Making Collection Operations Optimal with Aggressive JIT Compilation

Aleksandar Prokopec
Oracle Labs
Switzerland
aleksandar.prokopec@oracle.com

David Leopoldseder
Johannes Kepler University, Linz
Austria
david.leopoldseder@jku.at

Gilles Duboscq
Oracle Labs
Switzerland
gilles.m.duboscq@oracle.com

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

Abstract

Functional collection combinators are a neat and widely accepted data processing abstraction. However, their generic nature results in high abstraction overheads – Scala collections are known to be notoriously slow for typical tasks. We show that proper optimizations in a JIT compiler can widely eliminate overheads imposed by these abstractions. Using the open-source Graal JIT compiler, we achieve speedups of up to 20× on collection workloads compared to the standard HotSpot C2 compiler. Consequently, a sufficiently aggressive JIT compiler allows the language compiler, such as Scalac, to focus on other concerns.

In this paper, we show how optimizations, such as inlining, polymorphic inlining, and partial escape analysis, are combined in Graal to produce collections code that is optimal with respect to manually written code, or close to optimal. We argue why some of these optimizations are more effectively done by a JIT compiler. We then identify specific use-cases that most current JIT compilers do not optimize well, warranting special treatment from the language compiler.

CCS Concepts • **Software and its engineering → Just-in-time compilers**;

Keywords collections, data-parallelism, program optimization, JIT compilation, functional combinators, partial escape analysis, iterators, inlining, scalar replacement

ACM Reference Format:
Aleksandar Prokopec, David Leopoldseder, Gilles Duboscq, Thomas Würthinger 2017. Making Collection Operations Optimal with Aggressive JIT Compilation . In *Proceedings of 8th ACM SIGPLAN International Scala Symposium (SCALA'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3136000.3136002

SCALA'17, October 22–23, 2017, Vancouver, Canada
© 2017 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of 8th ACM SIGPLAN International Scala Symposium (SCALA'17)*, https://doi.org/10.1145/3136000.3136002.

1 Introduction

Compilers for most high-level languages, such as Scala, transform the source program into an intermediate representation. This intermediate program representation is executed by the runtime system. Typically, the runtime contains a just-in-time (JIT) compiler, which compiles the intermediate representation into machine code. Although JIT compilers focus on optimizations, there are generally no guarantees about the program patterns that result in fast machine code – most optimizations emerge organically, as a response to a particular programming paradigm in the high-level language. Scala's functional-style collection combinators are a paradigm that the JVM runtime was unprepared for.

Historically, Scala collections [Odersky and Moors 2009; Prokopec et al. 2011] have been somewhat of a mixed blessing. On one hand, they helped Scala gain popularity immensely – code written using collection combinators is more concise and easier to write compared to Java-style *foreach* loops. High-level declarative combinators are exactly the kind of paradigm that Scala advocates. And yet, Scala was often criticized for having suboptimal performance because of its collection framework, with reported performance overheads of up to 33× [Biboudis et al. 2014]. Despite several attempts at making them more efficient [Dragos and Odersky 2009; Prokopec and Petrashko 2013; Prokopec et al. 2015], Scala collections currently have suboptimal performance compare to their Java pendants.

In this paper, we show that, despite the conventional wisdom about the HotSpot C2 compiler, **most overheads introduced by the Scala standard library collections can be removed and optimal code can be generated with a sufficiently aggressive JIT compiler**. We validate this claim using the open-source Graal compiler [Duboscq et al. 2013], and we explain how. Concretely:

- We identify and explain the abstraction overheads present in the Scala collections library (Section 2).
- We summarize relevant optimizations used by Graal to eliminate those abstractions (Section 3).

- We present three case studies, showing how Graal's optimizations interact to produce optimal collections code (Section 4).
- We show a detailed performance comparison between the Graal compiler and the HotSpot C2 compiler on Scala collections and on several Spark data-processing workloads. We focus both on microbenchmarks and application benchmarks (Section 5).
- We postulate that profiling information enables a JIT compiler to better optimize collection operations compared to a static compiler. We then identify the optimizations that current JIT compilers are unlikely to do, and conclude that *language compilers should focus on optimizing data structures more than on code optimization* (Section 7).

Note that the purpose of this paper is not to explain each of the optimizations used in Graal in detail – this was either done elsewhere, or will be done in the future. Rather, the goal is to explain how we made Graal's optimizations interact in order to produce optimal collections code. To reiterate, we do not claim novelty for any specific optimization that we describe. Instead, we show how the correct set of optimizations is combined to produce optimal collections code. As such, we hope that our findings will guide the language implementors, and drive future design decisions in Scala (as well as other languages that run on the JVM).

2 Overheads in Scala Collection Operations

On a high-level, we categorize the abstraction overheads present in Scala collections into several groups [Prokopec et al. 2014]. First, most Scala collections are generic in their parameter type. At the same time, Scala relies on type erasure to eliminate generic type parameters in the intermediate code (similar to Java [Bracha et al. 2003]). At the bytecode level, collection methods such as `foldLeft`, `indexOf` or `+=` take arguments of the type `Object` (in Scala terms, `AnyRef`). When a collection is instantiated with a value type, such as `Int`, most method invocations implicitly allocate heap objects to wrap values, which is called *boxing*.

Second, collections form a class hierarchy, with their operations defined in base traits. Collection method calls are often polymorphic. A compiler that does not know the concrete callee type, cannot easily optimize this call. For example, two `apply` calls on `ArrayBuffer` share the same array bounds check, but if the static receiver type is the base trait `Seq`, the compiler cannot factor out the common bounds check.

Third, most of the collection combinators are parametric – they take a lambda that fully specifies the combinator's behavior. Since the implementation site does not know the implementation of the lambda, it cannot be optimized with respect to the lambda. For example, in the expression `xs.foreach(this.buffer += _)`, unless the compiler specializes the callsite (e.g. by inlining), the read of the `buffer` field cannot be factored out of the loop, but instead occurs for every element.

Finally, most collection operations (in particular, subclasses of the `Iterable` trait) create `Iterator` objects, which store iteration state. Avoiding the allocation of the `Iterator` and inlining the logic of its `hasNext` and `next` methods generally results in more efficient code, but without knowing the concrete implementation of the iterator, replacing iterators with something more efficient is not possible.

3 Overview of the Optimizations in Graal

Graal [Duboscq et al. 2013] is a dynamic optimizing compiler for the HotSpot VM. Graal is not the default top-tier compiler in HotSpot, but it can optionally replace the C2 default compiler. Just like the standard C2 compiler, Graal uses feedback-driven optimizations and generates code that speculates on the method receiver types and the taken control flow paths. The feedback is based on receiver type profiles and branch probabilities, and is gathered while the program executes in interpreter mode. When a speculation proves incorrect, Graal deoptimizes the affected part of the code, and compiles it again later [Duboscq et al. 2014].

An important design aspect in Graal, shared among many current JIT compilers, is the focus on intraprocedural analysis. When HotSpot detects that a particular method is called many times, it schedules the method for compilation. The JIT compiler (C2 or Graal) analyzes the particular method, and attempts to trigger optimizations within its scope. Instead of doing inter-procedural analysis, additional optimizations are enabled by inlining the callsites inside the current method.

Graal's intermediate representation (IR) is partly based on the sea-of-nodes approach [Click and Paleczny 1995]. A program is represented as a directed graph in static single assignment form. Each node produces at most one value. Control flow is represented with fixed nodes for which successor edges point downwards in the graph. Data flow is represented with floating nodes whose input edges point upwards. The details of Graal IR are presented in related work [Duboscq et al. 2013]. In section 4, we summarize the basic IR nodes that are required for understanding this paper.

In the rest of this section, we describe the most important high-level optimizations in Graal that are relevant for the rest of the discussion. To make the presentation more accessible to the audience, we show the optimizations at the Scala source level, while in fact they work on the Graal IR level. We avoid the details, and aim to give an intuition – where necessary, we point to related work.

3.1 Inlining

The inlining transformation identifies a callsite, and replaces it with the body of the callee. For example, the `sq` method:

```
1 def sq(x:Double) = x * x
2 def tss(xs:List[Double]):Double =
3   if (xs.isEmpty) 0
4   else sq(xs.head)+tss(xs.tail)
```

can be inlined into the `tss` method:

```
1 def tss(xs:List[Int]):Double =
2   if (xs.isEmpty) 0 else {
3     val h = xs.head
4     h*h+tss(xs.tail)
5   }
```

While the transformation itself is straightforward, the decision of when and what to inline is anything but simple. Inlining decisions are often based on hand-tuned heuristics and various rules of the thumb, which is why some researchers called it *black art* in the past [Peyton Jones and Marlow 2002].

On a high-level, Graal inlining works by constructing a call tree starting from the currently compiled method. The complete call tree may be infinite, so at some point Graal stops exploring and inlines the methods considered worthy.

3.2 Polymorphic Inlining

Inlining is only possible if the concrete implementation of the method that needs to be called is statically known. In Java or in Scala, this means having a precise static type for the receiver. In typical Scala collections code, that is rarely the case, since the convention is to use generic collection traits, such as `Set`, `Seq` or `Map`. The JVM uses *inline caches* at its polymorphic callsites, which store pairs of the possible receiver types and the method entry addresses. This design is similar to the one used in Smalltalk [Deutsch and Schiffman 1984].

Aside from relying on inline caches of the VM, a JIT compiler can emit *type switches* – a sequence of if-then-else type checks that dispatch to the concrete implementation. In Graal, such a type switch usually does not have more than 3-4 checks, and the types used in it are based on the callsite's receiver type profile. Consider the `tss` method:

```
1 def tss(xs:Seq[Int]):Int = xs.fold(0)(_+sq(_))
```

To avoid the indirect call, a JIT compiler with access to the type profile of `xs` can emit the following code:

```
1 def tss(xs:Seq[Int]):Int = xs match {
2   case xs:List[Int] =>     xs.fold(0)(_+sq(_))
3   case xs:ArrayOps[Int] => xs.fold(0)(_+sq(_))
4   case _ =>                xs.fold(0)(_+sq(_))
5 }
```

Note that in the previous snippet, the `fold` calls in lines 2 and 3 are direct. The benefit of type switches, as we show in Section 4.2, is that they allow additional inlining.

3.3 Canonicalization

In Graal, *canonicalization* refers to a compiler pass that applies different optimizations that simplify code, make it more efficient or bring it into a canonical form. This includes classical optimizations, such as strength reduction (`x*2` becomes `x<<1`), global value numbering [Click 1995] (two occurrences of the same value are shared in the IR), constant folding (`3*2` becomes `6`), branch or dead-code elimination, as well as JVM-specific optimizations such as replacing a type-check with a constant (when both the receiver type and the checked type are known). Canonicalization is applied incrementally throughout the IR until the compiler decides that there are no more canonicalization opportunities. Graal applies canonicalization repeatedly throughout the compilation process. Its defining feature is that it is a *local optimization* – it can be applied to a particular IR node by inspecting its immediate neighbourhood. It is therefore relatively cheap, and can be used as a subphase inside other optimizations.

3.4 Partial Escape Analysis and Scalar Replacement

Partial escape analysis (PEA) [Stadler et al. 2014], a control flow sensitive variant of escape analysis [Kotzmann and Mössenböck 2005], is an analysis that allows a compiler to determine if an object escapes the current compilation unit. This information can be used to perform optimizations such as scalar replacement that eliminates an object allocation and replaces it with the scalar values of the object.

Consider the example in Listing 1, in which the iteration state is kept in an `ArrayIterator` object. The `it` object is not stored into another object, passed to other functions or returned. Consequently, the `it` object is just an extra allocation in this code, and can be completely removed. Every read of `it.array` can be replaced with `a`, since that is the value in the preceding write to the `array` field in line 4. Similarly, every read and write to `it.current` can be replaced with a local variable.

Listing 1. Sum of squares using iterators

```
1 def tss(a:Array[Double]):Double = {
2   val it = new ArrayIterator
3   it.current = 0
4   it.array = a
5   var sum = 0.0
6   while (it.current < it.array.length) {
7     sum += sq(it.array(it.current))
8     it.current += 1
9   }
10  sum
11 }
```

3.5 Interaction between Optimizations

Each of the previously described optimizations usually improves program performance. However, improvements are higher when these optimizations are used together – one optimization often enables another. For example, inlining

typically introduces constants and simplifiable code patterns into the IR, thus introducing opportunities for constant folding, branch elimination, and other canonicalizations. Canonicalization, on the other hand, typically eliminates calls or makes polymorphic calls direct, enabling additional inlining. Partial escape analysis eliminates read-write dependencies, enabling additional constant-folding.

For each method that gets compiled, Graal expands the call tree, and inlines the callsites considered worthwhile. Graal then alternates between these optimizations until deciding that the code was sufficiently optimized, or that there are no more optimization opportunities. At this point, the optimized IR is passed to the remaining compilation phases.

We deliberately keep some concepts vague and high-level. Deciding when to stop expanding the call tree, which callsites are worthwhile, or when to stop applying the optimizations, requires careful tuning and is outside the scope of this paper. In Graal, we rely heavily on carefully chosen heuristics – we found that, when done correctly, this process usually results in optimal code.

4 Case Studies

4.1 Case Study: `foldLeft`

In this section, we consider a typical usage of the `foldLeft` function, and show how a JIT compiler, in our case Graal, can optimize it. Listing 2 shows the implementation of `foldLeft` from the Scala standard library.

Listing 2. The `foldLeft` implementation

```
1 @tailrec
2 def foldl[B](s:Int,e:Int,z:B,op:(B,A)=>B):B =
3   if (s==e) z
4   else foldl(s+1,e,op(z,this(s)),op)
5 def foldLeft[B](z:B)(op:(B,A)=>B):B =
6   foldl(0,length,z,op)
```

The `foldLeft` method [Bird and Wadler 1988] (on indexed sequence collections) forwards to the `foldl` method, which either returns the accumulator `z`, or calls itself recursively at the next index. The `@tailrec` annotation instructs the Scala compiler to convert this method into a `while` loop equivalent.

Listing 3. The `computeSum` user method

```
1 def computeSum(xs: Array[Int]): Long =
2   xs.foldLeft(0L)(_ + _)
```

The `computeSum` method invokes the `foldLeft`. However, there is more to its translation than what meets the eye. The Scala compiler implicitly wraps the JVM array into an `ArrayOps` object, so that it can call `foldLeft`. Furthermore, `foldLeft` has a generic return type, which erases to `Object`. The compiler must insert an `unboxToLong` call to return a primitive value. The transformed code resembles the one in Listing 4.

Listing 4. The expansion of the `computeSum` method

```
1 def computeSum(xs: Array[Int]): Long =
2   BoxesRunTime.unboxToLong(
3     new ArrayOps$ofInt(xs).foldLeft(0L)(_ + _))
```

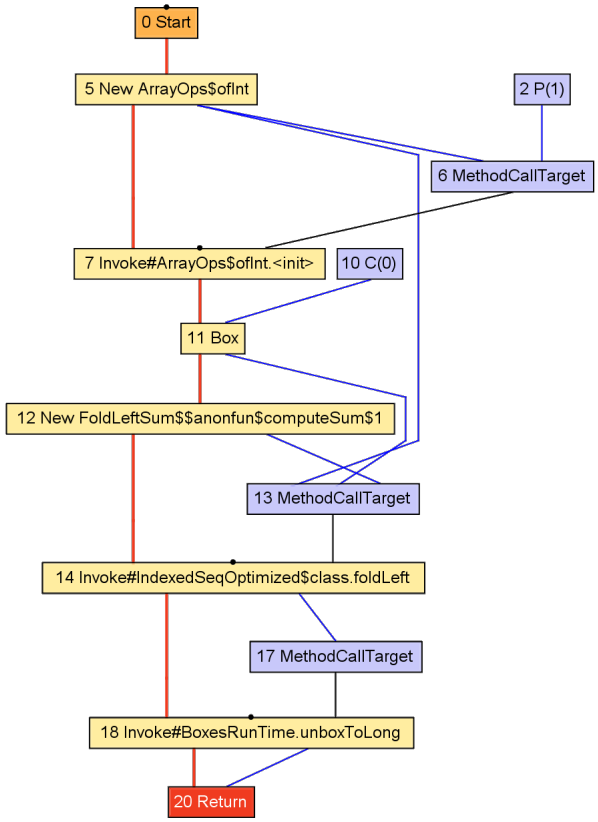


Figure 1. IR of `computeSum` after bytecode parsing

After Scalac transforms this code to JVM bytecode, and the JVM schedules the `computeSum` method for compilation, a JIT compiler, in our case Graal, produces the **intermediate representation** shown in Fig. 1. The yellow nodes denote control flow – for example, `Start` denotes the entry point into the method, `New` represents an object allocation, `Invoke` is a call to a method, `Box` denotes boxing of a primitive value, and `Return` represents the method's exit point. The purple nodes represent dataflow, and are called *floating nodes* – they *hang* from the control flow nodes, serving as their input or output. Their final position in the resulting machine code depends only on their inputs. The `C` node represents a constant, a `P(n)` node represents the *n*-th parameter, and a `MethodCallTarget` node wraps parameters for an `Invoke`. Finally, node 12 is a `Function2` allocation for the lambda `_ + _`.

As we will see soon, expression nodes are also encoded as floating nodes. For example, `+` encodes addition, `SignExtend` encodes a conversion from `Int` to `Long`, `==` represents equality, `InstanceOf` represents a type check, and `AllocatedObject` represents a reference to an allocated object. The list of all node types is outside the scope of this paper, so we only mention the relevant ones [Duboscq et al. 2013].

As most other modern JIT compilers, Graal relies on intraprocedural analysis – to trigger optimizations, it must

first inline the Invoke nodes in the IR of computeSum. To do this, Graal creates a **call tree** based on the body of computeSum.

Fig. 2(a) shows the initial call graph – the computeSum root node, represented with the expanded subgraph node (Sg), and the three Cutoff nodes: the ArrayOps constructor (1), the foldLeft (2) and unboxToLong (3). The inliner then inspects the call graph nodes, and explores the call graph, without actually inlining the calls yet. This subphase is called *expansion*. During the expansion, the inliner must parse the IR of the callees to find the nested callsites – for example, Fig. 3 shows the IR of foldLeft.

In Fig. 2(b), the inliner decides that it sufficiently expanded the call graph in the current round. Note that direct calls, such as foldLeft in node 4, are denoted with subgraph nodes (Sg). Indirect calls (whose receiver is statically unknown) are represented with type switch nodes (TS) – depending on the callsite type profile, the compiler may decide to embed a type switch for such nodes, as explained in Section 3.2. Note that the call to length (node 5) is indirect, since its caller (foldLeft from Listing 2) does not know the receiver type.

The inliner then forwards the callsite argument type information top-down through the call graph – in each method, it replaces the parameter nodes P(n) with the values from its callsite, and then triggers the canonicalization transformation from Section 3.3. The receiver types for length (node 5), apply (node 11) of the base sequence type GenSeqLike, and apply (node 12) of the Function type, thus become known, and these calls become direct – they are replaced with Cutoff nodes, to be re-examined, as shown in Fig. 2(c)).

The inliner must now decide what to actually inline into computeSum. The unboxToLong call (node 8) is a "no-brainer" – its IR is small, without nested calls, so it is immediately inlined. Note that, at this point, the body of the foldLeft method (node 4), is not yet fully explored. Therefore, the inliner does not yet believe that inlining the foldLeft method can trigger other optimizations, so it decides to leave it for the next round.

Fig. 4 shows a part of the computeSum after the 1st round – inlining unboxToLog introduced three *guard nodes* (FixedGuard). Roughly speaking, with guard 22, the compiler speculates that the true branch in Listing 5 will never be executed (if this speculation is incorrect, the code is deoptimized, and recompiled later without this speculation). Similarly, guards 24 and 26 speculate the absence of ClassCastException and NullPointerException, respectively. These speculations are generally more efficient than the variant with the branch and the exception code, but this code is not optimal – ideally, we would like to get rid of these guards altogether. This is currently not possible, since the guard conditions IsNull (node 21) and InstanceOf (node 23) are not constants. These conditions may become constant after more inlining.

Listing 5. The unboxToLong standard library method

```
1 public static long unboxToLong(Object x) {
2   return x==null ? 0 : ((Long)l).longValue(); }
```

The compiler then proceeds to the second inlining round, starting with expansion. In Fig. 5(a), the inliner completely explores the call tree beneath the foldLeft method, and inlines it in Fig. 5(b). The only remaining callsite in the computeSum method is now the ArrayOps constructor (Cutoff node 1). Inlining the other methods makes the guard conditions constant, as shown in Fig. 6(a) – the compiler now knows that null pointers and class cast exceptions are not possible, so canonicalization eliminates the guards, as seen in Fig. 6(b).

This code is now almost optimal. The only remaining performance overhead in the loop is the read (node 74) of the repr field in the ArrayOps wrapper (which wraps the native JVM array). The entries of the JVM array are read in node 75 (LoadIndexed). The inliner still has sufficient budget, so it inlines the ArrayOps constructor in the 3rd round, allowing other optimizations to reason about the fields of ArrayOps.

The ArrayOps wrapper object is represented by the node 110 (AllocatedObject) in Fig. 7. The ArrayOps constructor (node 7) writes the native JVM array (P(1)) into the repr field of the wrapper object. After that, the repr field is read again to determine the array length (and then again later in the loop, as was shown in Fig. 6(b)). This write-read dependency is detected during partial escape analysis and scalar replacement optimization, and all the repr reads are replaced with the floating node used in the corresponding write. Fig. 8 shows that, after PEA, the array length is computed directly from the parameter P(1) of computeSum. Consequently, the allocation of the ArrayOps object becomes dead code, and can be eliminated. PEA also eliminates pairs of Box and Unbox nodes from the apply method of Function2 (not shown).

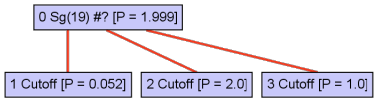
4.2 Case Study: Polymorphic foldLeft Callsite

One mitigating circumstance in the case study from Section 4.1 was that the receiver type of foldLeft was statically known. Dataflow analysis during inlining then determined the receiver type of the GenSeqLike.length, GenSeqLike.apply and Function2.apply. However, with a definition of computeSum in Listing 6, where global has a sequence super-type Seq[Int], pure dataflow analysis is no longer fruitful.

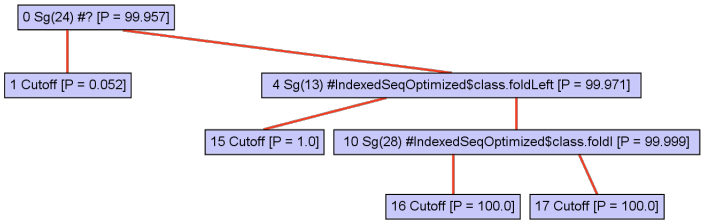
Listing 6. The computeSum with a global field

```
1 // This is set somewhere else in the program.
2 var global:Seq[Int] = null
3 def computeSum:Long = global.foldLeft(0L)(_+_)
```

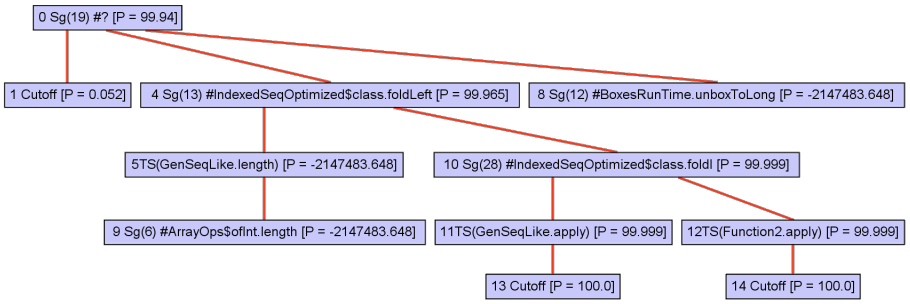
Without a whole-program analysis, which a JIT compiler generally cannot do, the exact type of global is unknown. However, a JIT compiler has receiver type profiles – a list of possible receiver types associated with probabilities, gathered during the program execution. Graal knows the receivers of the foldLeft call in line 3, and their probabilities.



(a) Before expansion.



(c) After inlining.



(b) After expansion.

Figure 2. Inlining call graph during the 1st inlining round

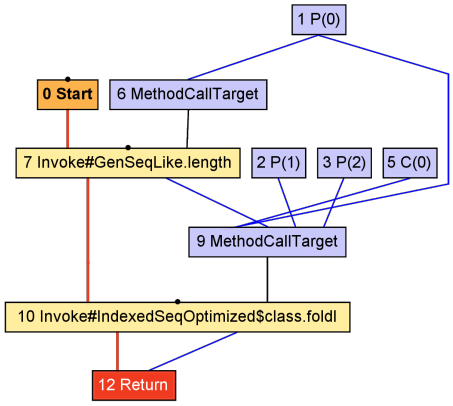


Figure 3. IR of `foldLeft` after bytecode parsing

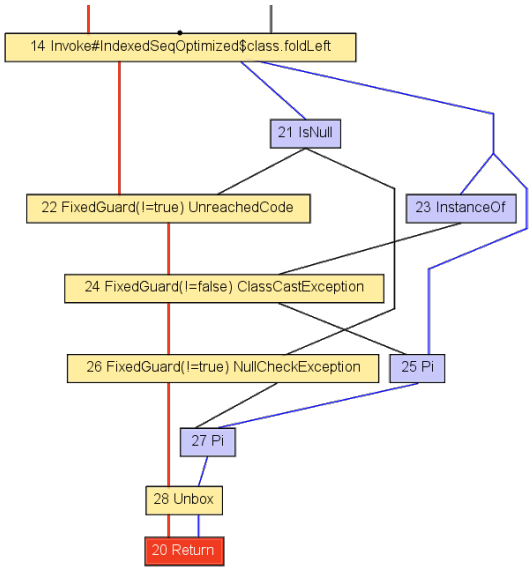


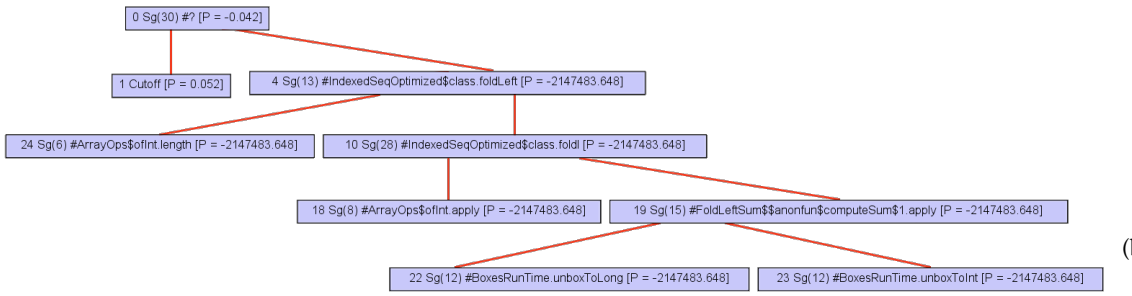
Figure 4. IR of `computeSum` after the 1st inlining round

Fig. 9(a) shows the call graph for Listing 6, with the profile indicating that `global` is either a Scala `List` or a `WrappedArray`. The type switch node 1 is replaced with if-then-else `InstanceOf` checks of the receiver (nodes 27 and 36), shown in Fig. 9(b) – in each branch, the implementation of `foldLeft` for that specific collection is inlined.

We emphasize that the receiver type profile is hardly ever available to a static compiler. Profile-based receiver type speculation, needed for optimality in use-cases such as Listing 6, is therefore much more amenable to a JIT compiler.

4.3 Case Study: `map`

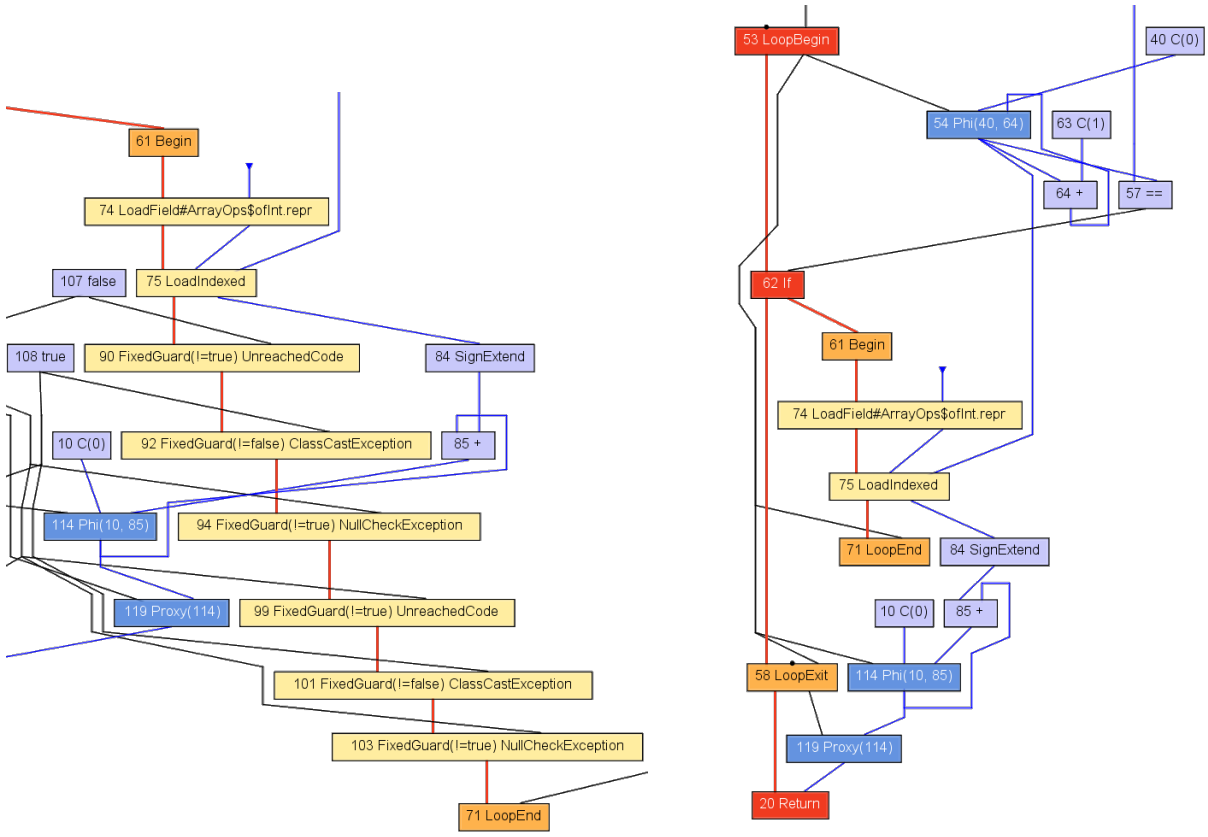
The previous case studies show how Graal optimizations interact when processing accessor operations. However, for operations that materialize a collection, Graal (and as we argue later, most current JIT compilers) is unlikely to produce optimal code. In this section, we study the `map` operation on



(a) After expansion.

(b) After inlining.

Figure 5. Inlining call graph during the 2nd inlining round



(a) Before canonicalization

(b) After canonicalization

Figure 6. IR of `computeSum` after the 2nd inlining round

`ArrayBuffer`. In Listing 7, numbers from an `ArrayBuffer` are incremented by one and stored into a new `ArrayBuffer`.

Listing 7. The `mapPlusOne` user method

```
1 def mapPlusOne(input:ArrayBuffer[Int]) =
2   input.map(_+1)
```

Note that the `map` operation is strict and it materializes a new collection – it must allocate a new buffer data structure to store the resulting elements. Next, the `ArrayBuffer` is generic in its type parameter `T`, and not specialized [Dragos and Odersky 2009]. The `ArrayBuffer` implementation internally keeps an array of objects, typed `Array[AnyRef]`. When adding a primitive value, such as `Int`, to an `ArrayBuffer` collection, the Scala compiler inserts a `boxToInt` call.

We only show the final IR of the `mapPlusOne` method after applying all the optimizations as explained earlier. Fig. 10 shows the initial part of the `map` loop, which reads a number

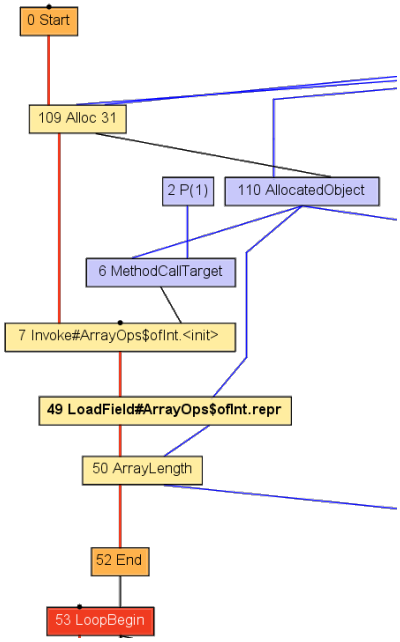


Figure 7. `ArrayOps` initialization before PEA

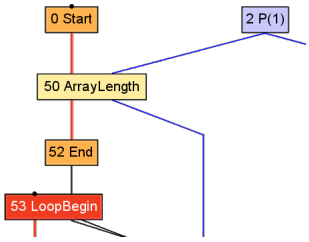


Figure 8. `ArrayOps` initialization after PEA

from the source array. Since the source array contains objects, Graal cannot guarantee that these objects are non-`null` or that they have proper types, so it needs to insert guards. An array read (`LoadIndexed`) is therefore followed by three guard nodes and an unboxing operation. Similarly, in Fig. 11, before storing the incremented primitive value back into the target array (`StoreIndexed` node 179), a `java.lang.Integer` object must be allocated (`Box` node 412). Partial escape analysis cannot eliminate this allocation, since the object escapes.

5 Performance Evaluation

In this section, we show the performance comparison of the snippets from Section 4, as well as additional Scala collection operation microbenchmarks, and several larger programs. We also include several Spark data-processing workloads, since Spark RDDs have overheads comparable to those in Scala collections [Zaharia et al. 2012]. We run all benchmarks on an 3.8 GHz Intel i7-4900MQ with 32 Gb RAM, with frequency scaling turned off. We used Scala 2.11, and the HotSpot implementation of JDK8, with the heap size set to 1

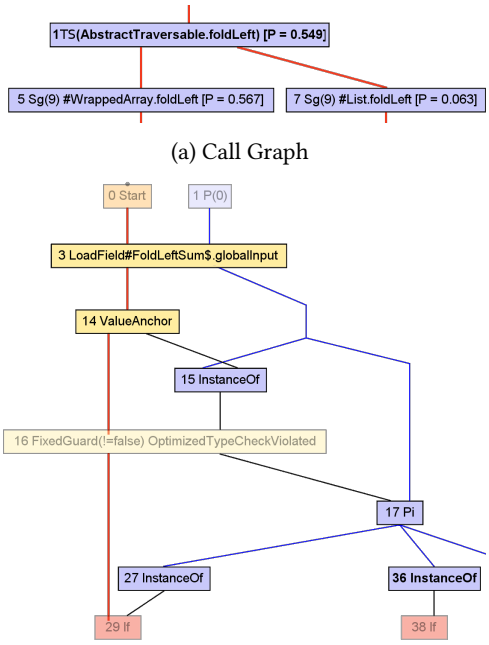


Figure 9. Creating the type switch in `computeSum`

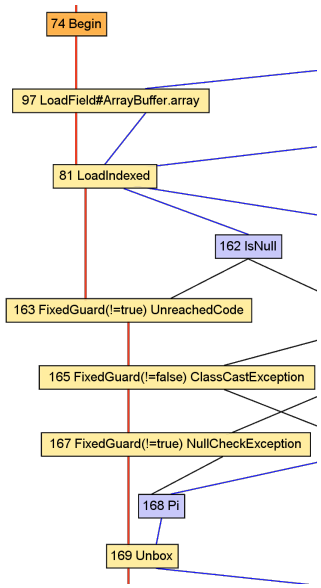


Figure 10. Reading from an array in the `ArrayBuffer.map`

Gb. We used standard evaluation practices for the JVM – we ran each benchmark until detecting a steady state, and then reported the mean value across several iterations [Georges et al. 2007].

Fig. 12 shows the performance comparison between Graal and the Hotspot C2 compiler. The first nine plots show collection microbenchmarks, and they include an equivalent,

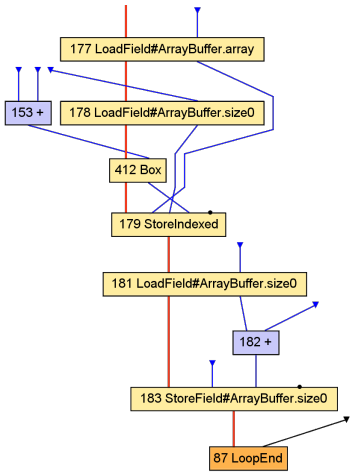


Figure 11. Writing to an array in the `ArrayBuffer.map`

manually optimized version of the code (we take care to avoid boxing, polymorphic dispatches and iterators). The last four plots are programs that heavily use collections, where we only compare Graal and C2 on the same program.

The case studies from Section 4 are shown in the first three plots of Fig. 12. The variants of `foldLeft` (on a primitive `Int` array) with the non-polymorphic and the polymorphic callsite are both 17× faster when run with Graal compared to C2. Graal's performance on this microbenchmark is equal to a manually written loop over an integer array. As explained in Section 4.3, the `map` snippet (on an `ArrayBuffer[Int]`) cannot avoid boxing, and here Graal is only 2.4× faster than C2. Our hand-optimized version uses `Int` arrays and avoids boxing, making it 7.8× faster than Graal. The difference is less pronounced on the `filter` microbenchmark, since filtering does not create new boxed objects (it only reuses the ones from the original collection). The `groupBy` benchmark materializes a lot of intermediate data structures, so our hand-optimized version is only 1.4× faster than the collections version compiled with Graal. The `copyToArray` (on an `ArrayBuffer`) benchmark is 3.9× faster on a hand-optimized version that uses `Int` arrays – the fast intrinsification of the `System.arraycopy` method is possible for raw integer arrays, but not when copying an object array to an integer array.

The `foreach(sum+=_)` benchmark computes the same result as the `foldLeft` benchmark. Graal uses PEA and scalar replacement to eliminate the `IntRef` object captured by the lambda, and produces optimal code. On the `find` and *standard deviation* benchmarks, Graal is 29× and 7.7× faster than C2, and close to optimal, with a few suboptimal loop code patterns (which we are currently working on to fix).

We also tested Graal on several larger collection-based programs – K-means, character histogram, phone code mnemonics, and CafeSAT (SAT solver implementation) [Blanc 2015], and we observed performance improvements with Graal ranging from 1.1× to 3.7×, compared to the C2 compiler.

We also compared Graal and C2 on several Apache Spark workloads, as shown in Fig. 13. Spark has a similar API as Scala collections, but also a similar implementation. From inspecting the Apache Spark source code, we concluded that Spark's `RDD` data types have implementations similar to Scala collections – an RDD operation typically uses an iterator object with an unknown implementation, traverses elements whose type is generic, and executes lambda objects whose implementation is unknown.

Accumulator, *Char-Count*, *TextSearch* and *WordCount* are Spark RDD data-processing microbenchmarks, which compute a sum of numbers, count the characters, search for a pattern in the text, and count the words, respectively. Here, Graal is consistently 1.2 – 1.3× faster than C2. The larger applications, *Chi-Square*, *Decision-Tree*, *Gaussian-Mix* and *PCA*, are standard Spark MLlib statistics and machine-learning workloads. We found that Graal is 1.2 – 1.8× faster than C2.

6 Related Work

Most of the high-level optimizations mentioned in this paper were already treated in earlier works. Escape analysis [Kotzmann and Mössenböck 2005] replaces unnecessary allocations with raw values, and partial escape analysis [Stadler et al. 2014] improves standard escape analysis by delaying the allocation into the control flow segment in which that allocation escapes. Inlining was studied extensively by many researchers in the past [Arnold et al. 2000; Ayers et al. 1997; Ben Asher et al. 2008; Chang et al. 1992]. One alternative approach to inlining is specializing classes based on profile information and type constraints [Sutter et al. 2004]. Global value numbering [Click 1995] is one of the basic optimizations used in SSA-based compilers, and many optimizations used in Graal's canonicalization are standard. Although compiler optimizations were studied extensively in the past, to our knowledge, very little work currently exists on how to optimally compose optimizations for concrete workloads, and this prompted us to share our insights.

Aside from an optimal code, program performance depends heavily on correct choice of data structures. One of the major data structure performance impacts on the JVM is implicit *boxing* due to type erasure. Due to its design decisions, this effect is more pronounced in Scala than it is in Java. While some concurrent data structures, such as Scala lock-free Ctries [Prokopec et al. 2012], FlowPools [Prokopec et al. 2012], or lock-free SnapQueues [Prokopec 2015], as well as some functional data structures such as Conc-Trees [Prokopec and Odersky 2016] and RRB Vectors [Stucki et al. 2015], depend on object allocation and pay the price of heap allocation in either case, performance of most sequential and lock-based concurrent data structures can be improved by avoiding boxing. Ever since erasure-based generics were

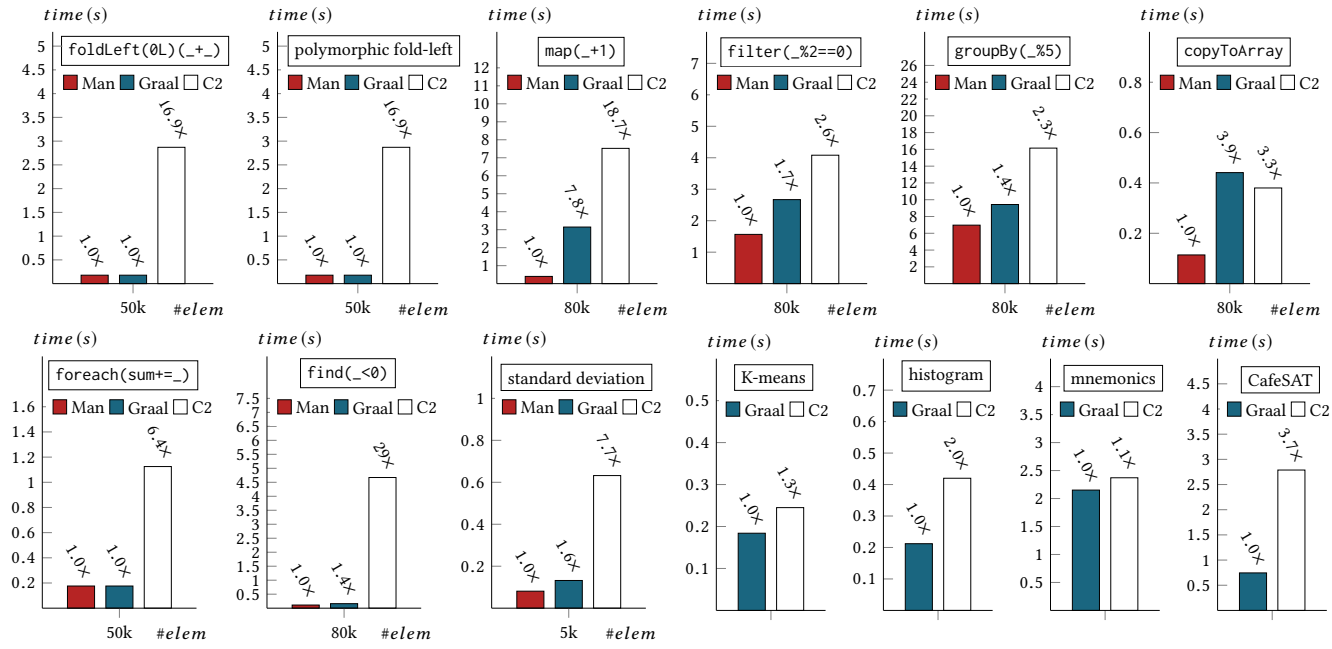


Figure 12. Collections running time comparison between Graal, HotSpot C2 and manually optimized code (lower is better)

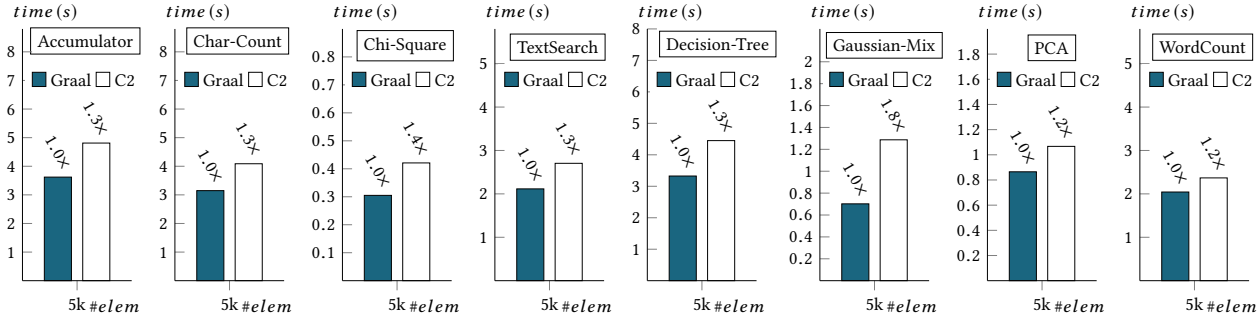


Figure 13. Apache Spark running time comparison between Graal and HotSpot C2 (lower is better)

added to Java 1.5 [Bracha et al. 2003], many mainstream languages, such as C#, Scala and Kotlin, decided to follow in its footsteps. While the .NET runtime, which C# runs on, supports runtime type reification (i.e. allows data structure specialization in the runtime), most JVM languages pay some performance overhead for boxing. Aside from type reification, here have been various attempts at reducing these costs. The DartVM uses pointer tagging to encode small integers within the pointer, but this optimization is limited to integers within a specific range. Scala type specialization [Dragos and Odersky 2009] and miniboxing [Ureche et al. 2013] are promising solutions that were used in the creation of several collection frameworks [Prokopec et al. 2014, 2015], their main downside being a large output code size.

In the past, the Graal compiler was evaluated on Scala workloads, specifically on the DaCapo and Scala DaCapo benchmark suites [Stadler et al. 2013]. At the time, Graal performance was slightly behind C2. Since that evaluation, Graal was extensively improved and it now beats C2 on most benchmarks that we are aware of.

7 Conclusion

The performance results reported in Section 5 represent an opportunity that is, at least currently, unexploited by the standard HotSpot C2 compiler. JIT compilers generally have access to branch frequency profiles, receiver type profiles and loop frequencies, so they are in principle capable of easily performing the same kind of (speculative) optimizations as those described in this paper. In fact, the case study from Section 4.2 should be a convincing argument of why this type of optimizations is best left to the JIT compiler. Speculation based on type profiles is a cheap (and potentially more accurate) alternative to whole program analysis. And if a speculation turns out wrong, deoptimizing the code segment

is simple. For example, if the `foldLeft` callsite from Section 4.2 encounters a new receiver type, Graal simply recompiles the enclosing method.

Speculating on the data structure invariants is generally expensive. First, reasoning about a data structure requires analyzing all its usages, which is usually outside of the scope of a single method. JIT compilers tend to avoid this, since they have a limited time budget available for compilation. For example, in the case study from Section 4.3, to know that a `null` value is never assigned to an `ArrayBuffer`, the compiler would need to analyze all usages of the `+=` method – the entire program. To avoid boxing the integers stored into an `ArrayBuffer`, a JIT compiler would similarly have to ensure that all usages of the `apply` method are followed by unboxing. Second, if the speculation turns out inaccurate, the deoptimization may involve modifying the existing data structure instances. Aside from shape analysis for dynamic languages, such as V8 or Truffle [Würthinger et al. 2012], and automatic object inlining [Wimmer 2008], we are not aware of many optimizations that speculate on the invariants of a data structure. In this regard, we see past work on static metaprogramming [Burmako 2013], monomorphization [Biboudis and Burmako 2014] and generic type specialization in Scala [Dragos and Odersky 2009] as very valuable, since it enables static data structure specialization – we encourage future research in this direction.

To conclude, in the context of the current VM technology, we leave the reader with the following two maxims:

- First, a JIT compiler is generally in a better position to do code optimizations compared to a static compiler. By focusing on the optimizations from this paper, the language designer may be doing a duplicated effort.
- Second, a JIT compiler (particularly on the JVM) is generally less focused on optimizing data structures. A high-level language designer may want to focus efforts on statically optimizing the language data structures or exposing primitives that allow users to do so.

We hope that these insights will be useful in the design of the languages that target runtimes with JIT compilation.

References

Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F. Sweeney. 2000. A Comparative Study of Static and Profile-based Heuristics for Inlining. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (DYNAMO '00)*. ACM, New York, NY, USA, 52–64. https://doi.org/10.1145/351397.351416

Andrew Ayers, Richard Schooler, and Robert Gottlieb. 1997. Aggressive Inlining. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI '97)*. ACM, New York, NY, USA, 134–145. https://doi.org/10.1145/258915.258928

Yosi Ben Asher, Omer Boehm, Daniel Citron, Gadi Haber, Moshe Klausner, Roy Levin, and Yousef Shajrawi. 2008. *Aggressive Function Inlining: Preventing Loop Blockings in the Instruction Cache.* Springer Berlin Heidelberg, Berlin, Heidelberg, 384–397. https://doi.org/10.1007/978-3-540-77560-7_26

Aggelos Biboudis and Eugene Burmako. 2014. MorphScala: Safe Class Morphing with Macros. In *Proceedings of the Fifth Annual Scala Workshop (SCALA '14)*. ACM, New York, NY, USA, 18–22. https://doi.org/10.1145/2637647.2637650

Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2014. Clash of the Lambdas. *CoRR* abs/1406.6631 (2014). http://arxiv.org/abs/1406.6631

Richard Bird and Philip Wadler. 1988. *An Introduction to Functional Programming.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK.

Regis Blanc. 2015. CafeSAT. (2015). https://github.com/regb/cafesat.

Gilad Bracha, Sun Microsystems, Norman Cohen Ibm, Christian Kemper Inprise, Martin Odersky Epfl, David Stoutamire, and Sun Microsystems. 2003. *Adding generics to the java programming language: Public draft specification, version 2.0.* Technical Report.

Eugene Burmako. 2013. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming. In *Proceedings of the 4th Workshop on Scala (SCALA '13)*. ACM, New York, NY, USA, Article 3, 10 pages. https://doi.org/10.1145/2489837.2489840

Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-mei W. Hwu. 1992. Profile-guided Automatic Inline Expansion for C Programs. *Softw. Pract. Exper.* 22, 5 (May 1992), 349–369. https://doi.org/10.1002/spe.4380220502

Cliff Click. 1995. Global Code Motion/Global Value Numbering. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI '95)*. ACM, New York, NY, USA, 246–257. https://doi.org/10.1145/207110.207154

Cliff Click and Michael Paleczny. 1995. A Simple Graph-based Intermediate Representation. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations (IR '95)*. ACM, New York, NY, USA, 35–49. https://doi.org/10.1145/202529.202534

L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '84)*. ACM, New York, NY, USA, 297–302. https://doi.org/10.1145/800017.800542

Iulian Dragos and Martin Odersky. 2009. Compiling Generics Through User-directed Type Specialization. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS '09)*. ACM, New York, NY, USA, 42–47. https://doi.org/10.1145/1565824.1565830

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, New York, NY, USA, 187–193. https://doi.org/10.1145/2647508.2647521

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, New York, NY, USA, 1–10. https://doi.org/10.1145/2542142.2542143

Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 57–76. https://doi.org/10.1145/1297027.1297033

Thomas Kotzmann and Hanspeter Mössenböck. 2005. Escape Analysis in the Context of Dynamic Compilation and Deoptimization. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE '05)*. ACM, New York, NY, USA, 111–120. https://doi.org/10.1145/1064979.1064996

Martin Odersky and Adriaan Moors. 2009. Fighting bit Rot with Types (Experience Report: Scala Collections). In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*

(FSTTCS 2009) (Leibniz International Proceedings in Informatics (LIPIcs)), Ravi Kannan and K Narayan Kumar (Eds.), Vol. 4. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 427–451. https://doi.org/10.4230/LIPIcs.FSTTCS.2009.2338

Simon Peyton Jones and Simon Marlow. 2002. Secrets of the Glasgow Haskell Compiler Inliner. *J. Funct. Program.* 12, 5 (July 2002), 393–434. https://doi.org/10.1017/S0956796802004331

Aleksandar Prokopec. 2015. SnapQueue: Lock-Free Queue with Constant Time Snapshots *(Scala '15)*. https://doi.org/10.1145/2774975.2774976

Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. 2011. A Generic Parallel Collection Framework. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II (Euro-Par'11)*. Springer-Verlag, Berlin, Heidelberg, 136–147. http://dl.acm.org/citation.cfm?id=2033408.2033425

Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. 2012. Concurrent Tries with Efficient Non-blocking Snapshots. (2012), 151–160. https://doi.org/10.1145/2145816.2145836

Aleksandar Prokopec, Philipp Haller, and Martin Odersky. 2014. Containers and Aggregates, Mutators and Isolates for Reactive Programming. In *Proceedings of the Fifth Annual Scala Workshop (SCALA '14)*. ACM, 51–61. https://doi.org/10.1145/2637647.2637656

Aleksandar Prokopec, Heather Miller, Tobias Schlatter, Philipp Haller, and Martin Odersky. 2012. FlowPools: A Lock-Free Deterministic Concurrent Dataflow Abstraction. In *LCPC*. 158–173.

Aleksandar Prokopec and Martin Odersky. 2016. *Conc-Trees for Functional and Parallel Programming*. Springer International Publishing, Cham, 254–268. https://doi.org/10.1007/978-3-319-29778-1_16

Aleksandar Prokopec and Dmitry Petrashko. 2013. ScalaBlitz Documentation. (2013). http://scala-blitz.github.io/home/documentation/

Aleksandar Prokopec, Dmitry Petrashko, and Martin Odersky. 2014. *On Lock-Free Work-stealing Iterators for Parallel Data Structures*. Technical Report.

Aleksandar Prokopec, Dmitry Petrashko, and Martin Odersky. 2015. Efficient Lock-Free Work-Stealing Iterators for Data-Parallel Collections. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 248–252. https://doi.org/10.1109/PDP.2015.65

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, New York, NY, USA, Article 9, 8 pages. https://doi.org/10.1145/2489837.2489846

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, New York, NY, USA, Article 165, 10 pages. https://doi.org/10.1145/2544137.2544157

Nicolas Stucki, Tiark Rompf, Vlad Ureche, and Phil Bagwell. 2015. RRB Vector: A Practical General Purpose Immutable Sequence. *SIGPLAN Not.* 50, 9 (Aug. 2015), 342–354. https://doi.org/10.1145/2858949.2784739

Bjorn De Sutter, Frank Tip, and Julian Dolby. 2004. Customization of Java Library Classes Using Type Constraints and Profile Information. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*. 585–610. https://doi.org/10.1007/978-3-540-24851-4_27

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, New York, NY, USA, 73–92. https://doi.org/10.1145/2509136.2509537

Christian Wimmer. 2008. *Automatic Object Inlining in a Java Virtual Machine*. Trauner.

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, New York, NY, USA, 73–82. https://doi.org/10.1145/2384577.2384587

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301

View publication stats