Applying Optimizations for Dynamically-typed Languages to Java

Matthias Grimmer
Oracle Labs
Austria
matthias.grimmer@oracle.com

Stefan Marr
Institute for System Software
Johannes Kepler University, Linz
stefan.marr@jku.at

Mario Kahlhofer
Institute for System Software
Johannes Kepler University, Linz
mario.kahlhofer@jku.at

Christian Wimmer
Oracle Labs
United States
christian.wimmer@oracle.com

Thomas Würthinger
Oracle Labs
Switzerland
thomas.wuerthinger@oracle.com

Hanspeter Mössenböck
Institute for System Software
Johannes Kepler University, Linz
hanspeter.moessenboeck@jku.at

ABSTRACT

While Java is a statically-typed language, some of its features make it behave like a dynamically-typed language at run time. This includes Java's boxing of primitive values as well as generics, which rely on type erasure.

This paper investigates how runtime technology for dynamically-typed languages such as JavaScript and Python can be used for Java bytecode. Using optimistic optimizations, we specialize bytecode instructions that access references in such a way, that they can handle primitive data directly and also specialize data structures in order to avoid boxing for primitive types. Our evaluation shows that these optimizations can be successfully applied to a statically-typed language such as Java and can also improve performance significantly. With this approach, we get an efficient implementation of Java's generics, avoid changes to the Java language, and maintain backwards compatibility, allowing existing code to benefit from our optimization transparently.

CCS CONCEPTS

• **Software and its engineering → Compilers**; *Interpreters*; *Dynamic compilers*; *Runtime environments*; *Object oriented languages*;

KEYWORDS

Java; bytecode interpreter; dynamic compilation; virtual machine; language implementation; optimization

ACM Reference Format:
Matthias Grimmer, Stefan Marr, Mario Kahlhofer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. 2017. Applying Optimizations for Dynamically-typed Languages to Java. In *Proceedings of ManLang 2017, Prague, Czech Republic, September 27–29, 2017,* 11 pages.
https://doi.org/10.1145/3132190.3132202

ManLang 2017, September 27–29, 2017, Prague, Czech Republic

1 INTRODUCTION

Java is a statically-typed programming language that is executed by highly efficient and mature virtual machines. Despite its success, various issues have been identified with Java's design including its type-erasure approach to generics, its missing compound value types, and the fact that primitives and arrays are treated differently from other objects.

Java implements generics by erasing type parameters to their bound (`Object`) when transforming source code to Java bytecode. In contrast, C++ uses compile-time template expansion for generics and generates native code that is separately optimized for each expansion. While this approach allows for type-specific optimizations and avoids issues with respect to primitive type parameters and generic arrays, code is not shared between separate expansions, which has a negative effect on the footprint of applications.

Also, Java distinguishes primitive types from references types, which requires it to use boxed representations for primitives in the context of generics. To ease their use at the language level, Java automatically boxes and unboxes primitives where necessary. This so-called autoboxing comes with a memory and performance overhead caused by the wrapper objects for the primitive values. In order to overcome these issues, the project Valhalla[1] explores the approach of generating different bytecodes depending on whether the generic type parameter is a primitive type or a reference type, which is a restricted version of C++'s approach. Project Valhalla's requirement is to maintain compatibility with existing bytecode. Another requirement is to avoid imposing Java semantics on the Java Virtual Machine (JVM). However, Project Valhalla's approach has implications for Java itself: primitive types and reference types do not have a common super type, hence, the wildcard operator (`?`) has to be deprecated and cannot be used anymore in this scenario.

In this paper, we present an alternative solution. We demonstrate a novel approach to executing JVM bytecode and show that optimizations designed for dynamically-typed languages can be used to implement Java's generics more efficiently. Furthermore, our approach avoids changes to the Java bytecode and the Java language itself, which makes it a completely transparent and backward-compatible solution.

[1] *Project Valhalla*, http://openjdk.java.net/projects/valhalla/

To demonstrate our approach, we use Truffle [30, 31], a language implementation framework that was designed for the implementation of dynamically-typed languages on top of a JVM, and use it to implement a Java bytecode interpreter, which we call TruffleBC. Our performance evaluation shows that it is possible to outperform state of the art JVMs by applying code and data optimizations designed for dynamically-typed languages to a statically-typed language such as Java. To summarize, our contributions are:

- We use VM and compiler technology originally designed for dynamically-typed languages to efficiently execute Java bytecode. Specifically, we introduce a Truffle-based Java bytecode interpreter and show that dynamic compilation techniques can be applied to statically-typed languages such as Java to improve their performance significantly.
- We use Java's generics as a case study and demonstrate how data structure specialization and type specialization of Java bytecode can improve the performance of Java programs.
- We provide a performance evaluation (peak performance and start-up performance) that compares TruffleBC to Oracle's HotSpot VM, which confirms our claim that dynamic-language VM technology can be efficiently used for Java.

2 BACKGROUND AND PROBLEM STATEMENT

The Java Virtual Machine (JVM) is a multi-language platform that supports executing various programming languages that compile to JVM bytecode. As a background for the remainder of the paper, we briefly explain Java bytecode and how it is normally executed by a JVM. Based on that, we describe how Java generics are currently implemented and outline the issues with the current approach. Finally, we explain Truffle and Graal, a language implementation framework that we use to implement an alternative execution environment for Java bytecode.

2.1 The Java Virtual Machine and Java Bytecode

Java bytecode is an instruction set for a virtual stack machine. This means that the operations are generally designed to consume operands from the stack and to push results back onto it. More specifically, the JVM distinguishes between three locations where data is stored: the operand stack of the stack machine, a method-local frame that contains local variables, and the Java heap where objects are stored. Java handles primitive types differently from reference types, which is also reflected in the bytecode. Java bytecode defines different instructions for working with primitives and with references. The instructions have prefixes and/or suffixes that refer to the types of their operands. For example, the `iload` instruction is of type `int`; it loads an `int` variable and pushes it onto the operand stack. In contrast, the `aload` instruction is of type *reference*; it loads a local reference variable and pushes it onto the operand stack.

2.2 Problem Statement

As sketched in the introduction, Java's design of handling primitive values and references differently poses problems for optimizing generics.

While Java generics are type checked at compile time, type parameters are then removed (type erasure) so that they are not available at the JVM level anymore. Thus, `ArrayList<Integer>` is compiled to the raw type `ArrayList` and the type parameter `Integer` is erased to its bound, namely `Object`. Inside the raw type all occurrences of the type parameter are represented as `Object`, so that the class `ArrayList` (see Listing 1) internally uses an `Object[]` to store its values.

Since only reference types can be mapped to `Object`, type parameters must be reference types in Java. Primitive types have to be boxed before they can be used as type parameters. Java's autoboxing makes it convenient to use generics also with primitives (see Listing 1). However, autoboxing causes a significant performance and memory overhead: it creates extra work for the garbage collector and requires an additional indirection during access. In particular, `Object[]` are less efficient to use than `int[]`. Our goal is to eliminate this overhead with dynamic compilation technology, which is normally used for dynamically-typed languages. We describe our approach in more detail in Section 4.

2.3 Truffle and Graal

For our case study, we use the Truffle [31] language implementation framework and build an interpreter for Java bytecode. We do not extend or modify an existing state-of-the-art JVM (e.g. Oracle's HotSpot VM) because this would require a complex re-engineering of this VM. We would need to adapt the interpreter, at least one of the two just-in-time compilers, the object representation in memory, and the garbage collector to experiment with our optimizations. Using Truffle allows us to easily implement our optimizations and also to reuse all other VM components (dynamic compilers, garbage collector, etc.) without modification.

Truffle is a platform for building high-performance language implementations in Java. Truffle language implementations are interpreters, running on top of a Java Virtual Machine. Source code (or any other code representation - in our case Java bytecode) is transformed to a graph of nodes (e.g. an abstract syntax tree), which can then be executed and is eventually dynamically compiled by the Truffle framework. Nodes within this graph represent instructions of the guest language, which can be evaluated. All nodes extend a common `Node` class and have a method that evaluates them. The whole graph is evaluated by executing these methods on the nodes recursively, starting with the root node. Note, for most purposes the graph has a tree structure. Thus, in the remainder of the paper we consider it to be a tree, even if it might technically not be one in the strict sense.

An important characteristic of Truffle trees is that they are *self-optimizing* [32]. Self-optimizations (we will also call them *specialization*) are typically based on profile information (e.g., type information) collected during execution. Based on this information, Truffle languages try to use the most specific operation possible in the context of an executing program to avoid the overhead of a more general solution. For example, Truffle trees specialize as

```
1  List<Integer> list = new ArrayList<>(10);  // Internally backed by an Object[]
2  list.set(0, 42);                           // auto-box 42 to an Integer object
3  // ...
4  int i = list.get(0);                       // auto-unbox 42
```

Listing 1: Java application using an `ArrayList`.

a reaction to type feedback, replacing a dynamically-typed add operation node (with different semantics depending on the types of the operands) that receives two integers with a node that only performs integer addition and is thus simpler. The Truffle framework encourages the optimistic specialization of trees, where nodes can be replaced with a more specialized (and thus more efficient) node. If the assumptions on which the specialization was based turn out to be wrong during further execution, a specialized node has to be reverted back to a form that provides more general functionality. Specialization via tree rewriting is a general mechanism for dynamically optimizing code at run time. This technique is the foundation for our optimization of bytecodes for reference types at run time.

To represent method activations, the Truffle framework has the notion of a `Frame`. It is essentially an array holding local variables and other data of a method, which provides support for specializing its values to primitive types to avoid boxing. Similarly, to implement dynamic data structures efficiently, Truffle provides an object storage model [29], which is based on a type called `DynamicObject`. The `DynamicObject` supports dynamic resizing of objects and allows adding and removing members at run time, which is optimized using maps [4]. Both mechanisms are used by us to represent method activations and objects efficiently and to enable our optimizations.

To achieve high performance, Truffle partially evaluates [30, 31] the trees and dynamically compiles them to optimized machine code. For dynamic compilation it uses the *Graal* compiler [9, 10, 12, 24–26]. Partial evaluation means that the Graal compiler inlines all node execution methods of a tree into a single method and hereby assumes that the tree remains stable. The compiler inserts so-called guards that check if the speculative assumptions still hold at run time. If one of the guards would fail (i.e., a node or a subtree would rewrite), then the machine code *deoptimizes* [15]. Deoptimization transfers the control back from compiled code to the interpreted tree, at which specialized nodes are reverted to a more general version. Truffle and the Graal compiler are part of the Graal VM, a modification of the HotSpot VM: it adds the Graal compiler and Truffle, but reuses all other parts of the HotSpot VM, including the garbage collector and the interpreter.

3 JAVA BYTECODE EXECUTION ON TOP OF TRUFFLE

This section describes the implementation of TruffleBC. It details our approach to implement an efficient execution system for Java bytecode with techniques for dynamic languages on top of Truffle.

A high-level overview of the architecture is given in Figure 1. TruffleBC is running on the Graal VM. To load bytecode into TruffleBC, we use the JVM class loader. However, bytecode is not directly executed on the JVM. Instead, TruffleBC parses the bytecode and

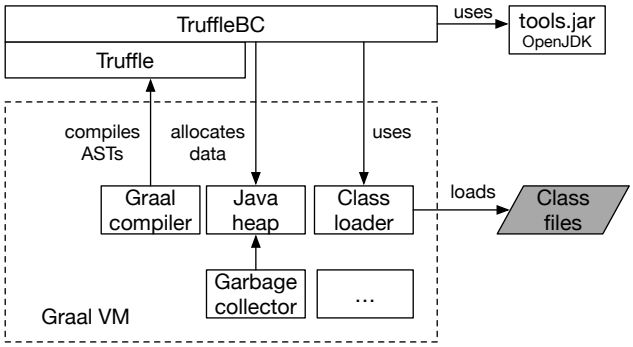

Figure 1: Layers of a Truffle language implementation: TruffleJava is hosted by the Truffle framework on top of the Graal VM.

transforms it to a different representation (see Section 2.3) which is then interpreted.

To represent Java objects, we use Truffle's `DynamicObject` type (cf. Section 2.3). The stack for local variables and the operand stack are represented with Truffle's `Frame` objects. Objects of type `DynamicObject` and `Frame` are regular Java objects that are allocated on the heap. Hence, TruffleBC reuses the garbage collector of the underlying Graal VM.

3.1 Local Variables, Operand Stack, Object Allocations

In Java bytecode, the number of local variables as well as the maximum size of the operand stack can be determined statically from the bytecode. Therefore, TruffleBC allocates a `Frame` object for every method invocation which is large enough to hold the local variables of this method as well as the operand stack. Taking Listing 2 and Listing 3 as an example, TruffleBC allocates one slot in the `Frame` to store the local variable `i` and another two slots for the operand stack, which has a maximum size of 2 to represent the comparison of `i` and `1000`.

To represent a Java object as a `DynamicObject`, we add a slot for every field of the object as well as a slot for its class, which is also represented as a `DynamicObject`. Using `DynamicObject`s instead of a direct mapping to Java objects allows us to change the type of fields at run time, which we need in order to specialize data at run time (see Section 4).

```
1  public static void foo() {
2    int i = 0;
3    do {
4      i++;
5    } while(i < 1000);
6  }
```

Listing 2: Java function `foo` with a counter variable incremented in a loop.

```
1  iconst_0
2  istore_0
3  iinc          0, 1
4  iload_0
5  sipush        1000
6  if_icmplt     2
7  return
```

Listing 3: Java Bytecode of function `foo`.

```
1  int bci = 1;
2  while (bci != -1) {
3    int next = bcNodes[bci].execute(frame);
4    bci = bcNodes[bci].successors[next];
5  }
```

Listing 4: The bytecode dispatch node.

3.2 Interpretation of Java Bytecode

TruffleBC provides a Truffle `Node` implementation for all 198 Java bytecodes, where each node implements the semantics of the corresponding bytecode in its `execute` method. The `Frame` object of a Java function is passed as an argument to the `execute` methods, which allows them to access local variables as well as the operand stack.

To support any unstructured control flow [20] that can be expressed with Java bytecodes, TruffleBC uses a so-called *bytecode dispatch node*. This node is the root node in each function. A bytecode dispatch node has an array of nodes called `bcNodes` to hold all bytecode nodes of a function. As sketched in Listing 4, the `execute` method of the bytecode dispatch node has a loop, which executes one of the bytecode nodes in each iteration, starting from bytecode index one (`bci = 1`).

Each bytecode node has a `successors` array of indexes identifying its possible successor bytecodes. Most nodes have only one successor, hence the `int[]` has usually only one element. Using Listing 3 as an example, the bytecodes `iconst_0`, `istore_0`, `iinc`, `iload_0`, and `sipush` have just a single unconditional successor. In contrast to that, `if_icmplt` has two possible successors. Depending on the result of the comparison, the control flow can be transferred to bytecode index 3 or index 7. Hence, the `successors` array for `if_icmplt` is `[3, 7]`. The array of successors is important for dynamic compilation (see Section 3.3); it allows the compiler to determine all possible successor bytecodes. When executing a bytecode node, the `execute` method returns an index to the `successors` array, which is then used to retrieve the successor `bci`. Execution continues until `bci = -1`, which indicates a return statement.

With respect to the example in Listing 3, the bytecode dispatch node transfers execution between the seven bytecode nodes with indices `[1, 2, 3, 4, 5, 6, 7]`. Figure 2 shows the bytecode dispatch node for the bytecode in Listing 3 and illustrates the control flow with red arrows.

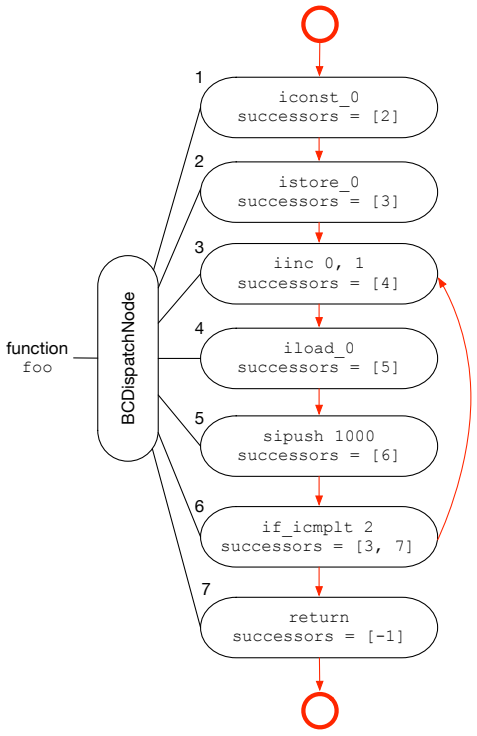

Figure 2: Truffle Graph for function `foo` of Listing 3.

Execution starts with `bci = 1`: `iconst_0` is executed. The `iconst_0` node has a direct successor (`bci = 2`, hence the `sucessors = [2]`). The bytecodes are consecutively executed until the `if_icmplt` node is reached. This node has two possible successors (`sucessors = [3, 7]`, the loop body at `bci = 3` and the loop exit at `bci = 7`). The bytecode dispatch node executes the `if_icmplt` node and sets the `bci` to either 3 or 7 after accessing the `successors` array. The successor of the `return` node is `bci = -1`, which signals a return from the function.

3.3 Compilation

If a method has been executed for a certain number of times, Truffle triggers its compilation by the Graal compiler, which inlines all `execute` methods of the graph nodes and all methods they call into a single compilation unit. When compiling the bytecode dispatch node, Graal unrolls the loop (`while (bci != -1)`, see Listing 4) until all paths through a function are expanded.

For the function in Figure 3, the compiler starts at `bci = 1` and determines the successor of the first node (`iconst_0`), which is the node `istore_0`. It then peels the first iterations and moves the execution of `iconst_0` and `istore_0` out of the loop. Listing 5 shows the loop after this peeling. Next, the compiler peels the execution of `iinc`, `iload_0`, and `sipush` out of the loop. At bytecode `if_icmplt` the compiler sees that this bytecode has two possible successors, namely `bci = 3` (i.e., the loop body)

```
1  bcNodes[1].execute(frame);              // iconst_0
2  int next = bcNodes[2].execute(frame);   // istore_0
3  int bci = bcNodes[2].successors[next];
4  while (bci != -1) {
5    next = bcNodes[bci].execute(frame);
6    bci = bcNodes[bci].successors[next];
7  }
```

Listing 5: Step 1: Unrolling the loop of the basic block dispatch node.

```
1  bcNodes[1].execute(frame);              // iconst_0
2  bcNodes[2].execute(frame);              // istore_0
3  merge_bci_3:
4  bcNodes[3].execute(frame);              // iinc 0, 1
5  bcNodes[4].execute(frame);              // iload_0
6  bcNodes[5].execute(frame);              // sipush 1000
7  int next = bcNodes[6].execute(frame);// if_icmplt 2
8  int bci = bcNodes[6].successors[next];
9  if (bci == 3) {
10   goto merge_bci_3;
11 } else if (bci == 7) {
12   while (bci != -1) {
13     next = bcNodes[bci].execute(frame);
14     bci = bcNodes[bci].successors[next];
15   }
16 }
```

Listing 6: Step 2: Unrolled loop of the basic block dispatch node.

```
1  bcNodes[2].execute(frame);              // iconst_0
2  bcNodes[3].execute(frame);              // istore_0
3  merge_bci_3:
4  bcNodes[3].execute(frame);              // iinc 0, 1
5  bcNodes[4].execute(frame);              // iload_0
6  bcNodes[5].execute(frame);              // sipush 1000
7  int next = bcNodes[6].execute(frame);// if_icmplt 2
8  int bci = bcNodes[6].successors[next];
9  if (bci == 3) {
10   goto merge_bci_3;
11 } else if (bci == 7) {
12   bcNodes[7].execute(frame);            // return
13   return;
14 }
```

Listing 7: Final stage: Unrolled loop of the basic block dispatch node.

and `bci = 7` (i.e., the loop exit). The compiler always checks if a path has already been expanded (which is the case for successor `bci = 3`). In that case it merges this path with the existing path by inserting a backjump (`goto merge_bci_3`), which guarantees that the loop expansion terminates. Listing 6 shows the loop after peeling `if_icmplt`. Finally, the compiler peels the `return` bytecode out of the loop. This bytecode node does not have any successors (indicated by a successor with `bci = -1`), which terminates the loop and the compiler finishes loop unrolling because all paths are expanded (see Listing 7). After unrolling this loop, the compiler further optimizes the code and eventually produces machine code.

3.4 Completeness

TruffleBC does not yet support all Java features. Our focus was on core features of Java, which are relevant to evaluate the specialization of bytecode and data at run time. Currently, TruffleBC does not support:

- **Reflection:** Java's reflection API is not yet implemented. We do not foresee any difficulties with it with respect to the focus of this paper, since solutions have already been shown for dynamic languages such as Python and JavaScript, which have similar reflective capabilities.
- **Multi-threading:** Threading is not yet supported. While thread-safety is a major concern for optimizations as discussed by Daloze et al. [7], we consider it out of scope for this paper.
- **JNI:** Native calls to C/C++ code via JNI are not yet fully supported. The C/C++ code is executed by Sulong [22], an implementation of C/C++ on top of Truffle. The native interface itself is implemented using Truffle's cross-language interoperability mechanism [13, 14].
- **Security and Serialization:** In contrast to a production JVM, TruffleBC does not yet implement a security manager as a production JVM does. It also does not yet support object serialization that is compatible with normal JVM semantics. However, we consider both aspects orthogonal to the ideas discussed in this paper.

4 SELF-OPTIMIZING JAVA BYTECODE EXECUTION

A Truffle language implementation uses two different levels of optimization: language implementers *optimize at the graph level* (i.e., they specialize nodes and data by implementing possible optimizations), while the Truffle framework itself provides *partial evaluation and dynamic compilation of Truffle trees*. Optimizations at graph level allow us to apply optimizations that go beyond the current level of optimizations for statically-typed languages.

This section describes the three optimizations that we apply to Java bytecode: data specialization, specialization of reference bytecodes, and removal of autoboxing and autounboxing. To better illustrate these optimizations we use the example in Listing 1 and decompose it into the following parts:

- Listing 8: The example shows the `ArrayList` constructor, which allocates a new `Object[]` using the `anewarray` bytecode.
- Listing 9: Because of type erasure, `ArrayList::set` requires an `Object` argument. Hence, the Java compiler automatically boxes the value `42` by inserting an invoke of `Integer.valueOf`, which returns the boxed object representation of `42`.
- Listing 10: The method `ArrayList::set` contains an `aastore` instruction that writes the value to the `Object[]`.
- Listing 11: The method `ArrayList::get` contains an `aaload` instruction that loads the value from the `Object[]`.
- Listing 12: The call to `ArrayList::get` returns a boxed integer value, which is automatically unboxed. The compiler inserts a call to `intValue()` on the boxed receiver value. The primitive value is then stored to variable `i`.

```
1  iload_1       // push size onto the operand stack
2  anewarray     // allocate a java/lang/Object array
```

Listing 8: ArrayList constructor.

```
1  bipush 42
2  invokestatic  // Method Integer.valueOf
```

Listing 9: Auto-boxing of argument 42 for `ArrayList::set`.

```
1  getfield      // elementData
2  iload_1       // push index argument onto the stack
3  aload_2       // push value argument onto the stack
4  aastore       // store value into array
```

Listing 10: Array store in method `ArrayList::set`.

```
1  getfield      // elementData
2  iload_1       // push index argument onto the stack
3  aaload        // load element from array
```

Listing 11: Array load in method `ArrayList::get`.

```
1  checkcast     // check if result is Integer object
2  invokevirtual // Integer.intValue
3  istore_1      // store restult into variable i
```

Listing 12: Auto-unboxing of return value and storing it to variable `i`.

4.1 Data specialization

To specialize the representation of a variable, an object field, or an element of an `Object[]`, we use run-time information to confirm that all previous accesses to this item have seen it as being of a specific primitive type. If that is the case, we specialize its representation and store the unboxed primitive value rather than the boxed version. TruffleBC applies this specialization to all data (local variables, array elements, and object fields) with a reference type. For local variables, we specialize the slots of the corresponding `Frame` object according to the types of values that are written to them. For *object fields*, we specialize the slots of the `DynamicObject` that is created whenever a class is instantiated. If a field or a variable is used for different primitive types and/or reference types, we revert the specialization and use type `Object` instead. This prevents re-specialization cycles and ensures that the optimization reaches a fixpoint. This strategy is identical to the one used by dynamic languages on top of Truffle [29]. Uninitialized `Frame` slots as well as uninitialized fields of a `DynamicObject` are marked, which allows us to return `null` if an uninitialized value is read.

We also optimize *arrays* whose element type is a reference type. The optimization is based on a simple storage strategy, similar to the approach of Bolz et al. [2] for dynamic languages. To minimize transitions between different storage strategies, we use *type mementos* [6] and record the types of all values that have been stored into an array object. This type information is fed back to the allocation site (in our case to the `anewarray` bytecode, Listing 8) of the array and if it indicates that all allocated arrays only contained primitive values of the same type, new array allocations at that site will use a primitive array instead of an `Object[]`. We apply this specialization per allocation site. Compiler optimizations like method inlining and splitting duplicate these allocation sites such that the specialization is performed locally for each replicated allocation site. For example, we aggressively inline the constructor of `ArrayList`, such that different `ArrayList` instances can specialize on different element types. If the Java program attempts to store a value with a different (possibly non-primitive) type into the array later on, the array will be reverted to an `Object[]`. We also deoptimize to `Object[]` if the user program attempts to store an explicitly boxed primitive value (e.g. `java.lang.Integer`), i.e., a boxed value that was explicitly allocated using the `new` keyword rather than by Java's autoboxing mechanism[2]. TruffleBC can alter the representation of arrays at runtime by wrapping an array into a `DynamicObject`. The `DynamicObject` wraps the array data (e.g., an `Object[]` or an `int[]` in the specialized case) including type information and the initialization state of all elements in order to be able to provide the same functionality as normal Java arrays (e.g., support for type checks). Again, to prevent re-specialization cycles, we pessimistically allocate `Object[]` once an array specialization was reverted.

TruffleBC ensures correct Java semantics by tracking whether the data has been initialized. For example, if TruffleBC specializes an `Object[]` to an `int[]`, it is necessary to keep track of uninitialized elements because reading an uninitialized array element must return `null` rather than an integer value 0.

4.2 Specialization of reference bytecodes

We also specialize reference bytecodes so that they can work with primitive values as well. We specialize such bytecodes to a version that uses the primitive types that we observe at run time. For example, if the element value of an `aastore` bytecode has the primitive type `int`, we specialize the instruction such that it can store the primitive `int` value directly into an array. If the behavior of the program changes during later execution in the sense that the value of the `aastore` instruction is not primitive anymore, we revert the bytecode specialization back to the generic case that can deal with both, reference types and primitive types. TruffleBC can specialize the following bytecodes to primitive types:

- Array accesses via `aastore` and `aaload`.
- Local variable accesses via `astore` and `aload`.
- Field accesses via `putfield` or `putstatic` and `getfield` or `getstatic`.
- The return instruction `areturn`.
- The type check via `checkcast` or `instanceof` treats primitive values like their boxed counterpart.
- The instruction `if_acmp` for performing a reference `comparison` [3]

[2]We distinguish between auto-boxing (implicit; Java inserts e.g. `java.lang.Integer.valueOf`) and an allocation of an object that boxes a primitive value (explicit; the Java developer writes e.g. `new java.lang.Integer`): We only specialize implicitly boxed primitives to primitive values. Our optimizations do not trigger for explicitly allocated boxed primitive values.

[3]Regarding the object identity of boxed primitives, the Java Language Specification discourages "[...] any assumptions about the identity of the boxed values on the programmer's part" [16]. The specification also states, that "[...] the value p being boxed is an integer literal of type int between -128 and 127 inclusive (§3.10.1), [...] then let a and b be the results of any two boxing conversions of p. It is always the case that a == b." [16], i.e. boxed integers between -128 and 127 are cached. Moreover, the specification mentions, that "The implementation may cache these [boxed primitive

- The `isnonnull` always returns `false` if the instruction is specialized on primitive values. Vice versa for `ifnull`.

4.3 Removing autoboxing and autounboxing

As a last step we remove the automatic boxing of primitive values. To remove autoboxing and autounboxing we substitute every function call to a boxing method (e.g. `Integer.valueOf`) by an identity function that directly returns the primitive argument value. Vice versa, if an unbox operation is performed on an object that has been specialized to a primitive value, we simply ignore the function call and continue with the primitive value. For example, an `invoke` bytecode, which would call an unboxing method (e.g. `Integer::intValue`) on a specialized primitive value, will be ignored by TruffleBC. If a method (e.g. `toString`) is invoked on a specialized primitive value, we lazily box this value and perform a regular method call.

After data specialization, bytecode specialization, and the removal of autoboxing/autounboxing, TruffleBC allows primitive values to flow through a program and to be used everywhere a reference is expected. These optimizations work nicely together in the `ArrayList` example of Listing 1; boxing/unboxing is completely removed, reference bytecodes specialize on primitive `int` values, and the `Object[]` of the `ArrayList` instance is specialized to an `int[]`.

5 EVALUATION

The main goal of the evaluation is to show that a statically-typed language like Java can benefit from optimizations originally conceived for dynamically-typed languages. For this, we focus on assessing whether specializing operations and optimizing data representation can improve the performance for code that uses Java generics.

TruffleBC cannot yet run large-scale benchmark suites like SPECjvm or DaCapo. Hence, we decided to use smaller benchmarks and focus the evaluation on programs that heavily use generics in performance-sensitive code where the boxing of primitives causes significant overhead. For a fair assessment, we further want to compare benchmarks that use generics with versions of the same benchmarks that use primitive arrays directly. Benchmarks that heavily use primitive arrays (e.g. the SciMark benchmarks) fit this requirement best. These benchmarks allow us to provide an alternative version where we replace the primitive arrays by Java's generic collections. With these two versions of each benchmark, we can evaluate whether dynamic optimizations improve peak performance of Java code that use generics.

Benchmarks. We use benchmarks from the SciMark benchmark suite,[4] benchmarks from the Are We Fast Yet suite [18] and others. More specifically, our benchmarks consist of a micro benchmark that uses a bubble-sort algorithm and larger benchmarks including a Fast Fourier Transformation (*FFT*), a dense LU matrix factorization (*LU*), array permutations (*Permute*, *Fannkuch*), an n-queens problem solver (*Queens*), a Jacobi successive overrelaxation (*SOR*), and a sparse matrix multiplication (*SparseMatMult*). We use two different versions of these benchmarks: the *ArraySuite* consists of the default implementations using primitive arrays; the *ListSuite* consists of modified versions of the benchmarks using `ArrayList`s instead of primitive arrays.

Experimental Setup. The benchmarks were executed on an Intel Core i7-4770 quad-core 3.4GHz CPU running 64 Bit Ubuntu 16.04.1 LTS with 16 GB of memory. We base the TruffleBC implementations on the Graal and Truffle version that is contained in the GraalVM 0.19 release. In this evaluation we compare three configurations

TruffleBC is the Truffle-based bytecode interpreter as described in this paper. However, in this configuration we do not specialize data or bytecodes, nor do we remove automatic boxing and unboxing. Also, in this configuration, TruffleBC does not use Truffle's `DynamicObject` for object allocations. It allocates regular Java objects and arrays and uses the `sun.misc.Unsafe` API to access members of an object.[5]

TruffleBC opt is the same Truffle-based bytecode interpreter as for TruffleBC. However, in this configuration we use Truffle's `DynamicObject` for object allocations, we specialize data (local variables, object members, and arrays), we specialize bytecode instructions, and also remove automatic boxing and unboxing.

HotSpot C2 is a regular Java HotSpot VM 1.8.0_92 using the server compiler.

Our benchmark harness reports execution time (lower is better) for each benchmark and its configuration. Whenever we report peak performance we executed the benchmark 1000 times with the same parameters to arrive at a stable peak performance. After these warm-up iterations (an iteration is a full run of a single benchmark), every benchmark has reached a steady state such that subsequent iterations are identically and independently distributed. This was verified informally using lag plots [17]. After warm-up, we executed every benchmark for another 1000 iterations and calculated the average using the arithmetic mean [11]. Where we report an error interval we show the standard deviation. Where we summarize across different benchmarks we report a geometric mean [11].

5.1 Micro benchmark

In the first part of our evaluation we use a micro benchmark, which sorts integer values using the bubble-sort algorithm. We evaluate two versions of this algorithm: first we sort an `int[]`, and then an `ArrayList<Integer>`. Figure 3 compares HotSpot C2, TruffleBC, and TruffleBC opt. We normalized the six configurations (HotSpot C2, TruffleBC, and TruffleBC opt running the array version as well as the list version) to the HotSpot C2 performance running the array version of the benchmark. The x-axis of the chart shows the different configurations; the y-axis shows the normalized average execution time (lower is better).

Replacing the `int[]` by `ArrayList<Integer>` makes the HotSpot C2 performance 4.5x slower and TruffleBC 4.2x slower. Using a generic class here introduces autoboxing/unboxing and stores the primitive values in an `Object[]` rather than an `int[]`,

values], lazily or eagerly" [16], which is what we are doing, caching all boxed primitive values.

[4] *SciMark 2.0*, Roldan Pozo and Bruce R Miller, access date: 2015, http://math.nist.gov/scimark2/index.html

[5] By investigating the IR of the Graal compiler we verified that object accesses using `sun.misc.Unsafe` get compiled to the same machine code operations as a regular Java object access.

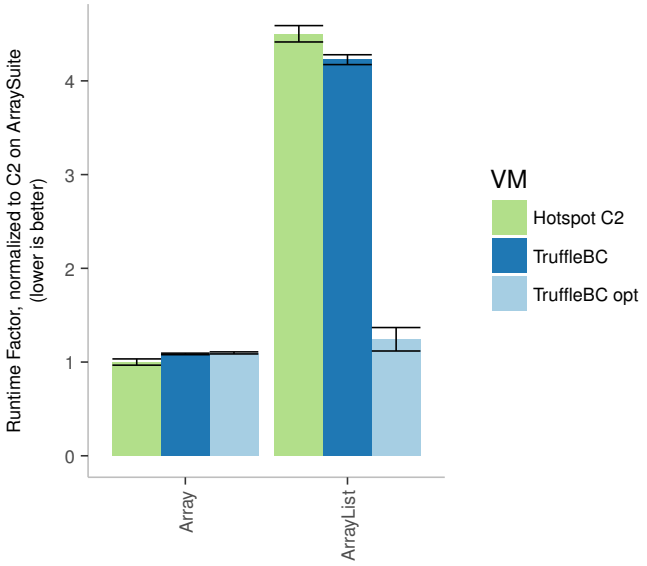

Figure 3: Micro benchmark (sorting an `int[]` and an `ArrayList<Integer>`) peak performance.

which causes a significant performance overhead. However, TruffleBC opt can remove this overhead almost completely and executes the version using an `ArrayList<Integer>` almost as fast as the version using a primitive `int[]`. TruffleBC opt is 3.6x faster than HotSpot C2 on the list version of the benchmark. The dynamic optimizations of TruffleBC remove boxing and unboxing, specialize the reference bytecodes to the primitive `int` type, and also ensure that the `ArrayList` is backed by an `int[]` rather than an `Object[]`. These performance numbers provide empirical evidence that the overhead, introduced by Java's generics implementation, can be removed by dynamic optimizations.

5.2 Peak Performance

In this evaluation we investigate the performance difference between TruffleBC and TruffleBC opt on a larger set of benchmarks. We also compare them to the peak performance of HotSpot C2. For that, we use the *ArraySuite* and the *ListSuite*; within each suite, we normalize numbers to HotSpot C2 performance. Figure 4 summarizes the individual benchmarks using box-plots; the bottom and the top of the box are the first and third quartiles, the band inside the box is the median. The × denotes the geometric mean. Figure 5 shows the individual benchmarks; the y-axis of the charts show the average execution time (lower is better) normalized to the HotSpot C2 performance.

Let us first consider the *ArraySuite*; on these benchmarks the dynamic optimizations do not trigger. TruffleBC opt is on average 8% slower than TruffleBC. Using `DynamicObject`s instead of regular Java objects or arrays introduces additional indirections when accessing data. The Graal compiler applies partial escape analysis with scalar replacement [26], which aims to remove most of these indirections. If an object does not escape the compilation scope and

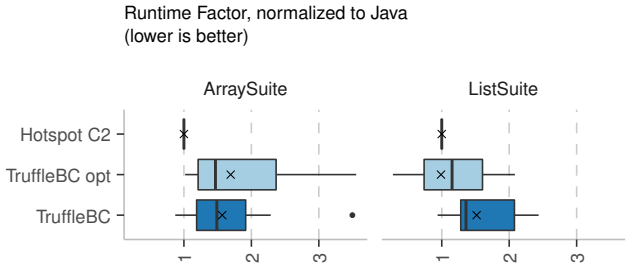

Figure 4: Peak performance benchmark evaluation summary (lower is better) of TruffleBC and T ruffleBC opt, relative to HotSpot C2.

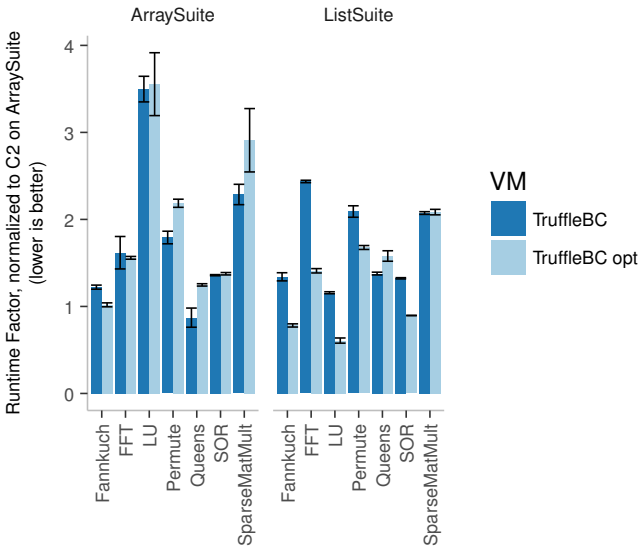

Figure 5: Peak performance benchmark evaluation (lower is better) of TruffleBC and TruffleBC opt, relative to HotSpot C2.

can hence be removed, there is no difference in performance (e.g. Fannkuch, FFT, LU, and SOR).

However, if we consider the *ListSuite* where all dynamic optimizations trigger, TruffleBC opt is 54% faster compared to the base version of TruffleBC. We measure the biggest speedup on benchmarks where data access contributes significantly to the performance of an application (e.g. Fannkuch, FFT, LU, Permute, SOR). On the other hand, we measure no performance difference between TruffleBC and TruffleBC opt on benchmarks where data access plays a minor role (e.g. Queens or SparsematMult).

The performance numbers indicate that dynamic optimizations come with a trade-off. In the evaluation's worst-case scenario (the ArraySuite), TruffleBC opt introduces an overhead of 8% compared to TruffleBC. However, in the evaluation's best-case scenario (the ListSuite), TruffleBC opt is 54% faster than TruffleBC.

In order to set our numbers in relation to an industry standard JVM we also compare TruffleBC and TruffleBC opt to HotSpot C2. Compared to HotSpot C2, TruffleBC is on average 57% (TruffleBC) and 69% (TruffleBC opt) slower on the benchmarks of the ArraySuite. A significant part of this overhead is caused by expensive method calls in Truffle, which are not yet optimized by the framework. Currently, Truffle method calls use `Object` arrays for parameter passing, which requires allocation and boxing and is thus more expensive than method calls in HotSpot C2. Work is under way to resolve this issue, and we expect the peak performance gap between TruffleBC and HotSpot C2 to be closed. This assumption is supported by benchmarks where Truffle inlines all methods of the benchmark into a single compilation unit (e.g. Fannkuch, Queens, or SOR). For these benchmarks, the performance overhead is 36% in the worst case.

If we consider the benchmarks of the ListSuite, TruffleBC (without dynamic optimizations) is slower on every individual benchmark compared to HotSpot C2. However, TruffleBC opt is able to outperform HotSpot C2 on Fannkuch (28%), LU (65%), and SOR (12%). On average, TruffleBC opt is 1% faster than HotSpot C2 on the ListSuite.

5.3 Warm-up Performance

Finally, we investigate the warm-up performance of TruffleBC and TruffleBC opt and analyze the impact of dynamic optimizations there. We are interested in the first 25 iterations because these iterations contain the specialization of bytecode instructions, the data specializations, and the elimination of autoboxing/unboxing. After 25 iterations our benchmarks are compiled and have reached their peak performance, which we informally verified using lag plots. The charts in Figure 6 show the warm-up of TruffleBC, TruffleBC opt, and HotSpot C2 for every individual benchmark. The x-axis shows the iterations. The y-axis shows the performance of the i-th iteration normalized to the peak performance (e.g., we use $\text{TruffleBC}_{\text{iteration=3}}/\text{TruffleBC}_{\text{peak}}$ to compute the third value). Data and bytecode specialization of TruffleBC opt introduce an additional overhead in the first iterations compared to TruffleBC. To quantify this additional overhead, we cumulate the relative overhead to peak performance within the first 25 iterations for each benchmark for all configurations (TruffleBC, TruffleBC opt, and HotSpot C2):

$$\text{Warmup}_{\text{bench}} = \frac{1}{25}\sum_{i=1}^{25}\frac{\text{TruffleBC}_{\text{iteration}=i}}{\text{TruffleBC}_{\text{peak}}}$$

	TruffleBC	TruffleBC opt	HotSpot C2
$\text{Warmup}_{\text{geom}}$	16.2x	20.5x	1.8x

Table 1: Average overhead to peak performance.

Table 1 summarizes the average overhead to peak performance on the first 25 iterations, which is caused by warm-up. Dynamic optimizations of TruffleBC opt come with a trade-off; the TruffleBC opt warm-up overhead in the first 25 iterations is 26% higher than that of TruffleBC. The Truffle framework currently focuses on peak performance; language implementations are built on a JVM, and Truffle trees start execution in HotSpot's interpreter, which increases the warm-up time. Also, the Graal compiler (used to compile Truffle trees to machine code) is designed to be a top-tier compiler optimizing for peak performance, which makes it comparably slow. Hence, HotSpot C2 (green line in Figure 6) has a significantly better start-up performance than TruffleBC. All benchmarks reach their peak performance within the first 5 iterations on HotSpot C2. For a more detailed discussion about Truffle warm-up performance, we would like to refer the reader to [19].

6 RELATED WORK

Applying optimization techniques for dynamically-typed languages to statically-typed languages has a long-standing tradition [1]. The just-in-time compilation technology for SELF [27] was the foundation for what eventually became the HotSpot JVM [21]. In the same vein, the work presented here explores how techniques that are currently used in JavaScript and Python VMs can be used in Java. Specifically, we rely on the work on SELF's maps [4], the Truffle object storage model [29], storage strategies [2], and mementos [6].

The Truffle object storage model provides the foundation for representing objects in TruffleBC in such a way that type-specialization of fields can be applied at run time. Variations of this technique and SELF's maps are widely used in JavaScript engines [5] and, for instance, in PyPy [3]. To the best of our knowledge, however, they have not been applied to statically-typed languages such as Java before. With Java's particular design interaction of primitives and generics, Java is a language where type parameters are checked at compile time, but are absent at run time, which leaves room for dynamic optimizations.

In addition to optimizing objects, collections are of great interest, because collections are the area where Java's generics are widely used. Storage strategies for collections provide the foundation for optimizations and have been explored for Python in PyPy [2]. They also have been applied to JavaScript, for instance in V8, where they are used in combination with type-feedback to the allocation site [6] in order to achieve optimal performance.

Approaches such as Miniboxing [28] use compiler optimizations and additional type information to generate specialized representations that encode generic data structures without boxing. However, since this is a static approach it is less flexible than our dynamic approach proposed here. Furthermore, Scala uses a `@miniboxed` annotation since the semantics are not transparent.

Work such as [33] and Chameleon [23] is more specifically targeted towards collections and provides either online or offline means to optimize them more for operations on collections in general, and thus goes beyond the type specializations demonstrated here. However, such more advanced optimizations and also the notion of just-in-time data structures [8] could be combined with the optimizations presented here.

Another aspect to be considered in our optimizations is, for instance, thread safety [7]. However, our work focused first on the performance potential for single threaded code, and we will investigate thread safety in future work.

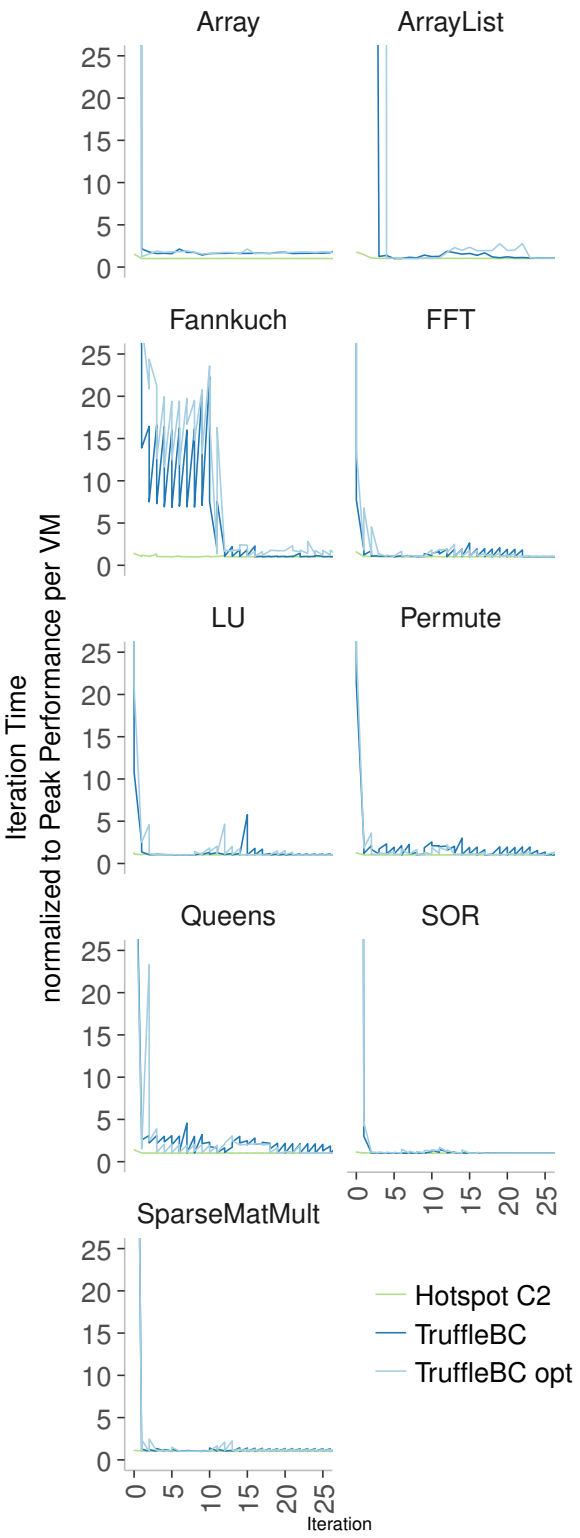

Figure 6: Warm-up plots of ListSuite benchmarks on the first 25 iterations.

7 CONCLUSION

In this paper we presented TruffleBC, a self-optimizing Truffle interpreter, which applies dynamic optimizations to Java bytecode. We applied dynamic compilation techniques designed for dynamically-typed languages and showed that they can be successfully used for Java, e.g., to implement Java's generics more efficiently. These optimizations include the specialization of reference bytecode instructions to primitive types, data specialization, and the removal of automatic boxing and unboxing.

We have seen that dynamic optimizations make the interpreter more complicated. Data structures as well as the implementation of bytecode instructions must be more flexible such that they can be adapted at run time, which increases the pressure on the dynamic compiler. Whenever the dynamic compiler can optimize and remove this additional complexity, we do not see a performance difference between a TruffleBC version without dynamic optimization and a TruffleBC version with dynamic optimizations. Overall, we measured a performance overhead of 8% of TruffleBC with dynamic optimizations on benchmarks where these optimizations did not trigger. However, on benchmarks where the dynamic optimizations trigger and data and bytecode can be specialized for primitive types, we measure an average speedup of 54%. On these benchmarks, TruffleBC also outperforms HotSpot C2 by 1% on average and is up to 65% faster in certain cases.

From our work we can conclude that dynamic language implementation frameworks like Truffle can be used to efficiently implement statically-typed languages like Java. Based on this implementation, we successfully demonstrated that Java programs using flexible typing (e.g., generic data types or the compatibility between primitive data types and `Object`) can be effectively optimized using compiler techniques that were originally designed for dynamically-typed languages.

ACKNOWLEDGMENTS

We thank all members of the Virtual Machine Research Group at Oracle Labs and the Institute of System Software at the Johannes Kepler University Linz for their valuable feedback on this work and on this paper. Oracle, Java, and HotSpot are trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

REFERENCES

[1] John Aycock. 2003. A Brief History of Just-In-Time. *ACM Comput. Surv.* 35, 2 (June 2003), 97–113. https://doi.org/10.1145/857076.857077

[2] Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. 2013. Storage Strategies for Collections in Dynamically Typed Languages. In *Proc. OOPSLA*. 167–182. https://doi.org/10.1145/2509136.2509531

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

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

[5] Maxime Chevalier-Boisvert and Marc Feeley. 2016. Interprocedural Type Specialization of JavaScript Programs Without Type Analysis. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 56. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 7:1–7:24. https://doi.org/10.4230/LIPIcs.ECOOP.2016.7

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

[7] Benoit Daloze, Stefan Marr, Daniele Bonetta, and Hanspeter Mössenböck. 2016. Efficient and Thread-Safe Objects for Dynamically-Typed Languages. In *Proceedings of the 2016 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '16)*. ACM, 18.

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

[9] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Speculation Without Regret: Reducing Deoptimization Meta-data in the Graal Compiler. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '14)*. ACM, 187–193. https://doi.org/10.1145/2647508.2647521

[10] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages (VMIL '13)*. ACM, 1–10. https://doi.org/10.1145/2542142.2542143

[11] Philip J. Fleming and John J. Wallace. 1986. How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results. *Commun. ACM* 29, 3 (March 1986), 218–221. https://doi.org/10.1145/5666.5673

[12] Matthias Grimmer, Manuel Rigger, Lukas Stadler, Roland Schatz, and Hanspeter Mössenböck. 2013. An Efficient Native Function Interface for Java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '13)*. ACM, 35–44. https://doi.org/10.1145/2500828.2500832

[13] Matthias Grimmer, Chris Seaton, Roland Schatz, Würthinger, and Hanspeter Mössenböck. 2015. High-Performance Cross-Language Interoperability in a Multi-Language Runtime. In *Proceedings of the 11th Symposium on Dynamic Languages (DLS '15)*. ACM.

[14] Matthias Grimmer, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. 2015. Dynamically Composing Languages in a Modular Way: Supporting C Extensions for Dynamic Languages. In *Proceedings of the 14th International Conference on Modularity (MODULARITY 2015)*. ACM, 1–13. https://doi.org/10.1145/2724525.2728790

[15] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI '92)*. ACM, 32–43. https://doi.org/10.1145/143095.143114

[16] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. 2000. The Java language specification. (2000), 768 pages.

[17] Tomas Kalibera and Richard Jones. 2013. Rigorous Benchmarking in Reasonable Time. In *Proceedings of the 2013 ACM SIGPLAN International Symposium on Memory Management (ISMM)*.

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

[19] Stefan Marr and Stephane Ducasse. [n. d.]. Tracing vs. Partial Evaluation: Comparing Meta-Compilation Approaches for Self-Optimizing Interpreters. In *Proceedings of the 2015 ACM International Conference on Object Oriented Programming Systems Languages; Applications (OOPSLA '15)*. ACM.

[20] Jerome Miecznikowski and Laurie Hendren. 2002. Decompiling Java bytecode: Problems, traps and pitfalls. In *International Conference on Compiler Construction*. Springer, 111–127.

[21] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java Hotspot(TM) Server Compiler. In *JVM'01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*. USENIX Association, 12.

[22] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Bringing Low-Level Languages to the JVM: Efficient Execution of LLVM IR on Truffle. In *Proceedings of Workshop on Virtual Machines and Intermediate Languages (VMIL '16)*. 10. https://doi.org/10.1145/2998415.2998416

[23] Ohad Shacham, Martin Vechev, and Eran Yahav. 2009. Chameleon: Adaptive Selection of Collections. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*, Vol. 44. 408–418. https://doi.org/10.1145/1543135.1542522

[24] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, and Thomas Würthinger. 2012. Compilation Queuing and Graph Caching for Dynamic Compilers. In *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages (VMIL '12)*. ACM, 49–58. https://doi.org/10.1145/2414740.2414750

[25] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, Thomas Würthinger, and Doug Simon. 2013. An Experimental Study of the Influence of Dynamic Compiler Optimizations on Scala Performance. In *Proceedings of the 4th Workshop on Scala (SCALA '13)*. ACM, Article 9, 8 pages. https://doi.org/10.1145/2489837.2489846

[26] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, Article 165, 10 pages. https://doi.org/10.1145/2544137.2544157

[27] David Ungar and Randall B. Smith. 1987. Self: The Power of Simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications (OOPSLA '87)*. ACM, 227–242. https://doi.org/10.1145/38765.38828

[28] Vlad Ureche, Cristian Talau, and Martin Odersky. 2013. Miniboxing: Improving the Speed to Code Size Tradeoff in Parametric Polymorphism Translations. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, 73–92. https://doi.org/10.1145/2509136.2509537

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

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

[31] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. 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! 2013)*. ACM, 187–204. https://doi.org/10.1145/2509578.2509581

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

[33] Guoqing Xu. 2013. CoCo: Sound and Adaptive Replacement of Java Collections. In *ECOOP 2013 – Object-Oriented Programming: 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, Giuseppe Castagna (Ed.). Springer, 1–26. https://doi.org/10.1007/978-3-642-39038-8_1