Interflow: Interprocedural Flow-Sensitive Type Inference and Method Duplication

Denys Shabalin
EPFL
denys.shabalin@epfl.ch

Martin Odersky
EPFL
martin.odersky@epfl.ch

Abstract

Scala heavily relies on a number of object-oriented abstractions to support its feature-rich collections library. There are known techniques that optimize those abstractions away in just-in-time (JIT) compilers, but applying them in the ahead-of-time (AOT) setting is problematic. Profile-guided optimization (PGO) lets AOT compilers apply some of the same optimizations that JIT compilers employ, but it comes at a high complexity cost.

In this paper, we introduce Interflow, an alternative approach towards ahead-of-time optimization of Scala programs which relies on interprocedural flow-sensitive type inference and method duplication. Our evaluation shows that an Interflow-based optimizing compiler built on top of the Scala Native toolchain outperforms existing PGO-based optimizing compilers for Scala.

Moreover, we demonstrate that Interflow and PGO can be combined to achieve further improvements. On our benchmarks, with both Interflow and PGO enabled, the Scala Native toolchain approaches the performance of the HotSpot JVM.

CCS Concepts • **Software and its engineering → Compilers**;

Keywords Ahead-of-time compilation, profile-guided optimization, flow-sensitive optimization, interprocedural optimization.

1 Introduction

The Scala programming language is well-known for its collections library [20] [23]. Collections make use of many features of the Scala programming language such as generics, closures, trait system, value classes, implicit parameters and implicit conversions to name a few. At the same time, the current implementation is known to incur significant performance overheads compared to their Java equivalents [4].

The main body of research around Scala collections performance [9] [21] [29] [24] has been focusing on improving collections performance in just-in-time (JIT) compiled environments such as HotSpot JVM [13] and GraalVM [30]. Modern JIT compilers take advantage of their ability to collect profile feedback and speculatively optimize code to achieve the best peak performance [2].

In comparison, ahead-of-time (AOT) compilers such as Scala Native [25] or Graal Native Image have not been able to reach the same level of performance. Unlike JIT compilers, an AOT compiler does not have the luxury of profiling the application at runtime and requires the use of a multi-step build process that is known as profile-guided optimization (PGO) to obtain profiles based on a simulated workload and recompile the application with those profiles in mind. Apart from the inevitable added build complexity, PGO does not guarantee that profiles collected during the build process are going to be representative of typical application workload. The profile-guided decisions are hardcoded into the resulting binary and may not change later at runtime.

In this paper, we present a design of an optimizing AOT compiler that uses flow-sensitive static type information to optimize Scala code. Furthermore, we study an implementation of this design on top of the Scala Native toolchain. Our performance evaluation demonstrates that, even without profile feedback, our implementation significantly improves upon its AOT counterparts and, when combined with PGO, closely approaches the performance levels of state-of-the-art JIT compilers.

The main contributions of this paper are:

- A design of a single pass optimizer that fuses five well-known techniques in a single forward-dataflow traversal of the whole program: flow-sensitive type inference, method duplication, partial evaluation, partial escape analysis and inlining. To the best of our knowledge, our work is the first to put these techniques together in a single-pass that exploits the synergies between these techniques to amplify their performance improvements. (Section 3)
- An implementation of three profile-guided optimizations for Scala Native: polymorphic inline caching, untaken branch pruning and profile-directed code placement. (Section 4)
- An experimental evaluation that compares static and profile-guided approaches to optimization in AOT setting. (Section 5)

2 Motivation

Scala heavily relies on virtual dispatch to support its feature-rich collection library. A typical example is the `map` method defined in the `TraversableLike` trait (Figure 1).

Apart from the natural polymorphism on the element type, this method is simultaneously polymorphic in:

```
def map[B, That]
  (f: A => B)
  (implicit bf: CanBuildFrom[Repr, B, That])
  : That = {
  def builder = {
    val b = bf.apply(repr)  // virtual call
    b.sizeHint(this)        // virtual call
    b
  }
  val b = builder
  this.foreach {            // virtual call
    x =>                    // 1 virtual call per iteration
      b.+=(f.apply(x))      // 2 virtual calls per iteration
  }
  b.result                  // virtual call
}
```

Figure 1. Definition of the map method in Scala collections.

1. The collection type of `this`. The same method implementation is used for most descendants of this trait (e.g., wrapped arrays, vectors, maps, sets). Based on the collection type, `foreach` dispatches with a new closure that wraps `f` and appends its result to the collection builder `b`.
2. In the operation `f` that is being applied to every element of the collection. Scala's closures are compiled as anonymous classes that extend `FunctionN` trait (where N is from 0 to 22 depending on the number of parameters) trait that has an `apply` method that's used to invoke a closure.
3. In the builder for the resulting collection through the `bf` argument. This parameter is implicit and is typically inferred by the compiler [20]. Nevertheless, users may still provide a custom instance of `CanBuildFrom` to produce a different collection type as the result of the `map` combinator.

Prokopec et al. [24] studied techniques that a state-of-the-art JIT compiler such as GraalVM is going to use to optimize this method:

1. Use Class Hierarchy Analysis [8] to check if methods such as `foreach` and closure's `apply` are never overridden. For our running example, this check would inevitably fail as `foreach` is overridden for every collection (dozens of methods) and `apply` is overridden for every closure (hundreds of methods).
2. Collect type profile on all of the three sources of polymorphism: `this`, `f` and `bf` local variables in this method.
3. If a type profile shows a few dominant types, the compiler will *speculatively* optimize this method assuming it is only used for dominant types. In that case, all of the virtual calls are going to be turned into static calls behind a guard that verifies that optimization assumptions are correct (i.e., the method is used only for the types that it was compiled for).
4. Otherwise, calls will be considered megamorphic, and all of the calls are going to be compiled as virtual calls based on table lookups. In this case, the optimizer cannot inline them, and the compiled code is going to incur a performance penalty.

In case of a megamorphic type profile, these techniques cannot optimize a given method *in isolation*. Megamorphic virtual calls are optimization barriers that prevent optimizations across the call boundary.

Inlining the whole `map` method and all of its transitive dependencies is the last resort to optimize such combinators. However, inlining is based on heuristics, so it is not guaranteed to succeed (for example if the caller is already big and does not have enough size budget to fit the result of the inlining).

In summary: the problem of optimizing away multiple nested layers of virtual dispatch has not been fully solved yet. While there exist techniques that can be used towards that end, they rely on heuristics that might fail to optimize invocations of highly polymorphic combinators in Scala collections.

3 Interflow

We designed Interflow with the primary goal of static devirtualization in mind. Specifically, we are interested in having collection combinators (such as `map`) in the Scala standard library to never pay the cost of the megamorphic virtual dispatch.

At its core, Interflow is a single pass optimizer that fuses five techniques: method duplication, flow-sensitive type inference, partial evaluation, partial escape analysis and inlining.

3.1 Intuition

The fundamental idea behind Interflow's approach to devirtualization is method duplication guided by whole-program flow-sensitive type inference.

Rather than attempting to create a single optimized version of a collection combinator such a `map`, Interflow duplicates it per context. For example, consider multiple calls to `map` with multiple different closures:

```
val arr = Array(1, 2, 3)
arr.map(_ + 1)

val vec = Vector(1, 2, 3)
vec.map(_ * 2)
```

With Interflow, these two calls will be transformed to use duplicates of `map` that are specialized for a combination of the exact collection type, the exact anonymous class used to map elements and exact builder for the resulting collection:

```
val arr = Array(1, 2, 3)
arr.`map<WrappedArray.ofInt, AnonFun1, ArrayCBF>`(_ + 1)

val vec = Vector(1, 2, 3)
vec.`map<Vector, AnonFun2, VectorCBF>`(_ * 2)
```

The duplicate versions of map will have more precise parameter types than the ones provided in the original version. The process will start over within each duplicate, and the virtual calls to foreach, += and apply are going to be statically routed to the corresponding implementations guided by precise types of the arguments.

The resulting code will be completely free of virtual calls on the hot path. In turn, a combination of inlining, partial evaluation, and partial escape analysis is going to optimize it further by removing intermediate closure and box allocations and avoiding allocation of the builder altogether.

It is important to highlight that the success of devirtualization here *does not depend on the inlining of the calls to map*. If map is not inlined, Interflow will have to allocate the outer closure, but all of the dispatch within the map is still going to be static.

3.2 Intermediate representation

Interflow relies on a typed object-oriented intermediate representation in SSA form [7].

Method bodies are represented as a sequence of basic blocks. Similarly to Swift Intermediate Language [1], we use basic block parameters rather than phi instructions. Every basic block contains a sequence of static assignments which is always followed by a single terminator which defines control-flow transition either to the successor basic blocks (conditional or unconditional jumps) or outside of the enclosing method (return or throw).

Static assignments contain an operation, and an optional unwind handler (target exception handling basic block in case operation throws an exception). Operations may take values (v) and types (T) as their arguments.

Values consist of canonical constants for all primitive data types (c), references to locals (l) and globaly unique references (n) that may refer to class, module, trait, method and field names.

For brevity, we omit details on primitive operations. They include binary arithmetic operations, comparisons, and conversions between primitive data types and generally have the same semantics as the corresponding operations in JVM bytecode [16].

Additionally, we omit operations and types that are necessary for interoperability with C code. In this paper, we focus purely on optimizing away object-oriented features such as classes, modules, arrays and operations on them.

```
BasicBlock ::=
  label l0(l1: T1, ...)       block label
  Let1, ...                   static assignments
  Terminator                  block terminator

Let ::=                       static assignment
  l  = Op                     let
  l1 = Op unwind l2           let unwind

Op ::=                        operations
  prim v1, ...                primitive op
  module n                    module access
  class-alloc n               class allocation
  field-load v, n             read from n
  field-store, v1, n, v2      write to n
  static-call n(v1, ...)      statically call n
  virtual-call v0, n(v1, ...) dynamically call n
  is-instance-of T, v         instance check
  as-instance-of T, v         cast
  array-alloc T, v            array allocation
  array-length v1             array length
  array-load v1, v2           read element
  array-store v1, v2, v3      write element

Terminator ::=                terminators
  jump l(v1, ...)             unconditional jump
  if v                        conditional jump
    then l1(v11, ...)
    else l2(v12, ...)
  return v                    return value v
  throw v                     throw value v
```

Figure 2. Interflow's instruction set.

```
T ::=                   types
  PrimT                 primitive type
  RefT                  reference type
  QualRefT              qualified reference type

RefT ::=                reference type
  class n               instance of class n
  module n              instance of module n
  trait n               instance with trait n
  array T               instance of array of T

QualRefT ::=            qualified reference type
  exact RefT            exact reference
  nonnull RefT          nonnull reference
  exact nonnull RefT    exact nonnull reference
```

Figure 3. Interflow's types.

3.3 Types and qualifiers

Interflow relies on an erased object-oriented type system (Figure 3).

Types are split into primitive values and reference types. Primitives represent built-in data types such as fixed-width integer, floating point numbers, and unit. Primitives may not be used interchangeably with reference types and must be explicitly boxed whenever a value escapes to a generic location.

References may refer to either classes, modules, traits or arrays. Similarly to the JVM bytecode, arrays are the only type whose generic information is preserved. Otherwise, the type system is erased and does not preserve information about source-level generics for any other data structures. Module types correspond to instances of Scala's singleton objects that can have at most one globally unique instance.

Reference types can be qualified with `exact` and `nonnull` qualifiers. Exact references are used to distinguish class references that may only point to the given class but not to any of the subclasses. Non-null references guarantee that the reference is dereferenceable. Qualifiers are not present in the unoptimized intermediate representation and may only appear as result of type inference in Interflow. We rely on type qualifiers to aid partial evaluation in the removal of instance checks and virtual calls.

3.4 Method duplication

Interflow is built around the framework of graph-free dataflow traversal initially introduced by Mohnen [18]. Similarly to Julia [3] we extend the original framework to support interprocedural type inference with support for mutually recursive functions.

We visit every method reachable from the application's entry points in forward dataflow order. Methods are visited exactly once per *signature*, a sequence of inferred parameter types. Original parameter types are only used for the entry points and are otherwise discarded.

Every single signature corresponds to a new *duplicate* of a method. Not all created duplicates survive Interflow since inlining may make some of them unreachable. Signatures do not contain return types; they are inferred through type inference within the method body.

Method duplicates might form cyclic dependencies that prevent their return types from being inferred without doing a fixpoint computation across the cycle. We perform cycle detection by maintaining a stack of currently visited methods. If a signature is visited a second time (i.e., it is part of a cycle), Interflow bails out and uses the original declared return type instead to unblock methods that depend on it.

Every duplicate goes through a unified forward dataflow traversal of fused type inference, partial evaluation, partial escape analysis and inlining. The fused traversal maintains a single flow-sensitive state (Figure 4) that is shared between all of the optimizations.

```
State ::=                      flow state
  (Locals, Heap)

Locals ::=                     locals state
  Empty                        empty state
  Locals, l -> LocalState      local state of l

LocalState ::=                 local state
  opaque l: T                  opaque value of T
  constant c                   constant value
  virtual k                    virtual object

Heap ::=                       heap state
  Empty                        empty heap
  Heap, k -> ObjectState       object allocation

ObjectState ::=                allocation state
  virtual T, LocalState1, ... virtual object
  escaped l: T                 escaped object
```

Figure 4. Interflow's flow-sensitive state.

The Interflow state is updated based on the current static flow of the program and can change after every visited instruction. Basic block starts in a state that *merges* states of its predecessors.

3.5 Flow-sensitive type inference

The goal of type inference is to infer and propagate type information from the entry point to the leaves of the program. Typing of most of the operations is trivial:

1. Primitive operations such as binary operations, conversions and comparisons produce an opaque value of the corresponding primitive type.
2. Field loads produce a value of the declared type.
3. Array loads produce a value of the element type.
4. Array length produces a value of a primitive numeric type.
5. Memory stores produce a value of primitive unit type.
6. Class, module, and array allocations produce a new exact non-null reference of the corresponding reference type.
7. We discard return types of methods that are called statically. Instead, we compute the types for all of the arguments based on current flow state and create a new duplicate for the callee method. Type inference for the current method is suspended until the duplicate is visited and its inferred return type becomes available.
8. Virtual methods use Class Hierarchy Analysis [8] to compute all possible targets under the closed world

assumption and add them as additional entry points for method duplication. We use the original declared return type as the result type for such calls.

9. Even though branching instructions do not produce values, visiting them might affect the local state if the branching condition is an instance check or null check. In those cases the local state is refined to include the information obtained from those checks (i.e., either narrow down the type or add `nonnull` qualifier to the reference type).

Interflow's state is flow-sensitive, meaning that based on the subsequent visited instructions, the state of a local variable may change based on the flow of the program.

Due to this, before Interflow visits a block it first needs to merge states of its predecessors:

- For local information to be present in the merged state it has to be present in all preceding states.
- If local state is the same in all preceding states, it remains the same in the merged state.
- If local state differs between multiple preceding states, type inference computes least upper bound of all types in preceding states.

State merging logic resembles handling of control-flow in expression-oriented languages with subtyping, but rather than computing a least-upper-bound of a result value, we compute a least-upper-bound of an intersection of all locals in preceding states. It mirrors the fact that control-flow in our intermediate representation does not produce values, but instead passes them along using basic block parameters.

3.6 Partial evaluation

We use partial evaluation to statically evaluate instructions with results that can be predicted at compile-time. It is especially useful to exclude impossible branches as it improves the precision of the inferred types.

Partial evaluation adds a `constant` value as an additional possible local state. Computing types of constant values is trivial.

Before type inference concludes that a state of a local is an opaque typed value, partial evaluation attempts to reduce instruction to a constant:

- Primitive operations take up to two primitive values and produce another primitive value. If both inputs are statically known values, the result is also statically known.
- Instance checks and casts can be resolved statically if the type of the receiver statically satisfies the check and is non-null.
- Virtual methods call targets can be resolved statically if the object type is an exact type, or if Class Hierarchy Analysis concludes that they have at most one possible target. It allows type inference to treat such methods as effectively static and duplicate them accordingly.
- Branching is second-class and does not produce a value, but it can also be partially evaluated to perform a direct jump if the branching condition is statically known.

Merge processing is extended to cover constant values:

- If a local has the same constant value in all preceding states, its state is set to that value in the merge state. If the local is passed as a basic block parameter, the parameter is discarded.
- If a local has at least two incoming states that differ, a basic block parameter is required for it. The resulting state is an opaque value with a type of least-upper-bound type of all incoming states.

3.7 Partial escape analysis and scalar replacement

Partial escape analysis (PEA) and scalar replacement [26] are crucial building blocks that extend partial evaluation with the ability to:

1. Elide non-escaping allocations.
2. Fuse multiple load and store instructions into a single store immediately before the object escapes.
3. Partially evaluate operations that rely on object identity.

PEA extends local state with a new state that corresponds to a virtual allocation. Non-escaping virtual allocations are treated as mappings from fields to their local state. This mapping is done through an allocation key (k), which uniquely identifies a virtual allocation within the heap state.

PEA naturally extends partial evaluation described previously with additional handling for virtual objects:

- Class and array allocations produce new virtual values as local states, with the corresponding allocation in virtual heap. All of the fields or array elements are set to corresponding default values.
- Field and array loads on virtual allocations use the corresponding local state for given field or array element.
- Field and array stores on virtual allocations update the corresponding local state for given field with the new state.
- Array length is statically known if an array is a virtual value.
- Comparisons between virtual and concrete objects are statically known (pre-existing runtime object may never have the same identity as not-yet-allocated virtual one).
- Partial evaluation of instructions that require exact non-null types can be performed statically on a virtual object.
- Whenever virtual object escapes, all objects reachable from it, also escape. They are materialized at the escaping location. The escaped objects are typed with corresponding `exact` and `nonnull` reference type. Once

an object escapes, its corresponding state in the virtual heap is changed to `escaped`.

Merge processing is extended to handle not only the local state but also the heap state. This is necessary to obtain a consistent view of the resulting flow state:

1. Only objects that are allocated in all preceding states survive the merge.
2. If object escapes in any of the preceding states it must escape in all of the preceding states. The actual object instance must be passed as an additional basic block parameter as object identity is not statically known any longer.
3. Objects that do not escape must have all of their fields merged. Similarly to locals, the values of the fields might escape in some states and not the others, and therefore extra parameters might be introduced for them.
4. Locals must be handled specially as well. If any of the states pass a virtual object as an argument, this object must be materialized in all preceding states, unless it is the same object.

The process is repeated until no more objects are materialized. Some additional aspects such as loop processing need additional machinery which is not described here. We defer to the original work by Stadler et al. [26] for the details.

3.8 Inlining

Effects of partial evaluation and partial escape analysis are quite limited if they are performed purely based on knowledge about a single method. Inlining is a natural way to increase the scope of those optimizations across method boundaries.

Calls to statically known method targets are considered for inlining. Interflow uses call-site specific information such as current state of every argument value and virtual state of any virtual value that is passed as an argument.

Inlining considers following inlining incentives:

- Method is small. A number of Scala methods do not perform any computation but redirect to another method immediately (e.g., bridge methods).
- Method is a class or module constructor. Constructors often form deep dependency chains due to deep inheritance hierarchy in Scala collections. They typically perform trivial actions such as initializing object fields with given arguments.
- Method is a field getter or setter. Getters and setters abstract away field access to satisfy Scala's uniform access principle but otherwise serve no other purpose.
- Method is explicitly annotated with `@inline`. Interflow respects end-user annotations even if they might produce suboptimal code.
- Method call has arguments that are virtual object values. Emitting such a call would force materialization incurring additional performance cost compared to an inlined call that can delay this allocation further. So we bias inlining decisions to inline calls with virtual arguments whenever possible.

The presence of an incentive is not sufficient to make an inlining decision. Additionally, we ensure that none of the following conditions that prevent inlining, hold:

- Method is explicitly annotated with `@noinline`.
- Inlining a method overflows the inlining budget for the current method.
- Inlined method is recursive. We rely on Scala's built-in tailrec support to transform tail recursion into loops.
- All of the arguments are opaque typed values. We do not need to inline calls where all arguments are opaque as method duplication provides the same optimization context without sacrificing size out of the current inlining budget.

Inlining is performed in a duplicate of current optimizer state. Locals are reinitialized only to include method arguments while heap state is preserved. Reusing the same state model allows us to perform the traversal of the callee method in the same pass as the traversal of the caller.

Once callee traversal is complete, the modified state after the call is a merge of all leaf states in the callee (i.e., states in all of the basic blocks that end with either return or throw instruction). It prevents objects that are returned from the callee to be materialized if they remain virtual in all of the leaf states.

4 Profile-guided Optimization

To complement the evaluation of Interflow, a purely static optimization technique, we additionally implement polymorphic inline caching [10], untaken branch pruning and profile-directed block placement [22].

Profile instrumentation. Our implementation of profile-guided optimization provides instrumentation to collect following data:

- Type profile at every virtual call site. This information is necessary to determine if a given call site has a few dominating types or if it is completely megamorphic.
- Frequency of every basic block. It provides information about which basic blocks are visited and which are never taken. The frequency of the entry basic block can also be used as call count of the given method.

Polymorphic inline caching. PIC is a technique to optimize virtual call sites which appear to be megamorphic from a static compiler's point of view.

Given type profiles, one may observe that it is common for a call site to have a few dominant types at runtime. Using this information, virtual calls are compiled to a sequence of tests for frequent type cases that redirect control flow to a static call before attempting a virtual method lookup.

Performance improvement for this approach comes from the fact that type tests and direct method calls are easier to correctly branch-predict for a modern out-of-order processor than indirect jumps. Additionally, static calls can be inlined directly enabling other optimizations which are not possible across the virtual call boundary.

Our implementation uses the standard treatment for virtual call sites that distinguishes monomorphic, bimorphic and megamorphic call sites. For call sites with two or fewer receiver types, we insert the corresponding type tests and deoptimize if the runtime type does not match any of them. Megamorphic call sites only cache the dominant type that has relatively probability of 0.9 or higher and otherwise fall back to full virtual call as a slow path. Generated type tests are annotated with branch probabilities based on the corresponding type profiles.

Untaken branch pruning. It is common for hot methods to have paths which are never taken. These code paths contribute to code bloat that gets amplified if those methods are aggressively inlined.

Untaken branch pruning uses block frequencies collected from the earlier runs of the application to detect code paths that are never taken. For every basic block that's never visited based on profile feedback, we compute a set of live SSA locals. The body of the basic block is replaced with a tail call to a method that takes all live SSA variables at that location as parameters. This method effectively hoists out the unreachable code path and all of its transitive dependencies.

Optimizations applied to the hoisted out code path are only limited in terms of inlining. We disallow inlining between hoisted out methods and regular methods. Otherwise, we use the same optimization pipeline to compile them as any other method.

Profile-directed block placement. LLVM [15] provides built-in support for profile-guided code placement as long the produced LLVM IR is annotated with branch weights. We emit that information based on basic block frequencies collected in instrumentation phase.

Profile-directed code placement can improve the layout of the hot loops by positioning more likely branches together in the generated machine code.

5 Performance Evaluation

With this evaluation we answer the following questions:

1. What's the speedup of Interflow optimization compared to the baseline Scala Native compiler?
2. What's the added optimization potential of PGO in combination with Interflow?
3. How does Scala Native with Interflow compare to Graal Native Image?

5.1 Environment and methodology

We evaluate our implementation of Interflow based on top of Scala Native 0.3.7 in combination with LLVM 5.0 and the Immix [5] garbage collector.

Interflow is implemented as an additional optimization pass that follows immediately after Class Hierarchy Analysis, but before any of the existing flow-insensitive optimizations.

All benchmarks are executed on a workstation-class Intel i9 7900X CPU. We use the latest LTS release of Java VM 1.8.0-1711-b11 (referred to as JDK8) as a baseline for comparisons with Native Image 1.0-RC1 (EE) and Scala Native 0.3.7. The heap is configured to use 1GB of memory.

The throughput numbers in this paper are based on 50th percentile of single iteration running time after a sufficient number of iterations to obtain stable peak performance. We run every benchmark in an isolated VM instance (or binary) for 1000 in-process iterations after warm-up and 20 independent runs.

5.2 Benchmarks

The majority of benchmarks (Bounce, List, Json, CD, DeltaBlue, Richards) are based on the original benchmarks of Marr et al. [17]. Scala Native's version of these benchmarks uses Scala collections whenever appropriate to replace Java-style looping constructs. The Permute benchmark uses the built-in permutations method of Scala Collections, rather than its own implementation to compute permutations.

Kmeans and Sudoku are exclusively collection benchmarks. Kmeans is the same benchmark as the one studied by Prokopec et al. [24].

Tracer is a raytracer written idiomatic Scala using an object-oriented representation for the Scene that includes the use of Scala collection's lists, ranges, and iteration over them.

Brainfuck is a naive interpreter of the Brainfuck [19] esoteric programming language. It exercises recursive descent parsing and interpretation based on pattern matching of a hierarchy of case classes. The interpreter runs a program that generates the lyrics of a programming folklore song 99 Bottles of Beer [12].

GCBench is a well-known binary trees benchmark by Boehm and Weiser [6] that is used to evaluate the state of garbage collector throughput on Scala Native and Native Image relative to the JDK8.

5.3 Throughput compared to Scala Native

Interflow achieves up to 3.09x speedups over Scala Native baseline compiler with a geometric mean speedup of 1.8x (Figure 5). It is not surprising given that Scala Native relies purely on Class Hierarchy Analysis for its devirtualization decisions which are not sufficient to provide optimal performance as we have discussed earlier.

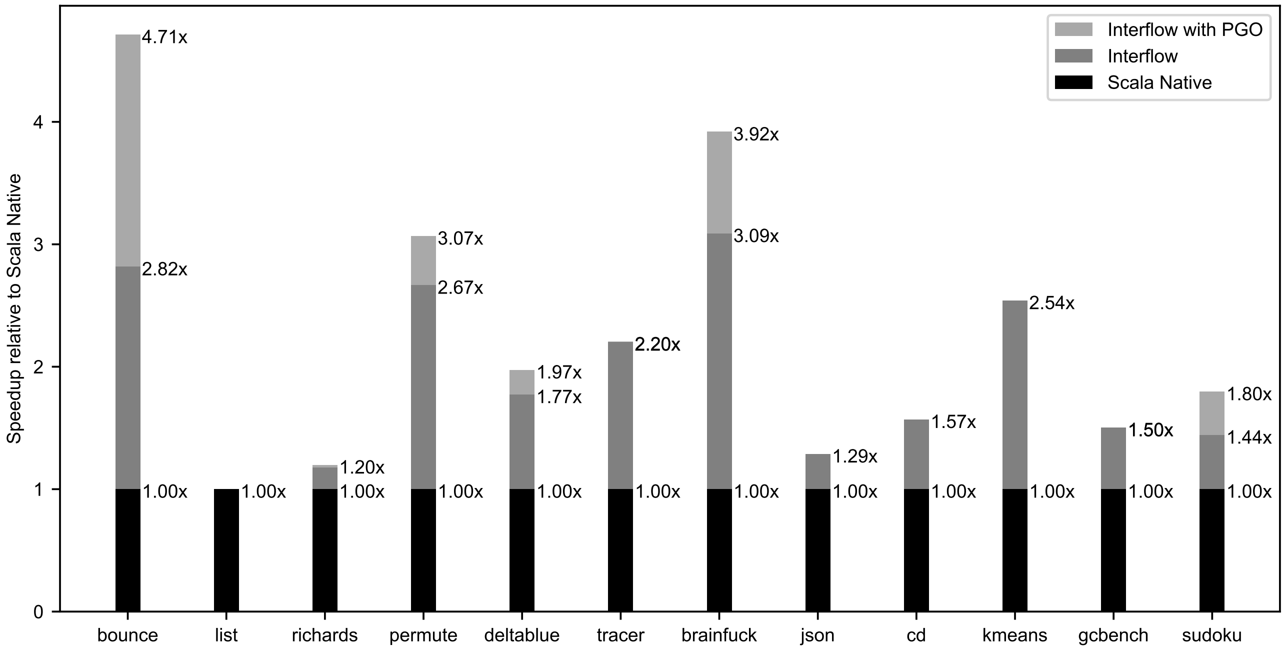

Figure 5. Throughput compared to Scala Native

The benchmarks that benefit most from Interflow are the one using Scala collections most on the hot path: Brainfuck, Bounce, Permute, Kmeans, Tracer, and Sudoku. It illustrates the effectiveness of static devirtualization based on flow-sensitive type inference and method duplication to optimize Scala collections.

PGO adds another geometric mean speedup of 1.09x on top of Interflow. Bounce and Brainfuck get most of its speedup due to improved code layout. DeltaBlue and Sudoku uncover additional devirtualization opportunities which InterFlow could not eliminate statically.

7 out 12 benchmarks show negligible improvements from PGO. In particular, Kmeans and Tracer benchmarks obtain their best results based on Interflow's static optimization despite their heavy use of Scala's collection combinators such as `map`, `foldLeft`, `forall` and `foreach`.

5.4 Throughput compared to Native Image

Both Scala Native and Native Image without PGO fail to achieve performance comparable to the quality of the warmed-up JDK8 with the geometric mean speedup of 0.49x and 0.47x accordingly (Figure 7). Both of the AOT compilers have problems with workloads that rely on Scala collections and can go as low as 4x slower than JDK8 (e.g., Bounce and Brainfuck).

Native Image's implementation of PGO obtains an impressive 1.48x speedup over the baseline Native Image, but it is still 27% behind JDK8. Benchmarks with most speedup correlate closely with the ones where Interflow makes the most impact: Brainfuck, Kmeans, DeltaBlue, Permute, Bounce, and Sudoku.

On the other hand, Interflow achieves 1.8x speedup statically without profile feedback. It has comparable performance to PGO version of Native Image with notable superior results on Bounce, Tracer, Kmeans, and Sudoku. In fact, the results on Kmeans are similar to the ones provided in [24] using aggressive use of JIT optimizations in GraalVM.

With the addition of PGO, Interflow is 1.96x faster than Scala Native and outperforms Native Image on all benchmarks except Richards (9% behind), DeltaBlue (2% behind) and Json (9% behind). Moreover, with the geometric mean of 0.96x relative to JDK8, Interflow with PGO gets close to the throughput of a fully warmed JIT compiler.

Both Native Image and Interflow lag behind the JDK8 in terms of garbage collection throughput based on GCBench results. Neither of the AOT compiler runtimes supports parallel or concurrent garbage collection. Scala Native's implementation of Immix performs favorably compared to classic generational collector used for Native Image.

6 Related Work

Prokopec et al. [24] studies the impact of aggressive JIT compilation on Scala collections. Their work postulates that profiling information enables a JIT compiler to better optimize collection operations compared to a static compiler. Interflow obtains comparable results on collections-heavy benchmarks such as Kmeans *purely ahead of time without any profile feedback*. We still observe performance improvements

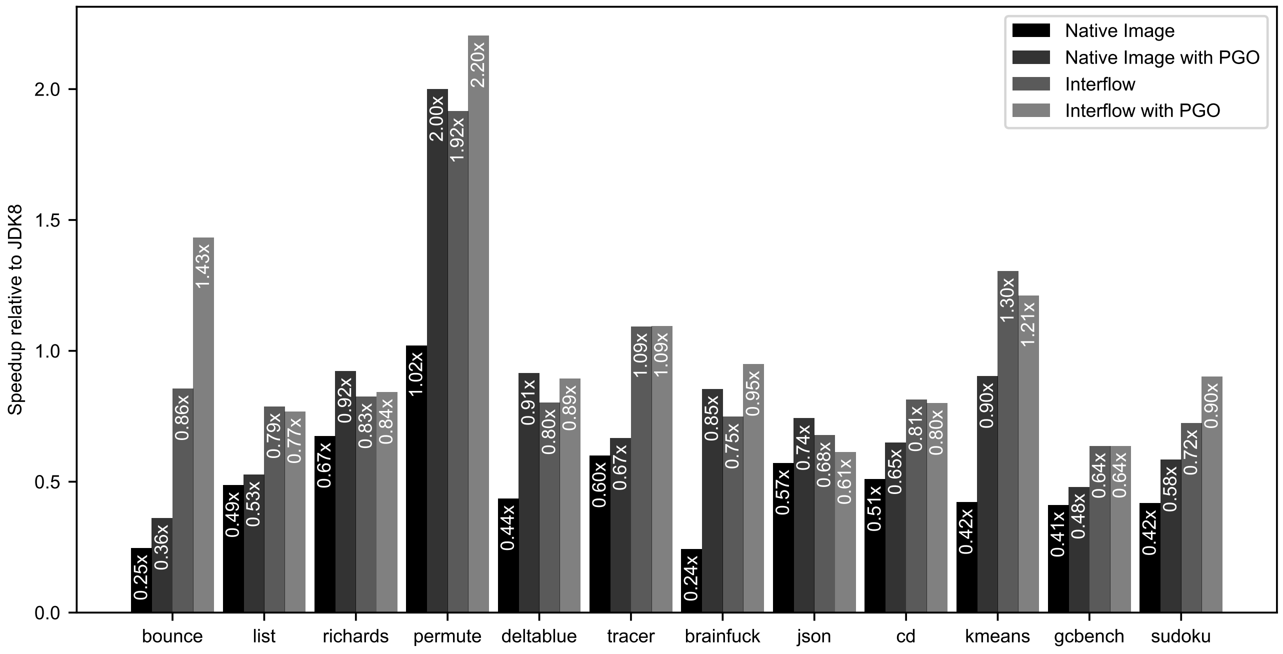

Figure 6. Throughput compared to Native Image, normalized by JDK8.

Implementation	Geometic Mean
Scala Native	0.49x
Native Image	0.47x
Native Image with PGO	0.73x
Interflow	0.89x
Interflow with PGO	0.96x

Figure 7. Geometric mean of the speedup over JDK8

thanks to profile-guided optimization, but our results suggest that it is not essential to many of our benchmarks.

The Julia programming language [3] performs interprocedural type inference to recover types from dynamically typed programs. Both Julia and Interflow visit programs in graph-free forward dataflow manner [18]. Unlike Julia, Interflow does not rely on fixpoint to recover return types of recursive definitions but instead uses original declared type. Both Julia and Interflow duplicate methods to improve performance. Julia relies on heuristics to prune the number of specialized instances that are possible due to its rich type system. Interflow, on the other hand, uses a simpler type system to avoid this problem altogether. In particular, we do not have any form of generic types apart from arrays, while Julia has tuples, union types, and generic composite types.

Languages with specialized-by-default generics such as C# [11] and C++ [28] rely on the end user to annotate programs to guide method duplication. Interflow neither relies on generic information (its type system is erased) nor provides any means for the user to influence method duplication. Specialization is automatic and transparent to the end user. Major C++ implementations such as GCC [27] and Clang [14] expand C++ templates once per use site and later deduplicate generated code at link time. Interflow, similarly to [11], visits each specialized method variant exactly once at link-time, reducing the impact of method duplication on the application compile time.

Scala specialization [9] and Miniboxing [29] extend Scala's erased generics with opt-in annotations to enable code duplication to reduce boxing overhead. Even though they both offer means to perform method duplication, we do not rely on method duplication to address the boxing problem, but rather as means to enable devirtualization, which in turn enables a combination of inlining, partial escape analysis and partial evaluation to eliminate unnecessary intermediate allocations. Both [9] and [29] create a fixed number of variants per each generic argument, leading to an exponential explosion of the number of specialized variants per generic argument. Interflow only creates duplicates that are reachable through forward dataflow from the application entry point at link time.

Petrashko et al. [21] suggests performing auto-specialization of Scala programs based on context-sensitive call graphs. Interflow does not construct call graphs (or any other analysis artifacts for that matter) but instead performs a single graph-free traversal of the original program in forward dataflow order [18]. We trade off accuracy for compilation speed and intentionally use an erased type system that closely mirrors the runtime type system, rather than Scala's rich type system. Our performance results illustrate that Interflow's precision is sufficient to devirtualize typical programs that rely on Scala collections.

Partial escape analysis was initially introduced by Stadler et al. [26]. Interflow builds upon their work by performing this transformation in the same pass as inlining and partial evaluation. PAE enables inlining to take decisions based on current flow-sensitive state at given call site. For example, calls with virtual not-yet-materialized arguments are inlined if possible to delay materialization.

7 Conclusion

In this paper, we presented Interflow, a design for an optimizing compiler that performs type inference, method duplication, partial evaluation, partial escape analysis and inlining in a fused single-pass traversal of the whole program.

We implemented Interflow on top of the Scala Native compiler toolchain. While related work on ahead-of-time compilation for Scala heavily relies on profile-guided optimization, Interflow is based on a purely static approach and achieves superior performance.

Additionally, we implemented three profile-guided optimizations to evaluate their impact on the code produced by Interflow: polymorphic inline caching, untaken branch pruning and profile-directed block placement. We observed that while the impact of PGO is less pronounced than the impact of Interflow, it is still necessary to achieve the best throughput. Moreover, our evaluation shows that results demonstrated by Interflow in combination with PGO closely approach the performance of the HotSpot JVM.

References

[1] [n. d.]. Swift Intermediate Language (SIL). https://github.com/apple/swift/blob/master/docs/SIL.rst. ([n. d.]). Accessed: 2018-06-04.

[2] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. 2004. A Survey of Adaptive Optimization in Virtual Machines. In *PROCEEDINGS OF THE IEEE, 93(2), 2005. SPECIAL ISSUE ON PROGRAM GENERATION, OPTIMIZATION, AND ADAPTATION*.

[3] Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. 2012. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145* (2012).

[4] Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2014. Clash of the Lambdas. *arXiv preprint arXiv:1406.6631* (2014).

[5] Stephen M Blackburn and Kathryn S McKinley. 2008. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *ACM SIGPLAN Notices*, Vol. 43. ACM, 22–32.

[6] Hans-Juergen Boehm and Mark Weiser. 1988. Garbage collection in an uncooperative environment. *Software: Practice and Experience* 18, 9 (1988), 807–820.

[7] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490.

[8] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*. Springer, 77–101.

[9] Iulian Dragos. 2010. Compiling Scala for performance. (2010).

[10] Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '91)*. Springer-Verlag, London, UK, UK, 21–38. http://dl.acm.org/citation.cfm?id=646149.679193

[11] Andrew Kennedy and Don Syme. 2001. Design and implementation of generics for the. net common language runtime. In *ACM SigPlan Notices*, Vol. 36. ACM, 1–12.

[12] Donald E Knuth. 1984. The complexity of songs. *Commun. ACM* 27, 4 (1984), 344–346.

[13] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java HotSpot client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization (TACO)* 5, 1 (2008), 7.

[14] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*. 1–2.

[15] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75.

[16] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2015. The Java (R) Virtual Machine Specification. https://docs.oracle.com/javase/specs/jvms/se8/html/. (2015). Accessed: 2018-06-04.

[17] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-language compiler benchmarking: are we fast yet?. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 120–131.

[18] Markus Mohnen. 2002. A GraphFree Approach to DataFlow Analysis. In *International Conference on Compiler Construction*. Springer, 46–61.

[19] Urban Müller. 1993. Brainfuck–an eight-instruction turing-complete programming language. http://en.wikipedia.org/wiki/Brainfuck. (1993). Accessed: 2018-06-04.

[20] Martin Odersky and Adriaan Moors. 2009. Fighting bit rot with types (experience report: Scala collections). In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 4. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[21] Dmitry Petrashko, Vlad Ureche, Ondřej Lhoták, and Martin Odersky. 2016. Call graphs for languages with parametric polymorphism. *Acm Sigplan Notices* 51, 10 (2016), 394–409.

[22] Karl Pettis and Robert C. Hansen. 1990. Profile Guided Code Positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90)*. ACM, New York, NY, USA, 16–27. https://doi.org/10.1145/93542.93550

[23] Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. 2011. A generic parallel collection framework. In *European Conference on Parallel Processing*. Springer, 136–147.

[24] Aleksandar Prokopec, David Leopoldseder, Gilles Duboscq, and Thomas Würthinger. 2017. Making collection operations optimal with aggressive JIT compilation. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*. ACM, 29–40.

[25] Denys Shabalin. 2015. Scala Native. http://www.scala-native.org. (2015). Accessed: 2018-06-04.

[26] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial escape analysis and scalar replacement for java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 165.

[27] Richard Stallman. 2001. Using and porting the GNU compiler collection. In *MIT Artificial Intelligence Laboratory*. Citeseer.

[28] Bjarne Stroustrup. 2000. *The C++ programming language*. Pearson Education India.

[29] Vlad Ureche, Cristian Talau, and Martin Odersky. 2013. Miniboxing: improving the speed to code size tradeoff in parametric polymorphism translations. *ACM SIGPLAN Notices* 48, 10 (2013), 73–92.

[30] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software.* ACM, 187–204.