École Polytechnique Fédérale de Lausanne

Optimizing Java on Truffle

by Katja Goltsova

Master Thesis

Prof. Viktor Kunčak
Thesis Advisor

Alfonso[2] Peterssen
Thesis Supervisor

Acknowledgments

First of all, I would like to thank my supervisor, Alfonso[2] Peterssen, for guiding me through this project, encriching me with a lot of compiler and JVM knowledge, always being ready to discuss, debug, and help, and not the least, for his enthusiasm — always an inspiration. My deepest thanks goes to Prof. Viktor Kunčak, for his attention to this project, patience to go through all the Java layers, and ideas to try.

A big thank you to the Espresso team: Thomas Garcia, Danilo Ansaloni and Allan Gregersen, for impromptu debugging sessions, explanations, and welcoming me to the team. To Gilles Duboscq, Christian Humer and Boris Spasojević, for their expertise and eagerness to share it.

I am deeply grateful to my parents for their warm support, to Matthieu for Scala, Java and life discussions, to Harshit, for being by my side, to Guillaume, for always finding something to cook a lunch out of. To Zofia, Jigyasa, and Asma, for dinners, walks, and laughs. You made these six, and all other months of my life much better.

Finally, thanks to Tushar for making sure I never miss a workshop, to Véronique for kind words and wordless dances, to Laure for her warmth, to Irina for swingout discussions, out in the cold, until 11pm; to Mikael, Keyne, Anne, Lucie, Dominique, Benoit, Adrien, Pascale and everyone I shared a dance with, for the wonderful community and a shared passion.

Katja Goltsova

Abstract

Java on Truffle is an early-stage implementation of a Java Virtual Machine in Java. So far its development has focused on compatibility and functionality, not addressing performance in a systematic way. This thesis presents a series of experiments on Java on Truffle performance, namely adding Class Hierarchy Analysis, improving receiver profiling at callsites of virtual and interface methods, splitting methods per-callsite, delaying the collection of profiling information, and investigating and improving `System.arraycopy` performance.

Contents

Chapter 1

Introduction

Many programming languages exist and are still created today. Building a language is a hard project: the developer needs to create many complex parts, including compiler and memory management. It takes years to write a compiler that can compete in performance with existing ones.

GraalVM [15] is a project which enables easier development of languages. Its key part is the highly optimizing Graal compiler, which can integrate with the HotSpot VM. This compiler can be used as a just-in-time (JIT) compiler for a JVM or be ahead-of-time compiled and run natively. The Graal compiler also provides a foundation for Truffle framework for implementing language interpreters.

The Truffle framework with support of the Graal compiler allows the developers to write interpreters that perform at a speed comparable to state-of-the-art compilers. For instance, GraalJS [5] performance is comparable [4] with V8 [13] performance. R, Ruby, Python, Java have Truffle-based implementations.

Java on Truffle is a spec-compliant implementation of a Java Virtual Machine. Its bytecode interpreter is implemented using the Truffle framework. Java on Truffle does not implement its own memory management, relying on the host garbage collector. Thanks to Truffle, the implementation is relatively simple, hence flexible. This allows adding advanced features for which HotSpot is too complicated, for example rich class redefinition capabilities.

Java on Truffle is an early-stage implementation, its performance has not been an area of focus. This thesis is a first attempt at structured performance improvement. As a part of this work, we performed a series of experiments with which we attempted to improve the performance.

We started by designing and implementing an opt-in Class Hierarchy Analysis to Java on Truffle. Class hierarchy in Java is dynamic, changing whenever a previously unobserved class is used, which triggers this class's loading. We track whether each class has zero, exactly one, or multiple

concrete implementors. We leveraged the existing implementation to track whether a given method has been overridden. This information is used to optimize `instanceof`, `invokeinterface` and `invokevirtual` instructions: for example, if class `C` has a single concrete implementor, `a instanceof C` if and only if `a.class == C.implementor`. This work has more applications in the (currently under development) ahead-of-time mode that are subsumed by invoke caches in the just-in-time mode. A notable challenge was to reconcile Class Hierarchy Analysis with class redefinition feature.

We then implemented several ideas of improving the profiling at callsites of `invokeinterface` and `invokevirtual`. The implementation of `invokeinterface` and `invokevirtual` caches the class and method resolution result of first few observed receivers to avoid doing costly method resolution on every call. We refer to this mapping from class to resolution result as the invoke cache. These caches are represented as linked lists, where entries are added at head. Suspected imperfections were duplicates in the cache due to different classes' implementations of a method resolving to the same method, having rarely executed entries take up space in the cache, and having the cache entries in an order, significantly different from frequency order, increasing cache traversal times. Integrating these prototypes into the codebase in a clean and composable way is a considerable engineering challenge and has not been done as a part of this work.

We followed by implementing method splitting for Java on Truffle. Originally, all invocations of a method go through the same call target. This means that the call target's profiling combines the information from all its invocations at very different callsites. It is sometimes beneficial to specialize a call target to a callsite, i.e. create a copy of that call target and collect profiling information for that callsite only. However, this comes with a cost of initializing and warming up the copy. We see this effect in benchmark results: some benefit from splitting due to specialization while others observe a regression due to the costs. This feature is scheduled to be merged to master (blocked by a bug in another component) but disabled by default with an option to enable by passing a flag.

Next we experimented with late profiling. This change is directed at reducing warmup costs for methods that are only called a handful of times throughout the execution of a program. At method callsite, Java on Truffle collects profiling information about the call. Application startup behaviour (in particular, VM startup) might not be representative of the rest of its execution. In addition, profiling information takes time to collect and memory to store, which does not pay off if a method is only ever called once or twice. This change replaced all the profiling logic at the callsite by a simple indirect call. We observed a regression rather than an improvement and discarded this change.

We followed by comparing Java on Truffle performance on jmh microbenchmarks to the HotSpot performance. One of important differences was the performance of `System.arraycopy` implementation. This is an important case because it underlies not only array copying, but also string operations. For performance reasons, instead of simply executing `System.arraycopy` bytecodes, Java on Truffle provides a *substitution* — its own implementation of `arraycopy`.

Investigating its performance, we fixed a bug in the substitution branch profiler, rewrote the substitution method as a node and enabled substitution splitting, which permitted the `arraycopy` code to specialize to each callsite. This brought an `arraycopy` performance improvement of 1.5x. These changes are merged.

The rest of this thesis is organized as follows. First we discuss the necessary background, including the Graal compiler, Truffle framework for language implementation, and Java on Truffle. Then we repeat the performance problem that this thesis tried to solve. Afterwards we discuss our attempts to improve performance, their effect and limitations. We conclude with future work on performance.

Chapter 2

Background

2.1 Terminology

This thesis discusses implementation of interpreters. Whenever we talk about code, we might refer either to the application code which is run on the interpreter, or the code that constitutes the interpreter implementation. We refer to the former as *guest code*, and to the latter as implementation, or *host* code. In particular, we are describing Java on Truffle, which includes a Java bytecode interpreter, written in Java. We refer to the application as *guest Java* and the interprerter as *host Java*.

2.2 Architecture overview

Figure 2.1 gives an overview of Graal infrastructure when running in JVM mode. Alternative running mode is discussed in section 2.3.

The Graal compiler integrates with the HotSpot VM as the JIT compiler. This allows running Java and other JVM languages such as Scala, Kotlin, Clojure. In addition, the Truffle language implementation framework enables developers to write their own implementation of an arbitrary programming language in Java, and through the integration between Truffle and the Graal compiler run it on JVM. Oracle Labs develops Truffle-based implementations of JavaScript (GraalJS), Ruby (TruffleRuby [12]), R (FastR [2]), Python (GraalPython [3]), WebAssembly (GraalWasm [6]), LLVM (SuLong [16]), and Java (bytecode). Third-party language implementations include grCuda, SOMns, TruffleSqueak [10], TruffleWasm [11], and Yona. Truffle is discussed in more detail in section 2.4. In particular, Java on Truffle implementation is described in section 2.5.

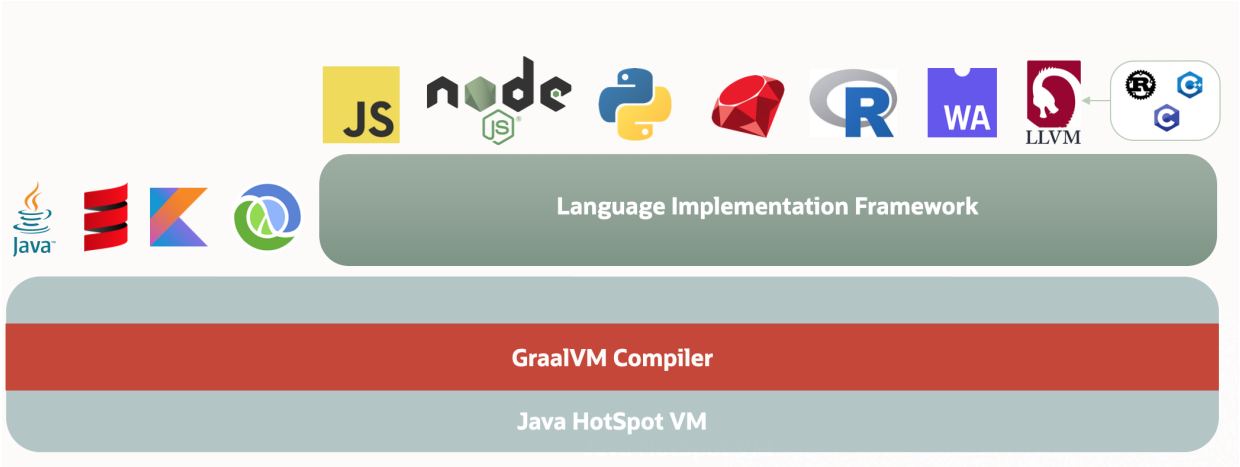

Figure 2.1: GraalVM architecture

2.3 Native image

There are three ways to run Java code in Graal ecosystem: first, on a JVM, plugging the Graal compiler as the just-in-time compiler. Second, natively. Third, executing its bytecodes with Java on Truffle. In this section we introduce the second.

The Graal ecosystem provides a way to compile Java code ahead-of-time into a standalone executable called *native image* of the original code. To distinguish between the ahead-of-time compilation that produces a native image and the just-in-time compilation by the Graal compiler, we call the former *building* a native image.

A native image of a Java application is built by ahead-of-time compiling the application, JDK, and Substrate VM. Substrate VM is the runtime system for native image that includes, for example, garbage collector and thread scheduler.

The Graal compiler is included in the native image for Truffle languages. As the application is running, its code is getting JIT-compiled, taking advantage of runtime profiling information. A deoptimizer to transfer from JIT-compiled to ahead-of-time compiled (the equivalent of the interpreted code on JVM) is included in Substrate VM.

Java is designed to execute in an evolving environment, with new classes being accessed and consequently loaded at runtime. This contradicts the idea of ahead-of-time (AOT) compilation: all classes that are accessed during runtime must be known in advance to be compiled. This is the *closed world assumption* made by the AOT compilation. It limits some Java features, for example reflection: all classes accessed through reflection must be specified in advance. To detect the classes that should be included to the image, the native image builder performs a static analysis prior to the AOT compilation.

First iterations of a native image run faster than first iterations of the same code, running on JVM, because native image is running compiled code immediately (even though it lacks profiling information and can be improved).

2.4 Truffle

Truffle [14] provides a set of building blocks that enable the programmer to write an interpreter which will be optimized by the Graal compiler. We describe most relevant such blocks in this section.

First, the developer defines subclasses of `Node` that correspond to the language constructs. For example, GraalPython has, among others, `IfNode`, `AssertNode`, `AssignmentNode`. In particular, subclasses of `AssignmentNode` correspond to functions and must define an `execute(VirtualFrame)`. `VirtualFrame` is Truffle's abstraction for a stack frame.

Each node that represents an operation must define a method (with arbitrary signature) to execute the node and get a result. To optimize, the developer can provide several *specializations* of this method that speculate on argument types or are guarded with boolean predicates on arguments.

Consider an example where the interpreter benefits from specializations. Consider a language that has a + operator, defined as concatenation on strings and as addition on integers. Say integers are represented in its interpreter as primitive Java `int`, and guest strings as `GuestString` type, which is a wrapper around a Java string (see `GuestString` stub on listing 2.1).

```
class GuestString {
  private final String underlying;
  public String getRawString() {
    return underlying;
  }

  // <..> implementation

  public static GuestString create(String underlying) {
    // <..> implementation
  }
}
```

Listing 2.1: Example representation of guest strings

Consider the implementation of + which specializes on the arguments type, provided on listing 2.2 (methods annotated with `@Specialization` represent a specialization of `execute`). From this code, Truffle automatically generates a concrete child of `AddNode` with an override of the `execute` method which checks if any of the specializations is applicable for the arguments, casts

the arguments to the appropriate type and calls the appropriate specialization. A specialization that was called at least once is *activated*. For each execution of a node, the generated code first traverses the activated specializations in activation order, checking if it can be applied to the current arguments. If no activated specialization applies, inactive specializations are checked in the order of declaration.

```
abstract class AddNode {
  public abstract Object execute(Object first, Object second);

  @Specialization
  public int executeInt(int first, int second) {
    return first + second;
  }

  @Specialization
  public GuestString executeString(GuestString first, GuestString second)
      {
    return GuestString.create(first.getRawString() + second.getRawString
        ());
  }
}
```

Listing 2.2: Example implementation of +

Let us illustrate further ways to specialize supported by Truffle. First, a specialization can assume not only types of the arguments, but also a boolean predicate on them, which is called a *guard*. For example, say that in the example language, addition of `int` and an empty string is defined and its result is the string representation of the integer. This behaviour can be implemented with the specialization on listing 2.3.

```
@Specialization(guards = "stringArg.isEmpty()")
public String executeIntPlusString(int intArg, GuestString stringArg) {
  return GuestString.create(Integer.toString(intArg));
}
```

Listing 2.3: Example implementation of int + empty string

Moreover, a specialization can be guarded by a global boolean predicate, which in Truffle is called an *assumption*. Since an assumption is a global predicate, synchronized between application threads if there are multiple, it would be inefficient to allow changing its value back and forth. Instead, the assumption must start as valid (true) and can be invalidated during the execution. All the compiled code guarded by the invalidated assumption is automatically invalidated.

Consider an assumption use case for the addition node above. Say the example language allows redefining operators, in particular +. To account for that, add an `isDefaultAddition` assumption that is invalidated when a user-defined implementation of + is observed. The specializations defined so far are only valid for the default behaviour of +, hence they must be

guarded with `isDefaultAddition`. As soon as the user defines a custom implementation of +, the assumption is invalidated and the specializations guarded by it are automatically invalidated. It is equivalent to guarding them with `false`, and since an assumption cannot become valid once it has been invalidated, these specializations will never be used again.

The last building block that we would like to describe is the `CompilationFinal` annotation. Compilers can aggressively optimize final fields, but not everything can be defined as final. Truffle provides an intermediate step: values that can change in the interpereter code but never in the compiled code. By annotation a field with `@CompilationFinal`, the developer lets the Graal compiler consider it as final, and perform the applicable optimizations.

2.4.1 Graal compilation

The compilation unit is a guest method. A method is added to the compile queue once its call count + the number of executed loop iterations inside surpasses a threshold. Several actions are associated with compilation, these actions are outlined below.

First, the interpreter nodes that execute the method are parsed into the Graal intermediate representation. This IR is *partially evaluated*, which means considering all final and CompilationFinal fields, as well as Assumptions, as constant, and performing constant folding. During this constant folding, the sizes (in Graal IR nodes) of the methods that are invoked by the method under consideration are computed; based on these sizes and the available budget, the inliner takes inlining decisions.

Finally, the method is compiled, and further invocations of this method will be executing the compiled code rather than the interpreter code. The switch back happens either through explicit invocation of `transferToInterpreter`, which continues execution of the current method invocation in the interpreter, or `transferToInterpreterAndInvalidate`, which in addition invalidates the compiled code, or when a compiler assumption no longer holds.

2.5 Java on Truffle

Java on Truffle is a Java Virtual Machine and implements all the components of the spec: a '.class' file parser, a bytecode verifier, a bytecode interpreter with data structures to represent guest code elements, the JVM native interface, the Java Debug Wire Protocol, etc. However, since most performance questions stem from the implementation of the interpreter, in this section we describe its architecture without discussing the details of other parts of the VM.

Unlike most of the other Truffle languages, Java on Truffle's interpeter ASTs are relatively flat. Exactly one node corresponds to a guest method. Depending on the method, this node can be

one of three types:

1. Most common case: bytecode node. These nodes correspond to guest Java methods which are executed by executing their bytecodes.
2. Native method node. These nodes correspond to guest Java methods which are marked as `native` and therefore have no bytecodes.
3. Substitution node. For some guest methods Java on Truffle defines an alternative implementation rather than executing their bytecodes. This implementation is called a *substitution*.

We continue by discussing the implementation of the bytecode node. Native method nodes are outside of the scope of this work. Substitution nodes are discussed in section 4.7.

Table 2.1: Expensive Java bytecodes

Bytecode	Other bytes	Stack effect	Semantics
`instanceof`	index	objectRef → result	check if the object on top of the stack is an instance of the type, identified by the `index` in the constant pool, put the result on the stack
`checkcast`	index	objectRef → objectRef	check if the object on top of the stack is an instance of the type, identified by the `index` in the constant pool, throw if not
`invokestatic`	index	arg1 arg2 ... → result	invoke the static method, identified by `index`, put the result on the stack
`invokespecial`	index	receiverRef arg1 arg2 ... → result	invoke the instance method, identified by `index`, on receiverRef, put the result on the stack
`invokeinterface`	index argCount 0	receiverRef arg1 arg2 ... → result	invoke the interface method, whose declaration is at `index` in the constant pool, on receiverRef, put the result on the stack
`invokevirtual`	index	receiverRef arg1 arg2 ... → result	invoke the virtual method, whose declaration is at `index` in the constant pool, on receiverRef, put the result on the stack

2.5.1 Bytecode execution

Bytecode node's central part is the interpreter loop, which iterates over the bytecodes of the method, executing them and jumping to the next bytecode according to the control flow. Most bytecodes are straightforward to execute, those are implemented directly in the interpreter loop.

Examples of such bytecodes are loads and stores from/to the stack, loads and stores from/to locals, loads and stores from/to arrays. These bytecodes are (1) cheap to execute (2) implemented in a straightforward manner, improving which might not be possible. They were not considered for improvement in this thesis.

`instanceof`, `checkcast` and `invokes` (`invokestatic`, `invokespecial`, `invokevirtual`, `invokeinterface`) are expensive. Their semantics is summarized in the table 2.1. To improve their performance, these bytecodes are implemented as nodes. On the first execution of one of these bytecodes, an instance of the corresponding node is created and stored in the array of *quick* nodes in the bytecode node. The bytecode is overwritten to refer to the created node. We call this operation *bytecode quickening*.

Let us go through an example of quickening: consider the Java code in listing 2.4 and its bytecode 2.5. In the bytecode representation of the `setup` method, `#2` , `#4`, `#5` are references to the constant pool, they are expanded in the comments.

```
interface CallRegister {
  void register();
}

interface SetupRegister extends CallRegister {}

interface Countable {
  static void count(Object o) {
    // <..>
  }
}

class Impl implements Countable, SetupRegister {
  // <..>
}

public class Example {
  static void setup(CallRegister o) {
    if (o instanceof SetupRegister) {
      o.register();
    }
    if (o instanceof Countable) {
      Countable.count(o);
    }
  }
}
```

Listing 2.4: Example Java code

```
static void setup(CallRegister);
  Code:
     0: aload_0                 // load local variable 0, i.e. 1st
```

```
    argument
 1: instanceof    #2         // SetupRegister
 4: ifeq          13         // if instanceof returned 0, branch to 13
 7: aload_0
 8: invokeinterface #3,  1   // InterfaceMethod CallRegister.register
    :()V
13: aload_0
14: instanceof    #4         // Countable
17: ifeq          24         // if instanceof returned 0, branch to 24
20: aload_0
21: invokestatic  #5         // InterfaceMethod Countable.count:(Ljava
   /lang/Object;)V
24: return
```

Listing 2.5: Bytecode for listing 2.4

Assume `setup` is executed for the first time with an argument which is both `SetupRegister` and `Countable`. The modified bytecode of the method is depicted on listing 2.6. `@n` denotes the position of the corresponding node in the array of quick nodes. Constant pool references are overwritten by the quick nodes indexes, but the corresponding information (class for `instanceof` and `checkcast`, method resolution seed for invokes) is stored in the quick nodes themselves.

```
static void setup(CallRegister);
  Code:
    0: aload_0
    1: quick         @0     // @0: InstanceOf(SetupRegister)
    4: ifeq          13
    7: aload_0
    8: quick         @1,  1 // @1: InvokeInterface(CallRegister.register
       )
   13: aload_0
   14: quick         @2     // @2: InstanceOf(Countable)
   17: ifeq          24
   20: aload_0
   21: quick         @3     // @3: InvokeStatic(Countable.count)
   24: return
```

Listing 2.6: Updated bytecode of listing 2.5 after the first execution

Figure 2.2 illustrates the resulting AST of the method. Unlike in dynamic Truffle languages, this AST is very flat.

2.5.2 Method invocation

Let us discuss how (guest) method invocation is implemented, using the example above. First, Java on Truffle needs to find which method to invoke. For `invokestatic` the constant pool entry

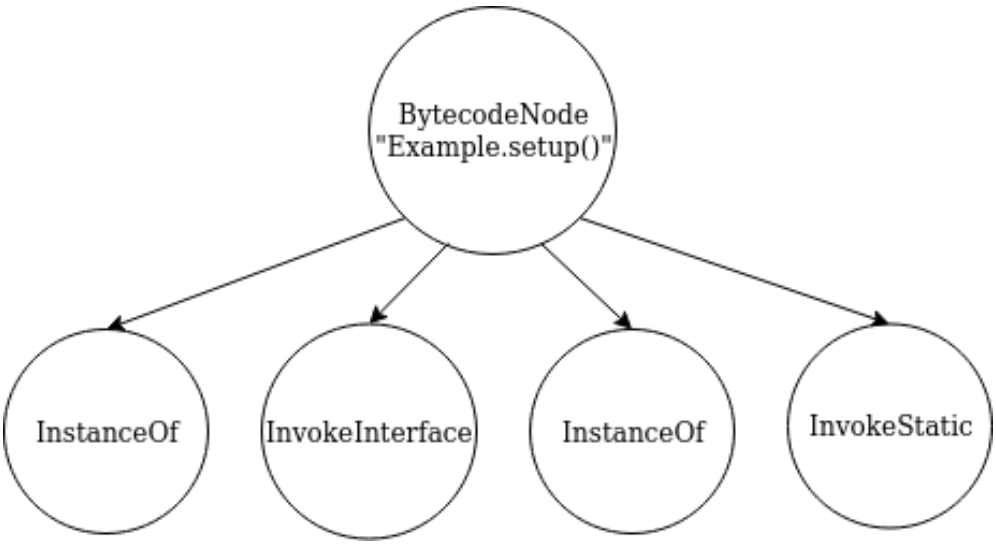

Figure 2.2: The AST of `Example.setup()` in the Java on Truffle interpreter

points to the method to execute; for `invokespecial` the target method is fully determined by the constant pool entry. However, for `invokevirtual` and `invokeinterface` the target method depends on the receiver type and must be resolved on each invocation.

Method resolution produces a method instance. Method instance contains its call target that has a one-to-one correspondence with a root node. As discussed in section 2.4, root node is executed, producing the method call result. This is summarized on figure 2.3.

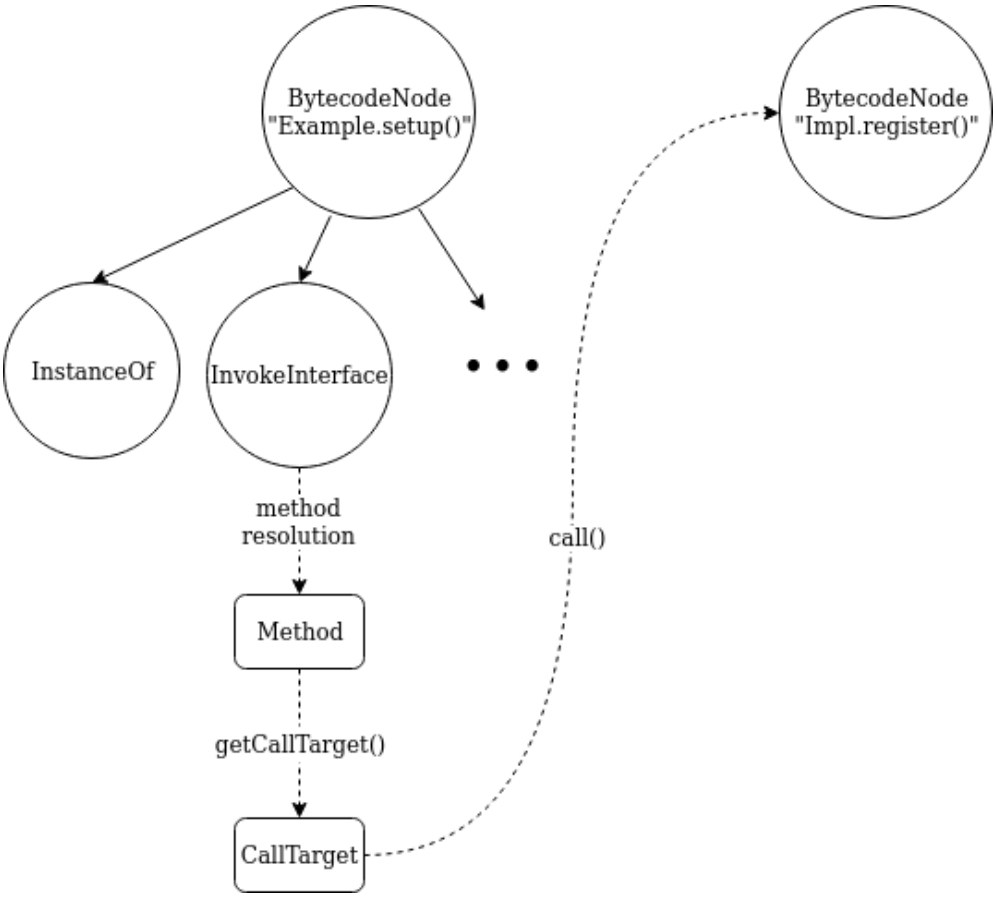

Figure 2.3: Invocation of `Impl.register()` from `Example.setup()` as seen by Java on Truffle

Chapter 3

Problem

Java on Truffle's development so far has been focused on reaching full spec compliance and implementing features that make it stand out. While the language implementation has been designed with performance in mind, it has not been specifically investigated. As far as performance is concerned, the focus has been on avoiding regressions. This thesis's goal is to better understand Java on Truffle performance by exploring several ideas of its improvement.

Chapter 4

Experiments

4.1 Benchmarks

To evaluate the impact of our changes, we measured Java on Truffle's performance with and without the change on dacapo and scala-dacapo benchmark suites. Whenever feasible, we wrote a microbenchmark that illustrated the effect that we expected from the change; when such a microbenchmark is missing, we explain why we did not believe it possible to show the intended behaviour on a tiny example. In this section we give an overview of the microbenchmarking infrastructure and the macrobenchmark suites.

We used the Java Microbenchmark Harness (JMH) [8] to develop and run change-specific microbenchmarks, using Java on Truffle as the JVM. JMH has a utility method to prevent dead code elimination. To make sure invocations of this method are not optimized away by the Graal compiler, we added a special handling of it in Java on Truffle.

We used dacapo and scala-dacapo benchmark suites to evaluate the impact of our changes. These suites consist of relatively big benchmarks where one iteration runtime varies from a few seconds to a few dozen seconds. These benchmarks are multi-threaded. Originally we tried using are-we-fast-yet (AWFY) benchmark suite to measure the performance, but they turned out to have too high variance between runs that was comparable to the change impact even when averaging over 5 runs.

4.2 Evaluation

The benchmarks were executed on Lenovo-P53 running Ubuntu 21.04, with Intel Core i7-9850H processor, 6 cores, and 32 GB RAM. Most of evaluations were run as native image rather than on HotSpot because of faster startup and more stable runs, as discussed in 2.3. JMH benchmarks

were run in the average time mode with 10 seconds per iteration, with 5 warmup iterations and 5 measurement iterations. For macrobenchmarks, one iteration equals to one run of the benchmark, the number of iterations varies benchmark-to-benchmark. The reported peak performance numbers were calculated over last 30% iterations. Unless specified otherwise, numbers for JMH are the average of 5 runs, for dacapo and scala-dacapo – of 3 runs.

There is a significant variance between different benchmark runs, especially pronounced for macrobenchmarks. It is caused by multiple sources of nondeterminism: methods get added to the compilation queue in different order, which causes different inlining decision due to both inlining nondeterminism and different evolution of budget. Garbage collection is triggered at different points in time. We attempt to mitigate these effects and obtain reliable performance results by averaging across runs.

For all benchmarks, we normalize the speedup to master (master is always 1, if the benchmark speedup after the change is less than one, it was slowed down; if the speedup is higher than one, the benchmark runtime was improved).

4.3 Class Hierarchy Analysis

4.3.1 Background

Class Hierarchy Analysis was first discussed by Jeffrey Dean et al. in 1995 [1]. Their idea was to track the class inheritance in the application and, combining it with analysis of type propagation, compute the set of methods that a given virtual call can invoke. If this set is small enough, the call can be devirtualized directly.

4.3.2 Class Hierarchy Analysis in Java on Truffle

Java on Truffle does not track the type flow in the guest program, hence we focused on tracking the inheritance information. The type hierarchy in Java is dynamic due to class loading: classes are loaded when they are first used rather than in advance. In our implementation, we tracked which types in the hierarchy have no concrete subclasses (including itself) – we call these types *leaves* – and which types have exactly one concrete subclass.

Moreover, prior to this work Java on Truffle already tracked which *methods* are leaves in the hierarchy, i.e. are concrete (not abstract) and have not been overridden. Notice that such leaf methods are not necessarily defined in a leaf class, it is sufficient for them to not have overrides in any of the subclasses of their defining class.

4.3.3 Applications

We used this hierarchy information to optimize `instanceof` checks: if a class has no concrete subclasses, no object can be an instance of it. If the potential supertype in the `instanceof` has exactly one concrete subclass, the subclassing check between the class of the object and this potential supertype can be replaced by the equality check on the class of the object and the concrete subclass of the supertype. To illustrate the same point with code, if class (or interface) `ImplementedOnce` has a single concrete subclass, then `obj instanceof ImplementedOnce` if and only if `obj.getClass() == getSingleConcreteSubclass(ImplementedOnce)`. We do not include the benchmarks for this change because the performance impact is very small: executing `instanceof` is cheap relatively to invokes, and only a fraction of `instanceof` checks has been optimized.

Originally we expected to use the class hierarchy information to optimize the invokes as well. However, we found out that the runtime profiling information subsumes our class hierarchy analysis in this case. Indeed, if a class (an interface) does not have a concrete subclass, instances of this class (this interface) do not exist and cannot serve as a receiver to an invocation of this class's (interface's) method. Hence this class's (interface's) methods are never invoked[1].

If a class has a single concrete implementor, which is also tracked by our class hierarchy analysis, its implementor is always the receiver class whenever methods of the ancestor class are invoked. As such, it is cached in the invoke cache (see 4.4.1 for more details) and the cache entry is used to serve the invocations. We considered another devirtualization technique found in literature [7] but found that it is also subsumed by the Java on Truffle invoke caches.

4.4 Invoke cache

4.4.1 Background

As discussed in section 2.5.2, the first step of executing `invokeinterface` and `invokevirtual` bytecodes is resolving the method for the observed receiver. This step is expensive, especially for interface methods. To mitigate that, Java on Truffle implements a specialization with a cache of size 8 that caches the resolved method based on the receiver type. Whenever the method is invoked, the cache is traversed and the receiver class is compared to each of the cached types. This is beneficial in two ways: first, straightforward, it is cheaper to do the class comparison rather than resolving the method every time. Second, and more powerful: cache entries are

[1] This is a slight omission. In fact, the bytecode verifier does not check that the receiver of an `invokeinterface` implements the respective interface (the respective check is performed for `invokevirtual`). Hence it is possible to observe an invocation of an interface method for the interface that has not been implemented by a concrete class, and such an invocation should throw an exception. However, these cases are exceptionally rare and improving them has no effect on performance.

added only in the interpeter and are compilation final. Hence, they are constants in the compiled code, and the compiler observes a devirtualized call, which can be optimized or inlined. Since the entries are compilation final, the loop that traverses the cache is unrolled into an if cascade, which compares the current receiver class to each of the cached entries.

These caches are implemented in a simple way. The entries are added on first come — first serve basis, to the top of the cache. Hence, the cache consists of the first 8 observed entries in the reverse order. This has several downsides. First, the cache might include rarely executed methods (for example, executed once during VM setup and never again). Second, the cache entries might be out of order execution frequency-wise. Lastly, since we cache on the receiver class, the cache can contain duplicates: two classes can share an implementation of a method, for example, if their common ancestor defines it and neither class overrides it. We made several modifications to the caches in order to address the points above.

4.4.2 Profiling

As discussed above, in the compiled code the cache turns into an if cascade, with the most recently added entry on top. The cache is not profiled. This means that the default probability of 0.5 is applied to the branches, and the compiler estimates the probability to observe the first added entry as 2^{-8}, i.e. 0.4%. Because of that, even though the call is devirtualized, the compiler prioritizes optimizing other cache entries.

To mitigate this effect, we added a counting condition profile to each cache entry that tracked whether the `if` or the `else` branch was taken and informed the compiler about the corresponding probability. The combination of the information from all profiles of the cache thus reflected the actual probability to follow a path in the if cascade.

We did not write a microbenchmark for this change. The idea behind the change was to observe whether the profiling helps the compiler needs to prioritize which of the entries in the if cascade should be optimized. In a small benchmark the compiler has enough budget for all the cases.

Notable limitation of this approach is that the profiling information is only collected in the interpreter. If the application behaviour changes once it has been compiled, the performance will not be optimal because this behaviour change cannot be detected. We attribute the performance regressions in several scala-dacapo benchmarks to this effect. The performance effect is shown on graph 4.1.

An extension of this approach would be to profile not only the cache entries to determine the relative frequency, but also the generic, non-devirtualized case, to notify the compiler whether it is useful to perform optimizations on the devirtualized cases (extreme example: if the declared method has a lot of overrides, such as `Object.toString()`, invocations of cached resolved methods account for only a small percentage of all invocations, therefore it is not improrant to

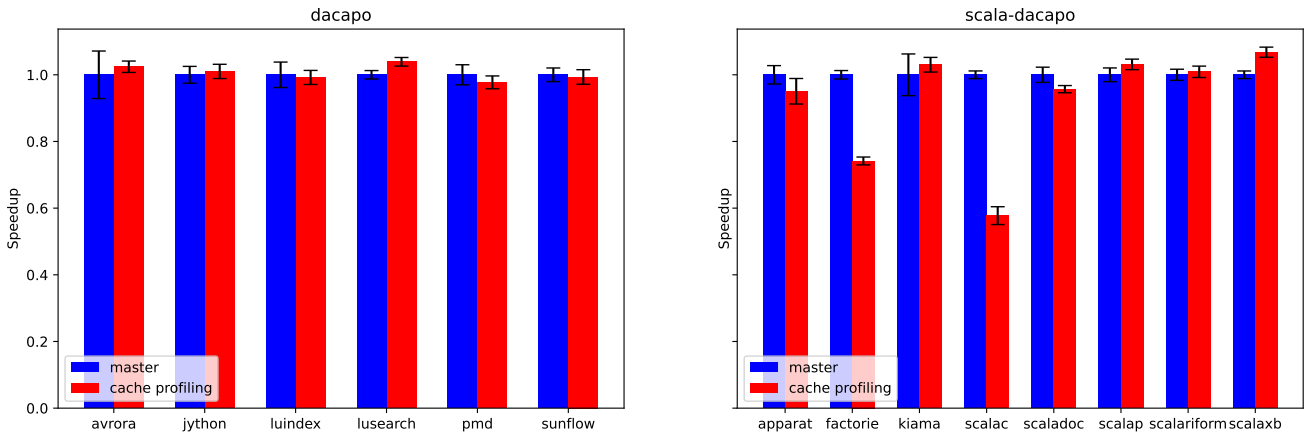

Figure 4.1: Impact of cache profiling

optimize their code). This is not possible due to the structure of the generated code: Truffle does not understand that the generic case is the fallback when the cache overflows and there is no if-else relation between them that can be profiled.

4.4.3 Sorting

Adding profiling was a non-invasive way to optimize the if cascade. Profiling did not help the compiler to reorder cache entries, because the compiler cannot reorder conditionals in the general case, as they might include side effects. Since we know that cache lookup does not have side effects, we manually sorted the cache entries by execution frequency.

To sort the cache entries, we counted the total number of executions served by the cache plus the generic case, as well as the number of times each cache entry has been executed. Once in a few hundred iterations we reordered the cache based on the recorded numbers.

```
/* setup: cache entries for invokeinterface in callInterfaceMethod filled
    by Impl1 (head), .., Impl8 (tail) */

private void callInterfaceMethod(BaseInterface i) {
        i.do(); // invokeinterface BaseInterface.do:()V
}

@Benchmark
public void benchmarkReverseCacheOrder() {
        /* interfaces == [Impl1, ..., Impl8] -- implementors of
           BaseInterface */
        for (int i = 0; i < interfaces.length; i++) {
```

```
                /* invoke frequency: ImplN.do() is called N times */
                for (int j = 0; j <= i; j++) {
                        callInterfaceMethod(interfaces[i]);
                }
        }
}
```

Listing 4.1: Sorting microbenchmark

For this change, we wrote a microbenchmark that represents the extreme case when the cache is full and their entries appear in the reverse order with respect to the frequency of execution (see 4.1). On this microbenchmark we observed a 10% speedup in the version with sorting. However, we saw little effect on macrobenchmarks (graph 4.2). It turned out that there were only a few cache sorts during the benchmarks execution, typically under 0.5% of sortedness checks resulted in cache reordering.

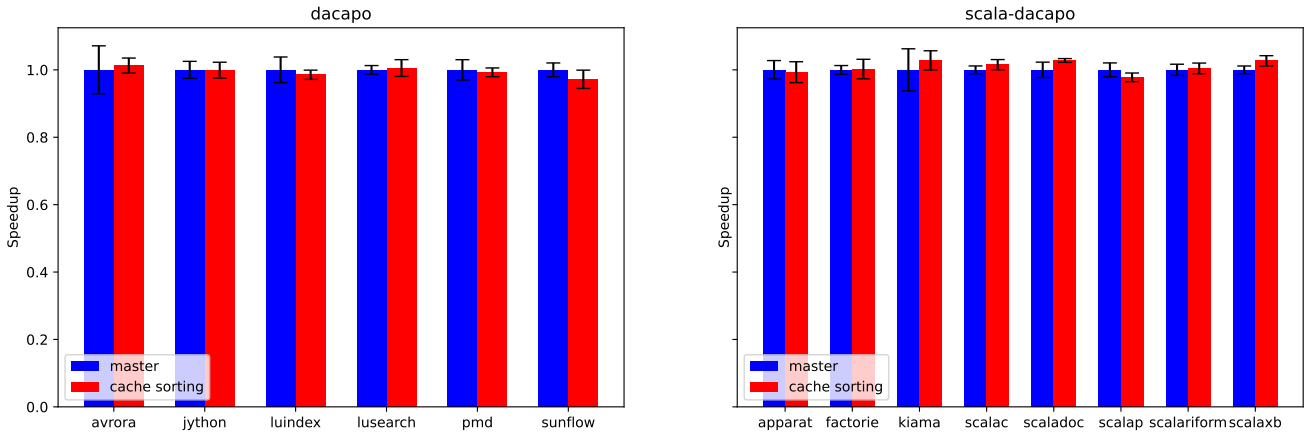

Figure 4.2: Impact of cache sorting

4.4.4 Pruning

In our last change to the cache content, we pruned rare entries. The original cache implementation simply kept first 8 observed entries; all following calls to other methods were indirect. By pruning entries, we freed up space in the cache.

The implementation was similar to cache sorting: we counted the total number of executions and the number of executions of each cache entry, and once in a a few hundreds iterations removed entries that accounted for less than 2% executions. We only considered for removal entries that have been added at least 100 calls prior to the time of pruning to avoid removing

entries that did not accumulate enough executions simply because they were added very recently.

```
/* interfaces == [Impl1, ..., Impl9] -- implementors of BaseInterface */
/* setup: cache entries for invokeinterface in callInterfaceMethod filled
    by Impl8 (head), .., Impl1 (tail). Impl9 does not fit in cache. */

private void callInterfaceMethod(BaseInterface i) {
        i.do(); // invokeinterface BaseInterface.do:()V
}

@Benchmark
public void benchmarkRareEntryCacheOverflow() {
       /* skip Impl1; calls to Impl2.do() .. Impl8.do() will be
          devirtualized, but not the call to Impl9.do() */
        for (int i = 1; i < interfaces.length; i++) {
                callInterfaceMethod(interfaces[i]);
        }
}
```

Listing 4.2: Pruning microbenchmark

We wrote a microbenchmark (see 4.2) in which a rarely (once) executed entry took a cache slot, hence a frequently invoked method did not fit in the cache. In this extreme case, we observed a 2x speedup with pruning.

However, while there were some gains on the microbenchmarks, there were also very significant regressions (graph 4.3). The reason behind is increased number of deoptimizations of the compiled code: removing rare entries means more additions to the cache are possible, and each addition results in invalidating the compiled code and transfering back to the interpreter. This effect is very visible on scala-dacapo benchmarks since Scala code typically has more potential cache entries at each invocation.

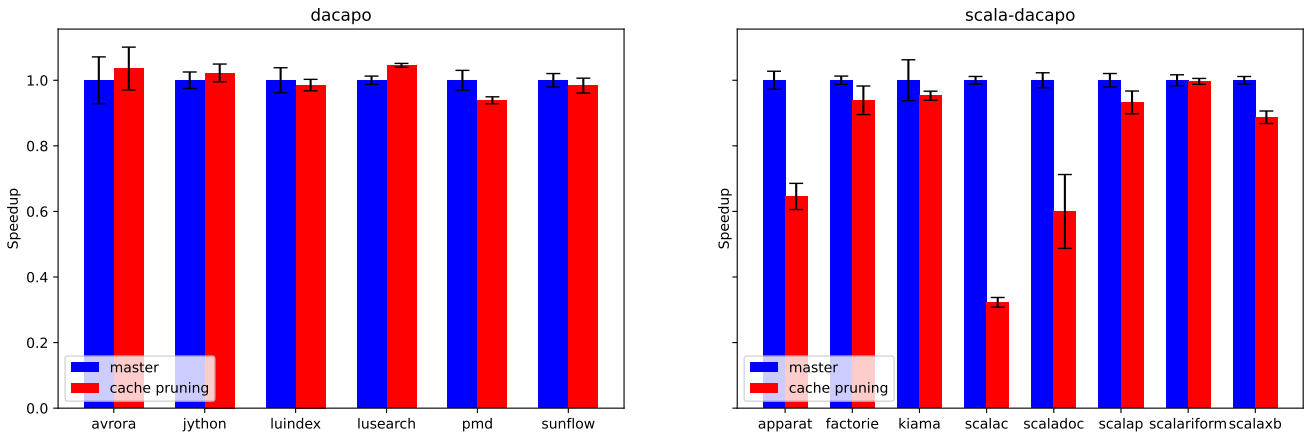

Figure 4.3: Impact of cache pruning

4.4.5 Cache deduplication

Lastly, we experimented with deduplicating the invoke caches. Currently, an invoke cache contains a duplicate method if the same method implementation is inherited by two receiver classes, which are observed early enough to fit in the cache. The effect of a duplicate is not only an extra occupied slot, but also more expensive inlining. This happens because the inlining algorithm observes two different dervirtualized calls, even though the call target is the same. Hence these two calls to the same guest method at the same guest callsite are inlined separately, spending extra budget.

In this experiment we made the inliner inline each method at most once. An easy approach to this is straightforward deduplication: for each receiver, resolve the target method and check if it is contained in the cache, instead of using the receiver class as key. However, as discussed earlier, method resolution is more expensive than looking up the class. We took a different approach: implementing a two-level cache, with the same first level – the receiver class mapped to the resolved method – and an additional second level, with the resolved method as the key. The call target invocation happens on the second level, so the inliner observes only one per resolved method call.

We did not write a microbenchmark for this change because the gain concerns the inliner budget, and it is very hard to impossible to write such a benchmark that inlining budget is sufficient to inline one copy of the resolved method but not multiple.

In this experiment, we observed improvements on almost all scala-dacapo benchmarks (graph 4.4), since Scala code tends to have more cache duplicates. We attribute the small re-

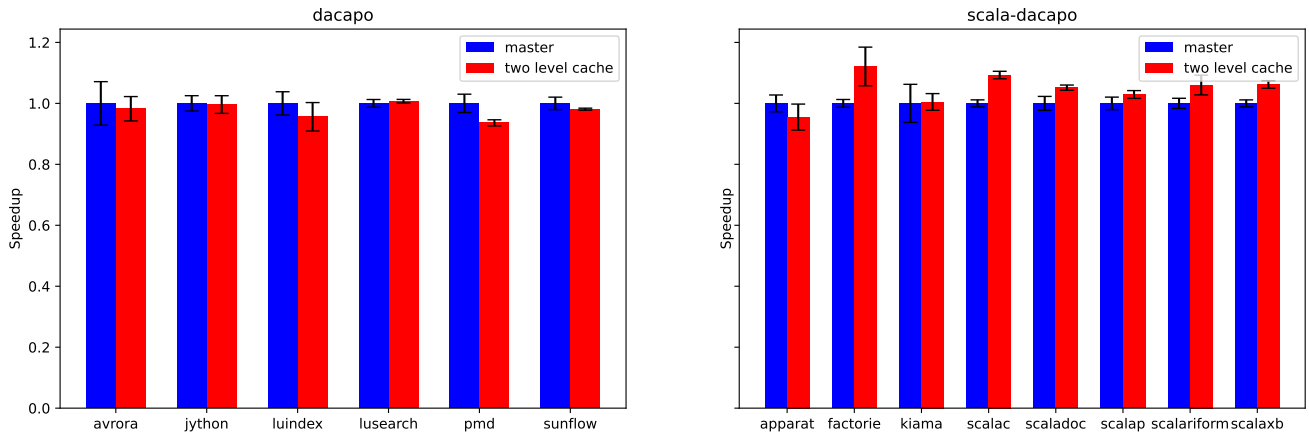

Figure 4.4: Impact of two level cache

gressions that we observe to losing the precise typing information in a deduplicated inlined method: with the previous cache implementation, method invocation for each receiver class was inlined separately, and the knowledge of the receiver class could have been used inside the inlined method. However, after deduplication a method is inlined at most once, for the most generic receiver type that has this method implementation.

4.5 Method splitting

As discussed above (section 2.5.2), Java on Truffle's representation of a guest method contains its call target, which is tied to a root node that executes the method. This means every time a guest method is executed, the same node is executed, and the profiling information inside the node is common for the all the invocations at different callsites.

It is sometimes beneficial to create a copy of the method's node for a given callsite and collect the profiling information inside the method only for that external callsite. Specifically, it provides an improvement if the polymorphism (multiple activated specializations, multiple cache entries) inside the method comes from the invocations from different callsites.

If the Truffle language communicates to the compiler when its nodes become polymorphic, Graal compiler can analyze it and detect when the reason of the polymorphism is multiple callers. If this happens, the compiler marks the call target for splitting. It means that within the splitting budget, the call target (hence its node) will be copied at each callsite.

If this monomorphisation is successful, it can be very beneficial. But it also comes with a cost: the node at each callsite is created anew, and the profiling has to be collected from zero.

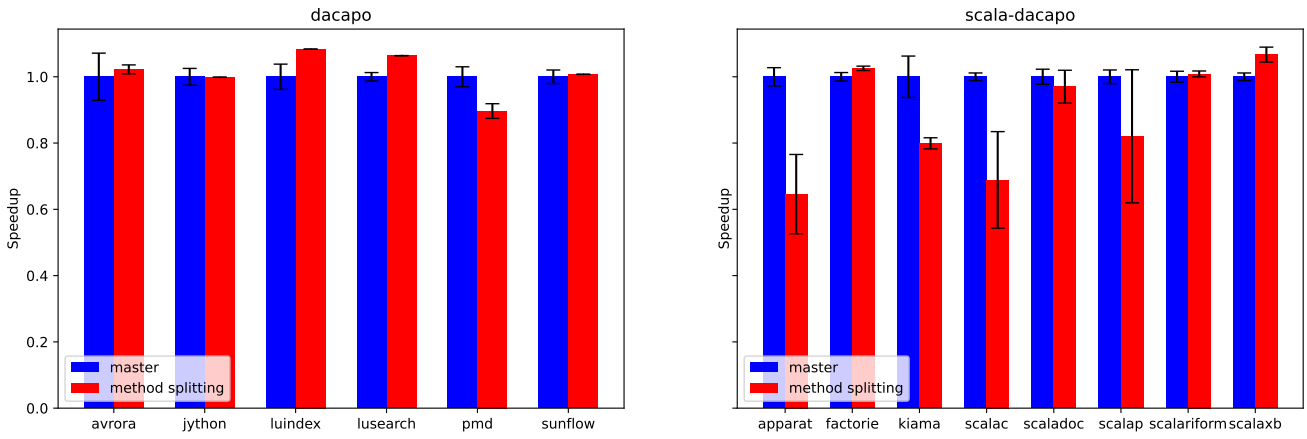

Figure 4.5: Impact of method splitting

This incurs a warmup cost. The benchmark results on the graph 4.5 confirms these effects.

4.6 Late profiling

As we discussed earlier, Java on Truffle collects profiling information at each callsite and for each `instanceof` invocation. This is useful if the enclosing method is long-running, but might be wasteful if it is only executed a few times. In addition, the profiling information at the application startup might no longer be relevant once the application is up and running. A particular case of this is the VM startup, which executes a lot of code, filling the caches with entries that are unlikely to be seen again. We experimented with starting to collect this information with a delay, doing slow indirect calls and full type checks in the meantime. We observed 5 to 10% performance gain on most benchmarks as illustrated by the graph 4.6.

4.7 Arraycopy

For more insight into performance of Java on Truffle, we turned to comparing it to HotSpot performance. To do that, we ran JMH microbenchmarks suite [9]: tiny benchmarks, evaluating the performance of JDK methods. Since we were comparing to HotSpot, we ran Java on Truffle in the JVM mode. It is possible but cumbersome to compare Java on Truffle, running the JMH benchmarks as a native image, to the native image of the benchmarks: JMH extensively uses reflection, which violates the closed world assumption and requires manual configuration.

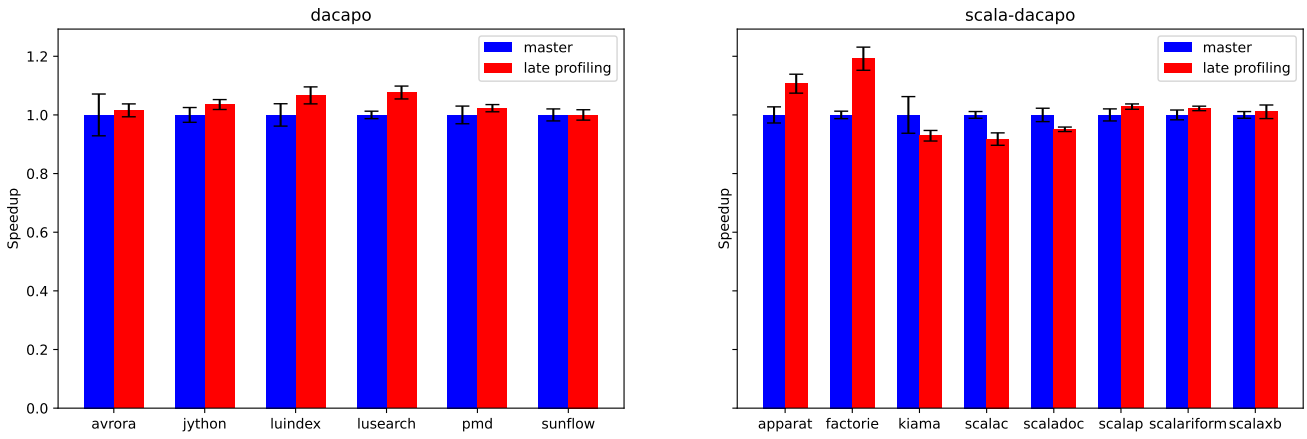

Figure 4.6: Impact of late profiling

We found that `System.arraycopy` is 3.8 times slower on Java on Truffle than on HotSpot. Since this is an important case, underlying not only array operations but also container and string operations, we started by investigating it.

`arraycopy` is implemented as a substitution. The reason behind it is that in host Java, `System.arraycopy` is a native method, and overhead of calling a native method for something as critical as `System.arraycopy` is too high. Instead, Java on Truffle implements its own `arraycopy`, which ultimately calls into the host one, copying between representations of guest arrays, but performs the necessary checks beforehand (whether both objects are arrays, whether the types of array elements are compatible, bounds check, etc – all of them in the order defined in the documentation).

Originally the substitution was implemented as a method, different cases in which (for example, primitive vs reference arrays) were guarded by a profile. We rewrote this method to a `Node` with suitable specializations, which allowed a more precise activation of cases. We improved copying of primitive arrays by writing a specialization for each primitive element type. Lastly, we enabled splitting of this substitution. By doing these changes, we improved the `System.arraycopy` performance 1.5x.

Chapter 5

Future work

First, when investigating the performance of `System.arraycopy`, we found that the corresponding JMH microbenchmark does several field reads. Java on Truffle supports Polyglot capabilities, i.e. calling into other Truffle languages. With this flexibility comes a performance cost: even if a given application is purely Java (which is the case for all benchmarks), when executing it, Java on Truffle still checks on every field read whether that field is an object from another language. As observed on the `System.arraycopy` example, if the application is read-heavy, these checks take a noticeable amout of time (about 15% in that specific case). Investigating a way to reduce the number of these checks would be beneficial.

Second, we hope that our framework of comparing Java on Truffle's performance and HotSpot's performance will be used to further investigate the cases where Java on Truffle's relative slowdown is above average and that it brings further insights.

Third, we believe that the splitting approach can be improved. Right now it is all or nothing: either all calls use the same node or each callsite gets its own copy. We hope that a hybrid approach of splitting only at often executed callsites will be explored.

Fourth, we are looking forward to the ahead-of-time mode of Java execution which will use the class hierarchy information more profoundly and incentivize further development of this analysis.

Bibliography

[1] Jeffrey Dean, David Grove, and Craig Chambers. “Optimization of Object-oriented Programs Using Static Class Hierarchy Analysis”. In: *European Conference on Object-Oriented Programming*. 1995.

[2] *FastR*. https://www.graalvm.org/r/.

[3] *GraalPython*. https://www.graalvm.org/python/.

[4] *GraalVM Community, GraalVM Enterprise, and V8 Benchmarks Comparison*. https://www.graalvm.org/javascript/.

[5] *GraalVM JavaScript*. https://github.com/oracle/graaljs.

[6] *GraalWasm*. https://www.graalvm.org/22.0/reference-manual/wasm/.

[7] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. “A Study of Devirtualization Techniques for a Java Just-In-Time Compiler”. In: *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA ’00. Minneapolis, Minnesota, USA: Association for Computing Machinery, 2000, pp. 294–310. ISBN: 158113200X. DOI: 10.1145/353171.353191. URL: https://doi.org/10.1145/353171.353191.

[8] *Java Microbenchmark Harness*. https://openjdk.java.net/projects/code-tools/jmh/.

[9] *JMH JDK Microbenchmarks*. https://github.com/openjdk/jmh-jdk-microbenchmarks.

[10] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. “GraalSqueak: Toward a Smalltalk-Based Tooling Platform for Polyglot Programming”. In: Oct. 2019. ISBN: 978-1-4503-6977-0. DOI: 10.1145/3357390.3361024.

[11] Salim S. Salim, Andy Nisbet, and Mikel Luján. “TruffleWasm: A WebAssembly Interpreter on GraalVM”. In: *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE ’20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 88–100. ISBN: 9781450375542. DOI: 10.1145/3381052.3381325. URL: https://doi.org/10.1145/3381052.3381325.

[12] *TruffleRuby*. https://www.graalvm.org/ruby/.

[13] *V8 JavaScriptEngine*. https://v8.dev/.

[14] Christian Wimmer and Thomas Würthinger. “Truffle: A Self-Optimizing Runtime System”. In: *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*. SPLASH ’12. Tucson, Arizona, USA: Association for Computing Machinery, 2012, pp. 13–14. ISBN: 9781450315630. DOI: 10.1145/2384716.2384723. URL: https://doi.org/10.1145/2384716.2384723.

[15] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. “One VM to Rule Them All”. In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pp. 187–204. ISBN: 9781450324724. DOI: 10.1145/2509578.2509581. URL: https://doi.org/10.1145/2509578.2509581.

[16] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. “One VM to Rule Them All”. In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pp. 187–204. ISBN: 9781450324724. DOI: 10.1145/2509578.2509581. URL: https://doi.org/10.1145/2509578.2509581.