

5.1 JavaScript

The JavaScript object model, as defined by the ECMAScript language specification edition 5.1 [6], requires a few extensions to be supported by our OSM. In order to retrofit the necessary semantics, we extend the object model by overriding the predefined operations from Section 3.2. For that, we provide three new boolean attributes as defined by the specification:

- *configurable*: denotes that the attributes of the property can be modified and that the property can be deleted.
- *writable*: denotes that the property's value can be set, otherwise it is treated as read-only.
- *enumerable*: denotes that the property is to be included when enumerating over an object's properties (getKeys, getValues).

Additionally, we add the boolean attribute *accessor* to denote accessor properties, for which the *get* and *set* operations have to invoke the getter and setter functions stored with the property.

Furthermore, we modify the OSM operations to take inheritance into account. JavaScript has a prototype-based inheritance model.

All objects have a hidden property that either is `null` or points to the object from which it inherits, i.e., its prototype. Unless an object has an own property with the same name, it inherits all properties of its prototype, and recursively from the prototype's prototype, thus forming a prototype chain. The prototype is stored as a property with the *hidden* attribute set to true. Prototypes, being just normal objects, are mutable. Therefore, a naive implementation would potentially have to walk the prototype chain on every property access. This differs from a class-based object model with an invariant set of members in the class hierarchy, where we know all members as soon as we know that an object is of a particular class.

To optimize property accesses and avoid walking the prototype chain on each access, we take the assumption that an object's prototype does not change. While prototypes can change, they rarely do in practice [20]. To exploit that, we use a combination of Truffle assumptions for checking the existence or absence of a specific property in the relevant prototypes and assigning a static storage location for the hidden property that stores the prototype, thus making it constant for a particular shape. Therefore, after checking the object's shape, we can directly access properties in the prototype. Only the generic method definition of the operation that is invoked for megamorphic accesses actually walks up the prototype chain. Figure 12 shows how we build an inline cache for a JavaScript expression `obj.x` that is found in the immediate prototype of `obj`. We add two additional nodes to the cache: The first node gets the prototype from the shape and reduces to a compile-time constant since the shape is known in the cache if the shape guard succeeds. The second node checks whether the property `x` in the prototype is stable using a Truffle assumption registered in the shape. This assumption is invalidated when a property with that name is removed or added to the prototype. An additional pair of nodes is inserted for each prototype passed along the prototype chain.

We also provide a fallback cache node for the rare case in which the prototype's shape is unstable. When the interpreter detects that the prototype is mutated, it invalidates the stable assumption and modifies the inline cache as depicted in Figure 13. Instead of the assumption check a normal shape check is done on the prototype. This shape check can again become polymorphic, i.e., if a new shape is encountered, it is added to the polymorphic dispatch chain.

5.2 Ruby

The Ruby object instance variable model is simpler than that of JavaScript. Objects logically contain a set of mappings from a string identifier to an object value. Instance variables are not declared before they are first set, and missing instance variables return the `nil` object, with no distinction being made between missing instance variables and those explicitly being set to `nil`. Different objects of the same class may have different sets of instance variables. Ruby does have explicit class declarations, similar to Python, and has a complex set of rules for class inheritance and method lookup, but this is not relevant for the implementation of instance variables as methods live in a different namespace. Finally, instance variables in Ruby are uniform without access modifiers or overloading as for example in the case of JavaScript's `length` property.

This means that the implementation of Ruby's instance variables using the OSM was straightforward. Objects are allocated with an empty shape and instance variables are added as properties on their first assignment. As in JavaScript, each source location where a property is read or written has an inline cache, with each entry in the cache specialized for a particular shape and storage location.

5.3 Towards Implementing Java

Our object storage model can also be used for implementing statically-typed languages. A Java implementation on top of Truffle (which is future work) can use the OSM for modeling instances of arbitrary Java classes. While such an implementation could use vanilla Java classes also for guest-language objects, emulating these classes with our storage model allows for specialized object formats that go beyond the Java language specification. For example, we could provide specialized representations of parametrized types, e.g., a `HashMap<String, double>` with String-typed keys (despite type erasure) and unboxed double values.

Java fields can be represented in the OSM as properties with additional attributes for access modifiers (private, protected, ...) and annotations. Non-access modifiers (static, final, volatile) can be modelled with corresponding storage locations. Finally, class metadata as well as method definitions can be stored as part of the *shared data* section in the shape. One caveat with using the OSM for languages that support multithreading is that we currently do not provide any guarantees about the thread-safety of shape transitions. However, since the fields of a class do not change after object creation, transitions are not required to implement Java as a guest language.

5.4 Performance

The benchmarks were run on an Intel Core i7-4850HQ quad-core processor at 2.3 GHz and 16 GB of RAM. We measured peak performance, i.e., a warmup phase preceeded each measurement to ensure a stable and comparable score.

Figure 14 and Figure 16 shows benchmark scores of our JavaScript and Ruby implementations with the following configurations of the OSM:

NoOpt OSM objects contain only properties of type `Object` without any type specialization. Primitive values are always stored as boxed objects. Still, inline caching is performed on the object's shape in order to specialize property accesses.

Types Properties have type information attached. Values are always boxed.

Unbox No type information is attached to properties with a reference type. Values are stored in unboxed mode in the primitive storage area.

Types+Unbox All features are enabled. Properties with a reference type have type information attached. Where possible, values are stored in unboxed mode in the primitive storage area.

Hash In the Ruby implementation we also compared against a fallback HashTable implementation of instance variables.

5.4.1 JavaScript

In Figure 14, we compare the aforementioned configurations of the OSM on the Octane benchmark suite version 2.0 [8]. We have excluded the benchmarks *RegExp* and *CodeLoad* as they are testing only aspects unrelated to the OSM, namely regular expression matching and JavaScript library loading speed, respectively.

Type specialization of locations with a reference type alone achieves a minor speedup with up to 9% on the *RayTrace* benchmark. Primitive type specialization, i.e., storing values in unboxed mode, achieves a much higher speedup, namely up to 2× on the *Box2D* benchmark. Notable improvements can also be seen on *RayTrace* (57%), *Richards* (38%), *DeltaBlue* (26%), and *Splay* (26%). Finally, combining those two optimizations shows that these speedups accumulate, with up to 2.4× on *Box2D*. Some of the benchmarks do not benefit at all from the optimizations in our OSM. These benchmarks mostly work on arrays or strings and therefore do not profit from fast object access.

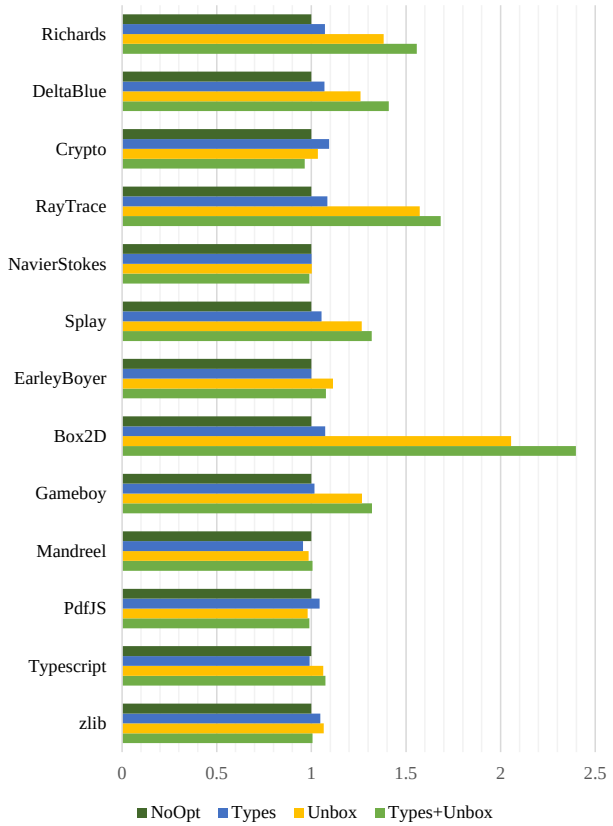


Figure 14: JavaScript Octane benchmark score comparison with different storage configurations, normalized to the NoOpt configuration

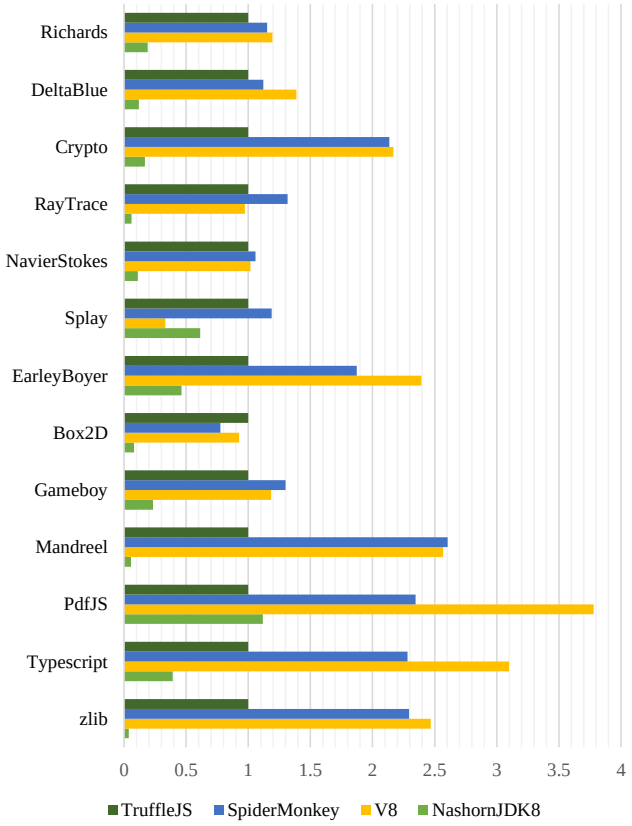


Figure 15: Octane benchmark score comparison against other JavaScript engines

Figure 15 shows a comparison of our Truffle JavaScript implementation against recent versions of V8 [9], SpiderMonkey [17], and Nashorn [19] as included in JDK 8u5. All engines feature a SELF-like map system for faster property access. V8 and SpiderMonkey are VMs written in C++ and optimized specifically for JavaScript and use tagging as a means to avoid the overhead of storing values as boxed objects. Recent V8 versions also have some type information in *maps* that allow the compiler to skip tag checks (values are still tagged). Nashorn, since it is based on Java, does not use tagging but instead boxes all values. This has a negative impact on object-oriented benchmarks that perform a lot of numeric computations, as can be seen with the *RayTrace* and *Box2D* benchmarks. Our implementation always attempts to store values unboxed (and untagged) and only boxes values if their type is polymorphic. Despite the fact that Truffle JavaScript runs on a general-purpose VM (the JVM), aggressive optimizations allow us to be competitive in terms of performance with special-purpose VMs.

5.4.2 Ruby

Figure 17 shows a comparison of our Truffle Ruby implementation against the latest versions of MRI, Rubinius, JRuby and a nightly build of Topaz[2]. The original implementation of Ruby, written in C and known as *MRI* or CRuby, is a conventional bytecode interpreter. It stores instance variables in a hash table of names to an index in an array stored in each object. The index is cached after first lookup and stored alongside bytecode instructions. The cache is verified with an explicit check against a version number stored in the receiver's class.

Rubinius is Ruby implemented with a VM core in C++ and using LLVM as a JIT compiler, but with much of the Ruby specific functionality implemented in Ruby. It stores instance variables that are statically known at parse time in fields within the object, but uses a hash table to store dynamically set instance variables. This results in drastically different performance depending on whether or not an instance variable happens to be visible to the parser – a modified version of the n-body benchmark where instance variables are hidden from the parser by mixing them in from a separate module after allocation performs at half speed in Rubinius. As Truffle does not try to guess instance variables using the parser, hiding them makes no difference to our performance and the modified version of n-body runs at the same speed.

JRuby is a conventional bytecode generating JVM implementation of Ruby, which our Truffle implementation extends upon. It uses the `invokedynamic` instruction, but not to the extent which Nashorn does. JRuby stores instance variables in a similar way to MRI, with variables stored boxed in an object array in each object with the index looked up in a hash map. As in MRI, an inline cache of the index lookup is guarded by the class version number. JRuby has experimental support for the same technique as Rubinius for statically allocating fields for instance variables, but we were not able to get it to work with our modified n-body.

Topaz is an implementation of Ruby using the RPython toolchain, as used to implement PyPy. Topaz is implemented in a restricted subset of Python and translated ahead of time to C. At runtime, a

[2] available at: http://ruby-lang.org/, http://rubini.us/, http://jruby.org/, and http://topazruby.com/

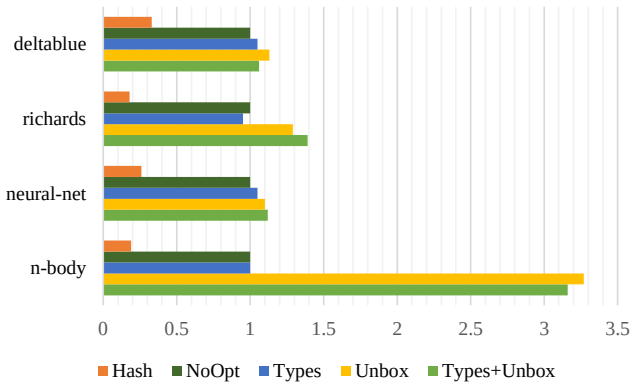


Figure 16: Ruby benchmark score comparison with different storage configurations, normalized to the NoOpt configuration

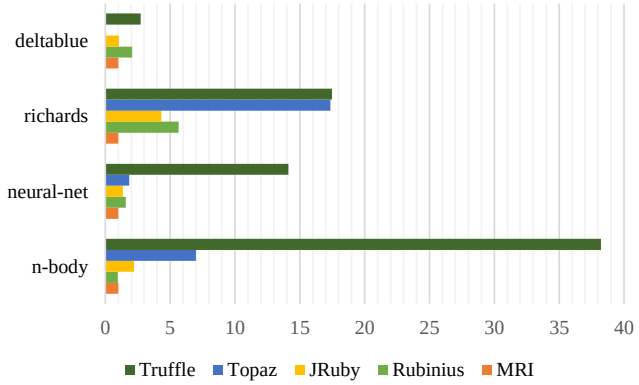


Figure 17: Benchmark score comparison against other Ruby engines, normalized to MRI

metatracing JIT traces execution of the interpreter and emits machine code. Topaz stores instance variables using a map of names to an index in separate arrays for objects and unboxed values, stored in each object. Unlike MRI where the map is a simple hash table, in Topaz it is a data structure similar to our shape, with transitions as variables are added. Topaz was not able to run the deltablue benchmark.

6. Related Work

Cross-language interoperability presents many challenges [4] spanning from memory layout compatibility, differences in the object model and the memory model, and differences in the model of parallelism to be supported. All such challenges can be found either when attempting to integrate two distinct language runtimes (e.g., Smalltalk and C [3]) or when languages share the same runtime (e.g., the JVM). Most of the research in the latter domain has been conducted addressing the issues arising from the sharing of core VM components such as the JIT compiler. For instance, it is very common that a JIT compiler designed to target a statically typed language has been modified in order to target also dynamic languages [1, 14]. The Truffle framework and its core JIT compiler, Graal [25], can be considered as a flexible platform for reusing core VM components such as the JIT compiler and the garbage collector. A distinguishing difference with existing approaches is that core VM components are offered to the Truffle language implementer in the form of compiler directives and APIs that are known to the runtime. The storage model described in this paper is one such core component available for language implementers to be used when developing a language execution engine. Other real-world examples of cross-language interoperability also exist in the context of the .NET common language runtime (CLR), where language implementers are required to adopt the same common object model for all languages [16]. The .NET object model comes with support for dynamically extensible objects by means of a shared API[3], and differs significantly from the Truffle approach, since developers must rely on a fixed, not extensible, set of APIs. Moreover, the Truffle OSM is designed to target Truffle AST interpreters, offering opportunities for optimization. Conversely, languages willing to support different semantics in .NET need to choose between emulating them on top of what the .NET platform offers or creating a dialect of the guest language.

An approach that is an alternative to extending existing runtimes for supporting multiple languages is represented by the VMKit project [7]. VMKit provides developers with reusable core VM components that can be glued together in order to obtain a managed language runtime by means of component reuse and composition. The VMKit approach differs from the Truffle approach in that the framework does not offer the language implementer any means for specializing the language runtime to the characteristics of a specific target language (e.g., by means of specialization and partial evaluation) or to reuse an object storage model. Rather, the object model design and implementation is left to the developer who has to take care of all the implementation aspects.

Regardless of cross-language interoperability, managed runtimes already feature optimization techniques such as polymorphic inline caches, which were first introduced in the SELF language runtime [2, 22]. We embrace the effectiveness of polymorphic inline caches and build on the SELF approach by providing an object model that can be used for implementing reusable inline caches applicable in the context of multiple languages. As we have shown, inline caches for different languages such as JavaScript and Ruby can be implemented with our OSM relying on the same mechanisms and still offering acceptable performance.

7. Future Work

While the Truffle Framework is already adopted by several language implementation efforts, it is still under active development. The Truffle object storage model presented in this paper addresses a core aspect of Truffle with the goal of providing a powerful way for language developers to implement high-performance language runtimes. To further increase and improve the languages running on the Truffle platform, we foresee the following research directions:

Language Support We claim that our OSM is generic enough to be used for a wide selection of Truffle language implementations. Future work should prove that this is indeed the case through increased adoption by applying our OSM to even more language implementations, also of different classes.

Thread Safety If the shape of an object does not change, our object model is already thread-safe in the sense of the Java Memory Model [15]. Thus, it can be used to safely implement languages with fixed object formats (such as Java) in a multi-threaded environment. Ensuring thread safety also across format-changing shape transitions is left as future work.

[3] The API includes built-in functions for dynamically changing the size of an object such as `ExpandoObject`, `DynamicObject`, `DynamicMetaObject`, `IDynamicMetaObjectProvider`.

Value Types The primitive storage in our system could be used to efficiently store value types (e.g., complex numbers) without the need of wrappers, even if those are not supported by Java. We plan to demonstrate this in a future project.

Reducing Polymorphism Sometimes, overly precise type and representation specialization can lead to an increase in types. If two shapes contain the same members (but differ in their types), they can be merged and their member types can be generalized to a common base type. This would reduce the number of shapes in the system.

Cross-Language Interoperability We think that the shared Truffle infrastructure provides a solid basis for executing multi-language programs with interoperability across language boundaries. In future work we will investigate non-intrusive and efficient ways to achieve this goal [11].

8. Conclusions

In this paper we introduced and evaluated Truffle OSM, a new, high-performance, object storage model for the Truffle language implementation framework. Truffle OSM defines a sophisticated memory layout for objects that can have varying shapes together with the necessary operations for accessing the members of these objects efficiently. Our approach is suitable for very efficient compilation, as it is based on type specialization and partial evaluation, allowing for optimizations such as polymorphic inline caches for efficient dynamic property lookup.

Given the heterogeneity of programming languages, there is no one-fits-all solution to the problem of providing language implementers with generic reusable components. Still, we believe that our one-fits-many approach is very convenient for language implementers, as it focuses on widely useful features rather than on language-specific ones.

Moreover, we believe that our approach encourages the usage of Truffle, as the Truffle OSM API allows the compiler reuse many aggressive optimizations for all the new Truffle languages that might rely on it. Future optimizations will therefore automatically benefit all users. Our performance evaluation shows that even though Truffle-based guest-language runtimes build on top of a general-purpose host VM, they can achieve performance competitive to that of a special-purpose VM designed for a specific dynamic programming language.

Acknowledgments

We thank all members of the Virtual Machine Research Group at Oracle Labs, as well as the Institute for System Software at the Johannes Kepler University Linz, for their support and contributions.

References

[1] J. Castanos, D. Edelsohn, K. Ishizaki, P. Nagpurkar, T. Nakatani, T. Ogasawara, and P. Wu. On the benefits and pitfalls of extending a statically typed language JIT compiler for dynamic scripting languages. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 195–212, 2012.

[2] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 49–70. ACM Press, 1989.

[3] D. Chisnall. Smalltalk in a C world. In *Proceedings of the International Workshop on Smalltalk Technologies*, pages 4:1–4:12, 2012.

[4] D. Chisnall. The challenge of cross-language interoperability. *Queue*, 11(10):20:20–20:28, Oct. 2013.

[5] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. 2013.

[6] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. 5.1 edition, June 2011.

[7] N. Geoffray, G. Thomas, J. Lawall, G. Muller, and B. Folliot. VMKit: A substrate for managed runtime environments. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 51–62, 2010.

[8] Google. Octane benchmark suite, 2014. URL https://developers.google.com/octane/.

[9] Google. V8 JavaScript engine, 2014. URL http://code.google.com/p/v8/.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, Third Edition*. Addison-Wesley Professional, 2005.

[11] M. Grimmer, T. Würthinger, A. Wöß, and H. Mössenböck. An efficient approach for accessing C data structures from JavaScript. In *Proceedings of the Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. ACM Press, 2014.

[12] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 21–38. Springer-Verlag, 1991.

[13] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–43. ACM Press, 1992.

[14] K. Ishizaki, T. Ogasawara, J. Castanos, P. Nagpurkar, D. Edelsohn, and T. Nakatani. Adding dynamically-typed language support to a statically-typed language compiler: Performance evaluation, analysis, and tradeoffs. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 169–180, 2012.

[15] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 378–391, 2005.

[16] E. Meijer and J. Gough. Technical Overview of the Common Language Runtime. Technical report, 2000.

[17] Mozilla Foundation. Spidermonkey JavaScript engine, 2014. URL http://developer.mozilla.org/en/SpiderMonkey.

[18] OpenJDK Community. Graal project, 2014. URL http://openjdk.java.net/projects/graal/.

[19] OpenJDK Community. Nashorn project, 2014. URL http://openjdk.java.net/projects/nashorn/.

[20] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12. ACM Press, 2010.

[21] L. Stadler, T. Würthinger, and H. Mössenböck. Partial escape analysis and scalar replacement for Java. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 165–174. ACM Press, 2014.

[22] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 227–242, 1987.

[23] T. Van Cutsem and M. S. Miller. Proxies: Design principles for robust object-oriented intercession APIs. In *Proceedings of the Dynamic Languages Symposium*, pages 59–72, 2010.

[24] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing AST interpreters. In *Proceedings of the Dynamic Languages Symposium*, 2012.

[25] 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 Onward!*, 2013.