Eliminating Abstraction Overhead of Java Stream Pipelines using Ahead-of-Time Program Optimization

ANDERS MØLLER, Aarhus University, Denmark
OSKAR HAARKLOU VEILEBORG, Aarhus University, Denmark

Java 8 introduced streams that allow developers to work with collections of data using functional-style operations. Streams are often used in pipelines of operations for processing the data elements, which leads to concise and elegant program code. However, the declarative data processing style comes at a cost. Compared to processing the data with traditional imperative language mechanisms, constructing stream pipelines requires extra heap objects and virtual method calls, which often results in significant run-time overheads.

In this work we investigate how to mitigate these overheads to enable processing data in the declarative style without sacrificing performance. We argue that ahead-of-time bytecode-to-bytecode transformation is a suitable approach to optimization of stream pipelines, and we present a static analysis that is designed to guide such transformations. Experimental results show a significant performance gain, and that the technique works for realistic stream pipelines. For 10 of 11 micro-benchmarks, the optimizer is able to produce bytecode that is as effective as hand-written imperative-style code. Additionally, 77% of 6 879 stream pipelines found in real-world Java programs are optimized successfully.

CCS Concepts: • **Software and its engineering** → **Compilers**.

Additional Key Words and Phrases: program optimization, static program analysis, Java 8

ACM Reference Format:
Anders Møller and Oskar Haarklou Veileborg. 2020. Eliminating Abstraction Overhead of Java Stream Pipelines using Ahead-of-Time Program Optimization. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 168 (November 2020), 29 pages. https://doi.org/10.1145/3428236

1 INTRODUCTION

Functional programming is no longer a niche programming paradigm. Although classic functional languages may remain mainly of academic interest only, functional language features are being integrated into mainstream languages, most importantly Java. Version 8 of Java was released in 2014 and included features such as lambda functions and the Stream API [Oracle 2014b,c], which enables functional-style processing of data. A 2017 study found that the adoption of lambda expressions is growing, and that they are mostly used for behavior parameterization such as in stream pipelines [Mazinanian et al. 2017]. As a simple example, Figure 1 shows two ways of computing sums of even squares: (a) using traditional imperative-style iteration and mutable state, and (b) using a functional-style stream pipeline. A stream pipeline consists of a source, in this case an array of integers, operations to be performed on the elements of the stream, here `filter` and `map`, and a terminal operation, such as `sum`. The advantages of functional programming are well known; most importantly, once familiar with this paradigm, the declarative style and absence of

Authors' addresses: Anders Møller, Aarhus University, Denmark, amoeller@cs.au.dk; Oskar Haarklou Veileborg, Aarhus University, Denmark, oskar@cs.au.dk.

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.
© 2020 Copyright held by the owner/author(s).
2475-1421/2020/11-ART168
https://doi.org/10.1145/3428236

```java
1 int sumOfSquaresEven(int[] v) {
2   int result = 0;
3   for (int i = 0; i < v.length; i++)
4     if (v[i] % 2 == 0)
5       result += v[i] * v[i];
6   return result;
7 }
```

(a) Imperative style.

```java
8  int sumOfSquaresEven(int[] v) {
9    return IntStream.of(v)
10          .filter(x -> x % 2 == 0)
11          .map(x -> x * x)
12          .sum();
13 }
```

(b) Functional style, using a stream pipeline.

Fig. 1. Two variants of computing sums of even squares.

side-effects tend to make code easier to read and write than the imperative alternatives, especially for more complex computations.

Despite this advance in language and library design, programmers sometimes avoid using streams for performance reasons. The authors of the 2017 study interviewed a developer from the Open Source project Cassandra on adoption of lambda expressions, who mentioned that *"...Unfortunately, we quickly realized that Streams and Lambdas were pretty bad from a performance point of view. Due to this fact, we stopped using them in hot path."* A developer at Oracle working on the HotSpot Java compiler wrote: *"In order to get the full benefit from JDK 8 streams we will need to make them optimize fully"* [Rose 2015]. In 2014, Biboudis et al. [2014] measured the performance of stream APIs in different languages, including Java 8 and Scala, on seven micro-benchmarks that compare stream pipelines with traditional imperative data processing. They found the Java 8 stream implementation to be the most mature with regards to performance, but also that the baseline imperative-style alternatives were much faster. For example, for their benchmark `sumOfSquaresEven`, which performs the computation shown in Figure 1, the stream approach suffered from a 60% performance degradation compared to the baseline implementation. For pipelines that include the `flatMap` operation, the performance overheads were even larger, and a later study shows performance losses that grow quickly in the number of intermediate pipeline operators [Kiselyov et al. 2017].

Interestingly, today – six years after the experiments by Biboudis et al. – their conclusions still hold, despite improvements in compilers and virtual machine technology. We have replicated their study in Java 13 using the OpenJDK Server VM (build 13+33) with default settings on a machine with an Intel i7-8700 @ 4.6GHz processor and 16 GB of memory. The results are shown in Figure 2.[1] As an extreme case, the `megamorphicMaps` benchmark is still around 52× slower when using streams compared to the imperative-style baseline.

One of the strengths of stream pipelines is that it is often easy to switch to parallel processing and thereby exploit modern multi-core CPUs. Although this may reduce the computation time, it is not an ideal solution. It wastes cores, and even for trivially parallelized pipelines, there is typically still a substantial overhead [Biboudis et al. 2014]. Moreover, parallel processing does not work well with stateful stream operations, such as `sorted`, or computations with side-effects.

The problem with abstraction overhead of stream processing is well known also for other programming languages than Java. This has motivated the development of, for example, the *strymonas* library for Scala and OCaml [Kiselyov et al. 2017], LinqOptimizer for C# and F# [Palladinos and Rontogiannis 2014], and ScalaBlitz for Scala [Prokopec and Petrashko 2013], however, those approaches are based on meta-programming capabilities that are not available in Java.

[1] The micro-benchmarks from Biboudis et al. [2014] can be found at https://github.com/strymonas/java8-benchmarks/blob/master/src/main/java/benchmarks/S.java.

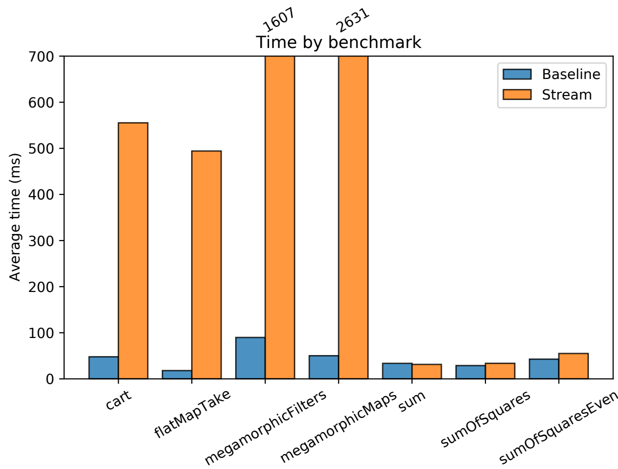


Fig. 2. Performance of baseline imperative implementation versus sequential Java streams.

Since the early versions of Java, the main approach to code optimization has been as part of just-in-time (JIT) compilation in the virtual machine. Few ahead-of-time (AOT) optimizations are performed by `javac`, since it is believed that the JIT optimizer is able to make smarter decisions at run-time based on profiling information. In principle, at run-time the optimizer could be able to deduce that a stream pipeline can be transformed into a for-loop to yield optimal performance, however, the experiments mentioned above show that this is often not the case in practice, even for simple stream pipelines. By manually tuning the HotSpot JIT settings for the `megamorphicMaps` benchmark to inline much more aggressively, we find that a 2× speedup can be obtained, but it is still an order of magniture slower than the baseline. Also, substantial modifications to the JIT settings compared to the defaults can of course degrade performance for other code. Although promising results have been obtained for the Graal JIT compiler on Scala code [Prokopec et al. 2017], AOT optimization techniques have advantages compared to JIT optimizations. First, a JIT compiler has to make fast decisions about what and when to optimize while running the program, whereas an AOT optimizer can be given time to perform more precise whole-program analysis. Second, JIT optimizations are known to be unpredictable, while AOT techniques allow the developer to know before program execution whether an optimization attempt succeeds. Third, new AOT optimizations can be deployed, for example in mobile apps, without requiring modifications to the JVM installations. These observations suggest that it may be time to start pursuing AOT optimization techniques for Java, to reach the full potential of functional-style Java code, most importantly for stream pipelines.

By the use of bytecode-to-bytecode transformations driven by a static program analysis, we combine the best of two styles of programming: the conciseness of functional-style stream pipelines at source-code level, and the efficiency of low-level imperative code at run-time. Among the salient features of our approach are that it does not require adding new Java language features or modifications of the application source code, it does not depend on API-specific knowledge (we demonstrate that it works on both push- and pull-style stream APIs without any adaptation), and it is predictable in the sense that the programmer can be informed ahead-of-time whether optimization succeeds for a given stream pipeline. Furthermore, as the transformations and the static analysis work on Java bytecode, this optimization technique is easy to integrate into existing program development processes.

In summary, the contributions of this paper are:

- We propose an ahead-of-time Java bytecode optimization technique that targets stream pipelines in Java code to make them as efficient as hand-written imperative code. Specifically, we demonstrate that applying a combination of well-known program transformations suffices to reach this goal, most importantly, method inlining and stack allocation.
- We present a static program analysis that simultaneously performs type and pointer analysis for driving the program transformations.
- We report from an experimental evaluation showing that applying the optimization to a suite of 11 micro-benchmarks makes 10 of them as fast as hand-written imperative code, in several cases leading to more than 10× speedup, and that 77% of 6 879 stream pipelines found in real-world Java programs are optimized successfully. The evaluation also demonstrates that the approach is not limited to Java's push-style streams but also works for a pull-style stream API, although with potential for improvements of the static analysis.

2 BACKGROUND: PULL- AND PUSH-STYLE STREAM APIS

Stream APIs can be implemented in two different styles. A *pull-style* stream API follows the iterator protocol, with a method `hasNext` for querying whether the stream has more elements and a method `next` for pulling out the next element from the stream. The iteration through the elements of the stream is controlled by the terminal operation, and each operation in the pipeline thus pulls the elements one at a time from its predecessor.

In a simple pull-style stream API, the `map` intermediate operation, which applies a given function to each element of the stream, can be implemented in Java as shown in Figure 3. The function allocates a new `PullStream` object to represent the intermediate mapping operation, which becomes the new head of the pipeline. When elements are queried from this head, it extracts an element from its predecessor in the pipeline (using `PullStream.this.next` to refer to the method of the outer class) and applies the supplied function to it before it is returned.

A *push-style* stream API instead includes a single method that takes a consumer action to apply to each element in the stream. When executing the stream pipeline, the source operation controls the iteration by pushing every element in the underlying data source to its consumer action until the data source is empty or until the pipeline terminates early (for example, the `findFirst` operation usually does not have to look at all the elements).

```
14 public abstract class PullStream<V> implements Stream<V> { ...
15     public <U> Stream<U> map(Function<? super V, ? extends U> f) {
16         return new PullStream<U>() {
17             protected U next() {
18                 return f.apply(PullStream.this.next());
19             }
20             protected boolean hasNext() {
21                 return PullStream.this.hasNext();
22             }
23         };
24     } ...
25 }
```

Fig. 3. The `map` intermediate operation in a pull-style stream API.

```
26 public abstract class PushStream<V> implements Stream<V> { ...
27     public PushStream<V> filter(Predicate<? super V> p) {
28         return new PushStream<V>() {
29             protected void exec(Consumer<? super V> c) {
30                 PushStream.this.exec(x -> {
31                     if (p.test(x)) c.accept(x);
32                 });
33             }
34         };
35     } ...
36 }
```

Fig. 4. The `filter` intermediate operation in a simple push-style stream library.

In a push-style stream API, the `filter` intermediate operation, which filters out elements that do not satisfy a given predicate, can be implemented as shown in Figure 4. A new `PushStream` object is allocated to form the new head of the pipeline. When we execute this pipeline by calling `exec` on the final stream object, the filtering operation constructs a new consumer and passes it to its predecessor in the pipeline (using `PushStream.this.exec` to refer to the method of the outer class). When this consumer is invoked with an element, it only forwards it to the next consumer if the element satisfies the predicate that was given to the `filter` function.

The stream API in Java's standard library is push-style, whereas Scala's views and C#'s Language-Integrated Queries (LINQ) are pull-style [Biboudis et al. 2014].

The Java stream implementation is quite complex. It provides specialized stream pipelines for the `int`, `long` and `double` Java primitives to avoid boxing at run-time, and pipelines can be executed in parallel using multiple threads. Additionally, certain characteristics of pipelines are recorded to perform further optimizations at run-time. For example, in the pipeline `list.stream().sorted() .sorted().collect(Collectors.toList())`, the second sorting operation is skipped, since the library infers that the stream is already sorted after the first one.

One of the main reasons why stream pipelines are less efficient than their imperative-style counterparts is that executing a stream pipeline involves many virtual calls. A Java stream pipeline with N elements and depth K will accumulate up to $N \times K$ virtual calls just to push the elements through the pipeline [Kiselyov et al. 2017]. Our optimization technique builds on the key observation that fully inlining the consumer chain will reduce this to a constant number of calls depending on the stream source. Method inlining (also called inline expansion) is a classic compiler optimization technique that replaces a call with the body of the method being called, with parameters and return value flow properly substituted to preserve the program semantics [Arnold et al. 2000; Detlefs and Agesen 1999].

If we take a deeper look into Java's stream implementation we see that its streams are backed by *spliterators*, which are similar to iterators but support more advanced features, such as splitting a data source into smaller chunks for parallel computation. Every Java class that implements the `Collection` interface inherits a default implementation of a spliterator backed by the collection's iterator implementation. Stream pipelines eventually end up calling the `forEachRemaining` method on the backing spliterator (unless they stop early due to short-circuiting operations such as `findFirst`) to push every element in the collection through a provided consumer function. For the default iterator-backed spliterator, this translates roughly into this code:

```
37 while (it.hasNext()) consumer.accept(it.next());
```

```
39 public class ArrayList<T> extends AbstractList<T> implements ... {
40     transient Object[] elementData;
41
42     @Override
43     public Spliterator<T> spliterator() {
44         return new ArrayListSpliterator(...);
45     }
46
47     final class ArrayListSpliterator<T> implements Spliterator<T> {
48         public void forEachRemaining(Consumer<? super T> action) {
49             int i, hi, mc;
50             Object[] a = elementData;
51             ...
52         }
53         ...
54     }
55     ...
56 }
```

Fig. 5. Excerpt of the spliterator implementation in Java's `ArrayList` class.

This is expensive, as it requires three virtual calls per element, even for a pipeline with zero intermediate operations. Therefore, to achieve better performance, collections can provide their own spliterator implementation. For example, the `ArrayList` class provides an efficient `forEachRemaining` spliterator implementation that is essentially a while loop over the internal array:

```
38 while (index < elementData.length) consumer.accept(elementData[index++]);
```

This means that if we can fully inline the execution of such a stream pipeline, we expose a primitive while loop with a single virtual call per element. If we can furthermore inline calls to the `consumer.accept` method, the code we are left with will resemble a hand-written imperative loop construct.

Inlining a method call is a relatively simple program transformation in itself. The key to be able to inline the relevant calls ahead-of-time is precise type information to enable virtual call resolution. The static analysis we present in Section 5 is designed to provide this information.

One complication to inlining is that spliterators sometimes rely on private fields in the source collection, in which case naively inlining will violate Java's access rules. An example is the spliterator in Java's `ArrayList` class, shown in Figure 5, where the `forEachRemaining` method accesses the field `elementData`, which is package-private (i.e., Java's default access mode). This prevents inlining the method into another package. A similar situation may occur when the default iterator-backed spliterator is used, since the implementations of `hasNext` and `next` in the source collection may also rely on private fields. This is not a concern for any of the most widely used collections in Java's standard library, but it can be an issue for non-standard stream sources. In Section 6 we discuss different options for how to handle these situations.

To be able to study optimization opportunities in a more controlled environment and to present manageable examples in the following sections, we have created a simple implementation of a push-style stream library with an API that is similar to that of Java's streams and also to stream implementations studied in related work [Biboudis et al. 2015]. This library suffers from the same performance issues as Java's standard stream library (see Section 8). Its API is shown in Figure 6. The library additionally supports pull-style streams, which allows us to explore the flexibility of our optimization techniques also for such a fundamentally different kind of stream API than the one in Java's standard library.

```
57 public interface IntStream<StreamT extends IntStream> {
58     /* Intermediate operations */
59     StreamT map(IntUnaryOperator f);
60     StreamT filter(IntPredicate p);
61     StreamT flatMap(IntFunction<? extends StreamT> f);
62     StreamT limit(long maxSize);
63     /* Terminal operations */
64     void forEach(IntConsumer c);
65     int reduce(int initial, IntBinaryOperator r);
66 }
```

Fig. 6. A simple `IntStream` interface, which can be implemented either pull-style or push-style.

Our current focus is on optimizing sequential (i.e., non-parallel) stream pipelines, since those are by far the most common in practice. (In 28 randomly selected open source Java projects from the RepoReapers dataset [Munaiah et al. 2017] that we could build we found 6 879 sequential stream pipelines and 49 parallel stream pipelines. A recent study by Khatchadourian et al. [2020b] confirms this finding.) Still, sequential stream pipelines are often used in multi-threaded applications, so we cannot assume a single-threaded execution environment when designing optimization techniques.

Another crucial observation we can exploit when optimizing stream pipelines is that the entire construction and execution of a typical pipeline take place locally within a single method. Objects of type `Stream` rarely appear as arguments or return values at calls to other methods than those in the stream library, nor are such objects stored in data structures on the heap. We have made a quantitative study of the top 100 Java projects on GitHub to experimentally verify this claim, and found that 93% of calls with streams as parameter or return types are to stream sources, intermediate, or terminal operations, and there is only one field access that involves stream objects per 200 stream operations in the code. Furthermore, all the spliterator objects, consumer objects, and other transient objects are only used internally within the stream operations. This means that all the information stored in these objects can be placed on the stack instead of in the heap, thereby eliminating the object allocations and reducing the need for garbage collection. As for inlining, this optimization, called stack allocation, is widely used and well understood [Choi et al. 1999; Park and Goldberg 1992].

The Java JIT compiler already tries to perform stack allocation,[2] but the escape analysis used in the Java JIT to drive stack allocation is limited by being intraprocedural only. We exploit the fact that stack allocation works even better in the AOT setting that allows more precise analysis, together with the aggressive inlining strategy described above. Conversely, stack allocation can also boost inlining. For example, if a method accesses private fields, it cannot be inlined at call sites in other classes, however, if the fields are moved to local variables then inlining can be performed without violating Java's access control mechanisms.

To understand how stack allocation can apply to stream pipelines, consider the `filter` operation from Figure 4. It contains an inner class, which the Java compiler lifts outside the method. The free variables of the inner class, `PushStream.this` and `p`, then become fields that are set by the constructor. Java code that roughly corresponds to the resulting bytecode is shown in Figure 7. Here, it is clear that every access to `PushStream.this` and `p` in the original method actually involves fields in objects in the heap. This heap allocation cannot be converted into stack allocation without information about the code that calls the `filter` method and uses of the resulting stream object via its `exec` method. By statically analyzing the entire pipeline, our approach can obtain the required

[2] https://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html#escapeAnalysis

```
67  private static class PushStream$1<V> extends PushStream<V> {
68      private final PushStream<V> previous;
69      private final Predicate<? super V> predicate;
70
71      PushStream$1(PushStream<V> previous, Predicate<? super V> predicate) {
72          this.previous = previous;
73          this.predicate = predicate;
74      }
75
76      protected void exec(Consumer<? super V> c) {
77          previous.exec(x -> {
78              if (predicate.test(x)) c.accept(x);
79          });
80      }
81  }
82
83  public PushStream<V> filter(Predicate<? super V> p) {
84      return new PushStream$1<>(this, p);
85  }
```

Fig. 7. The `filter` intermediate operation from Figure 4, flattened such that the inner class is now lifted outside the `filter` method, similar to the structure of the bytecode.

information. Also note that the `exec` method in Figure 7 cannot be inlined unless we also perform stack allocation of `PushStream$1`, because the fields are declared as **`private`**.

In summary, these observations suggest that method inlining and stack allocation, which are two classic optimization techniques, can be effective together in AOT optimization of stream pipelines.

3 APPROACH OVERVIEW

Figure 8 shows the structure of our approach. We start the optimization process by analyzing the compiled program with an off-the-shelf pointer analysis [Bravenboer and Smaragdakis 2009; Lhoták and Hendren 2003; Sridharan and Bodík 2006]. The purpose of this phase is to find the segments of the bytecode that correspond to stream pipelines in the program and to find the concrete types of the stream sources.

We then process each stream pipeline individually. As mentioned in Section 2, stream pipelines rarely span multiple methods, and the stream objects are rarely stored in the heap, so tracking the flow of stream objects is trivial (we skip pipelines where this is not the case). For each pipeline we perform a flow- and context-sensitive type and pointer analysis. The result of this analysis consists of an abstract state for every analyzed control flow graph node (in JVM bytecode, a node corresponds to a bytecode instruction), for each call context. Informally, an abstract state maps each local variable and object field to an abstract value, which is a pair of a Java type and an abstract points-to value. For the types, we distinguish between *abstract types* τ and *concrete types* $\overline{\tau}$ [Agesen 1995], where an abstract value with type τ can represent any object that is a subtype of τ, while $\overline{\tau}$ only represents objects that have exactly the type τ. The abstract points-to values use allocation-site abstraction [Chase et al. 1990] as the heap model. The analysis is explained in more detail in Section 5.

We then use facts from the analysis result in the next phase to guide the optimization transformations. The type information allows us to resolve calls for the inlining transformation, while the pointer information is used in the stack allocation transformation to redirect field accesses on stack allocated objects to the corresponding local variables. After the pipeline has been transformed,

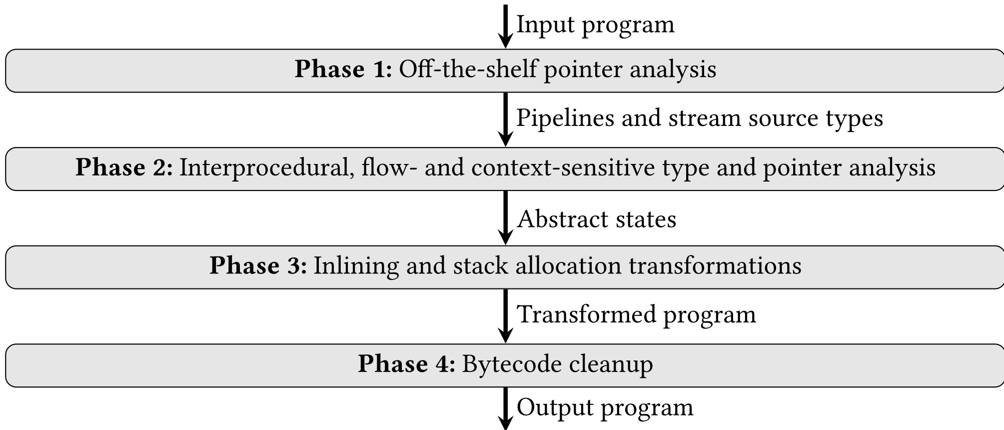


Fig. 8. Flow diagram of the approach.

we feed it to a cleanup phase that can remedy some inefficiencies that are inherent to the two transformations.

It may be the case that the analysis aborts. This happens when it can determine that the analysis result would not allow optimizations to take place. If, for example, the analysis discovers that a pipeline object flows out of the boundary of the analyzed method, for instance to a static field, then after this point the analysis (and therefore also the transformation) cannot make any useful assumptions about the state of the object. Another condition for aborting is over-approximation of an unanalyzable call that leads to an unusable analysis result. These and other cases are described in more detail in Section 5.

Example. To get an intuitive understanding of the approach before we explain the details in the following sections, we can apply it to the example stream pipeline shown in Figure 9a. It is presented as Java source code for readability although the actual technique works on Java bytecode.

The pre-analysis finds a stream pipeline in the `sum` method. The initial abstract state for the main analysis is seeded with abstract values from the pre-analysis for the local variables and the stack. The method contains one local variable for the argument `v`, which is given the abstract value (`int[]`, unrelated). Here, unrelated denotes a pointer value that cannot refer to any allocation site that appears during the analysis and whose field values are not tracked in the abstract heap. Since the stream source is constructed by a static method call in this example, it does not require additional type information to be resolved. (We show an example in Section 4 that needs the pre-analysis to infer the concrete type of the stream source object.)

The first instructions encountered during the main analysis phase correspond to the call to `IntPushStream.of(v)`. We resolve this call by finding the static method on `IntPushStream` with the correct signature and continue by analyzing the resolved method:

```
86 public static IntPushStream of(int[] arr) {
87     return new IntPushStream() {
88         public void exec(IterConsumer c) {
89             int i = 0;
90             while (i < arr.length) c.accept(arr[i++]);
91         }
92     };
93 }
```

```
94  public static int sum(int[] v) {
95      IntPushStream stream = IntPushStream.of(v);
96      return stream.reduce(0, Integer::sum);
97  }
```

(a) An example method, sum, containing a stream pipeline.

```
98  public static int sum(int[] v) {
99      // inlined IntPushStream.of
100     int[] IntPushStream_of_arr = v;
101     IntPushStream stream = null;
102     int[] IntPushStream$0_arr = IntPushStream_of_arr;
103     // inlined IntPushStream.reduce
104     IntPushStream IntPushStream_reduce_this = stream;
105     int IntPushStream_reduce_initial = 0;
106     IntBinaryOperator IntPushStream_reduce_r = null;
107     Reducer IntPushStream_reducer = null;
108     int Reducer_state = IntPushStream_reduce_initial;
109     IntBinaryOperator Reducer_operator = IntPushStream_reduce_r;
110     // inlined IntPushStream$0.exec
111     IntPushStream$0 IntPushStream$0_exec_this = (IntPushStream$0)
            IntPushStream_reduce_this;
112     IterConsumer IntPushStream$0_exec_c = IntPushStream_reducer;
113     int IntPushStream$0_exec_i = 0;
114     while (IntPushStream$0_exec_i < IntPushStream$0_arr.length) {
115         // inlined Reducer.accept
116         Reducer Reducer_accept_this = (Reducer) IntPushStream$0_exec_c;
117         int Reducer_accept_v = IntPushStream$0_arr[IntPushStream$0_exec_i++];
118         // inlined Integer.sum
119         int Integer_sum_a = Reducer_state, Integer_sum_b = Reducer_accept_v;
120         Reducer_state = Integer_sum_a + Integer_sum_b;
121     }
122     return Reducer_state;
123 }
```

(b) The sum method after inlining and stack allocation.

```
124 public static int sum(int[] v) {
125     int state = 0, i = 0;
126     while (i < v.length) state += v[i++];
127     return state;
128 }
```

(c) The resulting sum method after the cleanup phase.

Fig. 9. Optimization of a stream pipeline.

This `of` method constructs an instance of an anonymous subclass of `IntPushStream`. Following Java conventions this subclass could be named `IntPushStream$0`. We update our abstract state with this allocation site named ℓ_1 and continue by analyzing the constructor of `IntPushStream$0` (not shown here) and see that the implicitly passed `arr` parameter is stored as a field that is also named `arr` in the subclass. The abstract value that is returned from `IntPushStream.of` is (`IntPushStream$0`, ℓ_1), and the heap part of the new abstract state is the map $[\ell_1 \mapsto \{\texttt{arr} \mapsto (\texttt{int[]}, \text{unrelated})\}]$. The next instructions in `sum` allocate the lambda argument to the `reduce` function, causing the abstract state to be updated with a new allocation site, ℓ_2. The abstract value

of the receiver of the reduce call is $(\overline{\texttt{IntPushStream\$0}}, \ell_1)$ and contains precise type information such that we can resolve the call according to Java's virtual method invocation semantics.[3] We thus continue by analyzing IntPushStream.reduce, shown below, where the abstract values of the receiver and the arguments are $(\overline{\texttt{IntPushStream\$0}}, \ell_1)$, $(\overline{\texttt{int}}, \top)$, and $(\overline{\lambda_0}, \ell_2)$, respectively, where λ_0 denotes the type of the object created for the method reference Integer::sum.

```
129  public int reduce(int initial, IntBinaryOperator r) {
130      Reducer reducer = new Reducer(initial, r);
131      this.exec(reducer);
132      return reducer.state;
133  }
```

Continuing analysis in this method, after allocating the Reducer at allocation site ℓ_3, we have the abstract heap

$$[\ell_1 \mapsto \{\texttt{arr} \mapsto (\texttt{int[]}, \text{unrelated})\}, \ell_2 \mapsto \{\}, \ell_3 \mapsto \{\texttt{state} \mapsto (\overline{\texttt{int}}, \top), \texttt{reducer} \mapsto (\overline{\lambda_0}, \ell_2)\}]$$

and the abstract values of **this**, initial, and r are $(\overline{\texttt{IntPushStream\$0}}, \ell_1)$, $(\overline{\texttt{int}}, \top)$, and $(\overline{\lambda_0}, \ell_2)$, respectively. The Reducer carries the state field to keep track of the running sum while the pipeline executes, and a reference to the reducer method. The purpose of that method is to compute a new state from a stream element and an old state, as a left-fold operation.

To resolve the call to exec, the analysis looks up the abstract value of the receiver (**this**), which is $(\overline{\texttt{IntPushStream\$0}}, \ell_1)$ in this case, so it can continue the analysis in IntPushStream$0.exec where the abstract value of the first argument is $(\overline{\texttt{Reducer}}, \ell_3)$. Inside exec (see lines 88–91) the field arr of ℓ_1 is accessed twice. The analysis can precisely resolve these accesses using the abstract state since the abstract points-to value of **this** is ℓ_1. The call to Reducer.accept is resolved and the analysis continues in that method:

```
134  public void accept(int v) {
135      state = reducer.applyAsInt(state, v);
136  }
```

Here it is even more crucial that the analysis has precise type and points-to information for **this**, as this allows it to look up the value of reducer on ℓ_3 in the abstract state and resolve the call to λ_0.applyAsInt. After finishing the analysis, we have the information necessary to unambiguously resolve the calls at every analyzed call site.

The following phases perform optimizing transformations on the analyzed pipeline. Like the analysis it operates on stack-based Java bytecode, but we will outline the transformation as if it happens directly on Java source code. The first transformation phase applies inlining and stack allocation transformations. Local variables are used in place of the object's fields and are also allocated for the parameters of inlined methods.

We can look into how the analysis result is used to transform the call to IntPushStream.of(v) in Figure 9a. The analysis resolved the callee such that it can be inlined into the sum method:

```
137  int[] IntPushStream_of_arr = v;
138  IntPushStream stream = new IntPushStream() {
139      public void exec(IterConsumer c) {
140          int i = 0;
141          while (i < IntPushStream_of_arr.length)
142              c.accept(IntPushStream_of_arr[i++]);
143      }
144  };
```

[3] https://docs.oracle.com/javase/specs/jvms/se13/html/jvms-6.html#jvms-6.5.invokevirtual

Notice that a fresh local variable has been generated for the `arr` parameter.

The next step is to allocate `IntPushStream$0` on the stack instead of on the heap (see line 138). First, the **new** instruction is replaced with **null** to preserve the operand stack layout. The class has one field of type `int[]` so a local variable is allocated for it (`IntPushStream$0_arr` at line 147 below). The variable is associated with the allocation site ℓ_1 of the `IntPushStream$0` object, and the constructor is inlined. Inside the constructor of `IntPushStream$0`, the array is stored as a field on the object. According to the abstract state for the field write instruction, the object that is written to is the object allocated at site ℓ_1. The field write instruction can therefore be redirected to write to the local variable allocated for the object (see line 148):

```
145  int[] IntPushStream_of_arr = v;
146  IntPushStream stream = null;
147  int[] IntPushStream$0_arr = null;
148  IntPushStream$0_arr = IntPushStream_of_arr;
```

Whenever the transformation later finds an instruction that accesses the `arr` field of ℓ_1, it is similarly replaced with an instruction that instead accesses `IntPushStream$0_arr`.

The result of the transformation phase is shown in Figure 9b. This transformed method can be executed as is but is rather large and filled with redundancies. The last transformation phase aims to reduce the code size and the number of local variables in the resulting code. It does so by identifying and removing duplicate aliasing local variables, unused variables, and redundant bytecode instructions. After these transformations, we end up with the code shown in Figure 9c. Notice that the resulting code is a simple while-loop that iterates over the array, without any virtual calls, similar to what a programmer would likely write if not having streams available.

In the following sections, we describe how each of the four phases work more generally, and with more details about the analysis and transformations.

4 PHASE 1: PRE-ANALYSIS

The optimization process begins with a preliminary analysis. Its goals are (1) to identify stream pipelines within the analyzed program, and (2) to restrict the set of possible concrete types for stream sources. This information allows us to subsequently use an expensive analysis, which is specialized for guiding our optimizations, only at the program points where it is needed. The concrete types for the stream sources are used for seeding the analysis in phase 2.

Consider the example stream pipeline marked with gray in Figure 10. By simply using the type information available in the Java bytecode of the compiled program, it is trivial to find all local

```
149  class Application {
150    private void method(...) {
151      List<Integer> list = new ArrayList<Integer>();
152      // ...
153      boolean anyMatching = list.stream()
154                            .map( x -> x * y )
155                            .anyMatch( x -> x > z );
156      // ...
157    }
158  }
```

Fig. 10. A stream pipeline (marked) as part of a bigger program. Note that the lambdas are excluded from the code considered by the main analysis.

variables of type `Stream`, which allows us to recognize the segment of bytecode constituting the pipeline. (In case the stream objects are passed as parameters or return values of non-application methods, or they are stored in or retrieved from fields in objects, we simply give up optimizing the pipeline, as mentioned earlier; this can be improved in future work.) Application code, such as the two lambdas in Figure 10, is considered outside the pipeline by the main analysis although it is being inlined in the transformation phase. However, we do analyze the (implicit) constructors for the lambdas, to be able to detect if stream objects escape from the pipeline code. An exception is made for the `flatMap` operator where the callback creates a stream object directly involved in the execution of the pipeline, and must therefore be included in the main analysis.

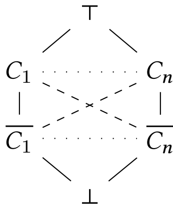
In many cases, the stream source type is trivial to infer (as in the `sum` example in Figure 9a), but other cases require information about dataflow. When compiled to bytecode, the call to `list.stream()` simply contains `List` as the receiver of the call. At run-time it is up to the JVM to dispatch the call to the implementation of the concrete type of the receiver of the call, in this case `ArrayList`. It does not require an advanced analysis to figure out that the concrete type of the receiver is indeed `ArrayList` in this simple example, but the receiver is not always allocated in the same method as the stream pipeline, as it could be passed as an argument to the method or reside in a field of an object. To handle such situations, we can apply an off-the-shelf pointer analysis to statically find the concrete type of the stream source object. Several such tools are available, including Soot [Lhoták and Hendren 2003; Sridharan and Bodík 2006], Doop [Bravenboer and Smaragdakis 2009], and WALA [Dolby et al. 2010]. Instead of simply selecting every non-abstract subclass of the declared type `List` of the `list` variable, such analyzers can narrow the set of possible concrete types by safely over-approximating the dataflow in the program. Usually, this gives us a single concrete type for each pipeline source. In case multiple possible concrete types are found, one possibility is to optimize the pipeline separately for each of them and then branch at run-time based on the actual type. Since we typically only need the type information for a small number of expressions in the program, a demand-driven analysis [Späth et al. 2016; Sridharan and Bodík 2006], which only analyzes the relevant part of the code, is a good fit.

5 PHASE 2: INTERPROCEDURAL ANALYSIS

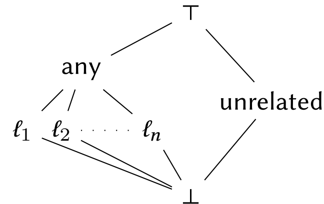
We express the main analysis as a monotone framework [Kam and Ullman 1977], which requires a lattice of abstract states. Abstract values are defined as elements of the lattice

$$\mathit{Values} = \mathit{Type} \times \mathit{Pointer}$$

and Type and $\mathit{Pointer}$ are illustrated in Figure 11. The Type lattice contains all concrete and abstract types, for modelling classes, interfaces, primitives, and arrays. The dashed edges between the types represent the class and interface inheritance relations. The least upper bound of two Java types is not always a single unique class or interface, which is why this simple lattice is chosen.



(a) Type lattice for all classes, interfaces, primitive types, and array types $C_1 \ldots C_n$.



(b) $\mathit{Pointer}$ lattice for allocation sites $\ell_1, \ell_2, \ldots, \ell_n$.

Fig. 11. Lattices for type and pointer components of abstract values.

In the $Pointer$ lattice, any represents a value that can point to any allocation site that occurs in the pipeline. The lattice element unrelated represents values that can point to objects unrelated to the execution of stream pipelines or objects allocated before the analysis entry point. Values that are references to objects are modeled by object labels, $\ell_1, \ell_2, \ell_3, \ldots, \ell_n \in ObjectLabels$. Objects are abstracted by their allocation sites, $ObjectLabels = Contexts \times Nodes$. Here, $Nodes$ is the set of control flow graph nodes (i.e., bytecode instructions), and $Contexts$ is the set of all call contexts (explained below), so the allocation sites are qualified by contexts (also called heap cloning or context sensitive heap) [Nystrom et al. 2004; Smaragdakis et al. 2011].

An abstract state defines an abstract value for each local variable and operand stack cell (together referred to as the set $Cells$), and it carries an abstract heap that maps each object label to a map from fields to abstract values:

$$States = \overbrace{(Cells \rightarrow Values)}^{\text{Stack and locals}} \times \overbrace{(ObjectLabels \rightarrow Fields \rightarrow Values)}^{\text{Heap}}$$

To achieve context sensitivity we apply the well known call-string technique with unbounded length [Sharir and Pnueli 1981]. This allows us to precisely analyze stream pipelines of arbitrary depth. Such an extreme choice of context sensitivity is of course not scalable to Java code in general, but stream pipelines are relatively small. For modeling object constructions, we pick the entire current call context as context for the object labels. The analysis is also flow sensitive, so the full analysis lattice has an abstract state for each context and control flow graph node:

$$Contexts \rightarrow Nodes \rightarrow States$$

The analysis is invoked for each pipeline found in phase 1. Such a pipeline is marked by a range of bytecode instructions that contain all instructions relevant to its execution. Thus the analysis can start at the first of these instructions when initialized with a sensible initial abstract state. This state contains the concrete type information from the pre-analysis necessary to resolve the call to the stream source. The analysis then proceeds to analyze the pipeline code. If a critical loss of precision occurs on the way, the analysis aborts and no optimization of the pipeline is performed.

The transfer functions of the intraprocedural parts of the analysis are straightforward. Generally they model the modification to the abstract operand stack and local variables after executing bytecode instructions. For instance, the `getfield` instruction on a field F of type τ looks at the topmost value (τ', p) of the abstract stack and proceeds by case analysis to figure out which value v to replace it with:

$$v = \begin{cases} lookup(p)(F) & \text{if } p \in ObjectLabels \\ \bigsqcup_{\ell \sqsubseteq p \,\wedge\, filter(\ell,\tau')} lookup(\ell)(F) & \text{if } p = \text{any} \\ (\tau, p) & \text{if } p \in \{\top, \text{unrelated}\} \\ (\bot, \bot) & \text{if } p = \bot \end{cases}$$

The function $lookup(\ell)(F)$ looks up the abstract value of field F in abstract object ℓ in the current abstract state. In the first case, $p \in ObjectLabels$, we simply look up the field value, which precisely captures the semantics of `getfield`. If $p =$ any then the result value is the least upper bound of the abstract values for that field on all relevant objects. The predicate $filter(\ell, \tau')$ filters the set of object labels ℓ according to the type τ': If τ' is a concrete type then only abstract objects of exactly that type are included, otherwise abstract objects that are subclasses of τ' are included. If $p \in \{\top, \text{unrelated}\}$, then v cannot be refined further than (τ, p), as p could point to an object that the analysis does not track.

For `putfield` instructions, the two topmost values on the stack are popped. Let (τ_o, p_o) and (τ_v, p_v) denote the abstract values of the object reference and assigned value, respectively. If

unrelated $\sqsubseteq p_o$ and there exists an object label ℓ such that $\ell \sqsubseteq p_v$ the analysis aborts, as ℓ can escape from the analyzed part of the program. If $p_o \in \mathit{ObjectLabels}$, a strong update on the object can be performed,[4] otherwise a weak update on all the object labels $\ell \sqsubseteq p_o$ that have the corresponding field is performed.

For the interprocedural part of the analysis we define transfer functions for method calls. The first part of a method call is to resolve the callee. If the call instruction is `invokestatic` or `invokespecial` this is easy,[5] otherwise the callee depends on the run-time type of the receiver. If the type component of the abstract value of the receiver is precise, then we can use Java's virtual method lookup procedure. If not, we might be able to exploit that the targeted method is *final* or that the declaring class is *final*. Otherwise the analysis resorts to over-approximation without involving the callee body, as follows.[6] If any value (τ, p) flows into an over-approximated call (either as receiver or argument) where there exists some $\ell \in \mathit{ObjectLabels}$ such that $\ell \sqsubseteq p$ and $\mathit{filter}(\ell, \tau)$ holds, then the analysis aborts, as the method could modify the full reachable heap from this object, leading to a very imprecise heap. Otherwise, if the callee can be uniquely resolved, analysis continues in the resolved method in a new context with the current abstract heap. After the analysis of the method finishes, we merge the abstract states at all reachable return instructions in the method and continue analysis in the caller with the merged abstract heap. The return value is the topmost value on the stack in the merged state.

It is possible to construct stream pipelines whose structure is not statically fixed, for example by applying a stream operation conditionally as in the following example.

```
159  IntStream s = /* some source */;
160  if (shouldSquare) s = s.map(x -> x * x);
161  return s.filter(x -> x % 2 == 0).sum();
```

This causes the analysis to abort due to a failed resolution of a call target, in this case at a call that appears as part of the terminal operation, no matter if Java's stream library or the simple push- or pull-style implementation described in Section 2 is used.

Callbacks from the stream library to the application code, such as the lambdas in Figures 1b, 9a and 10, are not analyzed unless it is necessary. These must be analyzed if a pipeline object flows into such a method (the lambda can capture a reference to the pipeline), or if it is the callback from a `flatMap` operator.

Calls to methods implemented in native code are handled by over-approximating as explained above. To prevent this from aborting the analysis in common cases we use custom models for a few core methods in the Java standard library (e.g., `Class.getName` and `System.arraycopy`). Other typical obstacles to sound and precise static analysis for Java, such as reflection or dynamic class loading, are not a concern, because we only apply the analysis to the stream library code, and stream libraries do not use such mechanisms. (If the analysis should encounter use of such mechanisms, it simply aborts.)

Context-sensitive analysis with unbounded call strings may diverge for programs that contain recursion. It is not trivial to detect ill-natured recursion, as the call stack may legitimately contain the same method multiple times during the execution of a stream pipeline, if the pipeline contains multiple instances of the same intermediate operation. To ensure termination, we therefore abort

[4] Strong updating [Chase et al. 1990] is sound in this situation because of the use of flow sensitivity and full context sensitivity.

[5] `invokestatic` is a direct method call while `invokespecial` resolves the callee by traversing the superclasses of the enclosing class until a matching method is found.

[6] We could instead apply some variant of Class-Hierarchy Analysis [Dean et al. 1995] to find potential callees and merge the results of analyzing those methods.

the analysis if the length of a call string exceeds 1 000. This bound is well above what is needed to admit analysis of most pipelines.

With this analysis lattice and these transfer functions, the analysis runs using a standard worklist algorithm that repeatedly applies the transfer functions until either a fixed-point is reached, in which case we proceed to the transformation phase, or the analysis aborts due to one of the conditions described above. In summary, the realistic situations where the analysis aborts are (1) an object created in the analyzed part of the code may escape that part of the code (this happens for around 2% of the pipelines in our experiments, see Section 8.2), (2) a call target cannot be resolved with sufficient precision (happens for around 14% of the pipelines), and (3) recursion causing the analysis to diverge (happens for less than 1% of the pipelines).

Handling Java's Stream Library. The analysis presented above suffices for simple stream libraries, such as the one described in Section 2, but not for more complex ones. As mentioned earlier, the Java standard library stream implementation is quite complex and uses different code paths for sequential and parallel computation, and for short-circuiting and non-short-circuiting pipelines. To obtain sufficient analysis precision to fuel optimizations, we need to avoid analyzing certain paths that are not taken in actual runs of the code. We achieve this by including constant propagation [Callahan et al. 1986] in the abstract values and by making the analysis control sensitive (also called branch sensitive) to take branch conditions into account for refining abstract values and for eliding dead code.

A common source of precision loss is the use of *stream flags* in the library code. Every stream pipeline has a set of flags that are queried at different stages of execution. If we cannot analyze these queries precisely, the analysis loses too much precision to be useful. The flags are bitmasks that are computed at run-time by the static initializer of the `StreamOpFlag` enum class. The code in a static initializer of the class is run the first time the class is accessed and is mainly used to populate static fields. Since we do not necessarily want to limit ourselves to a whole-program analysis, we cannot make assumptions about when this initialization happens and in what state the static fields of the class are in at the analysis entry point. However, we observe that if the fields have the `final` modifier, they cannot have been reassigned after initialization.

Analyzing `StreamOpFlag`'s initializer statically requires loop unrolling to be precise enough to be useful. We take a simpler approach: Instead we utilize an on-demand dynamic pre-analysis that takes a snapshot of the reachable heap after initializing static fields and preserves abstract values for fields that are marked final. This allows us to get precise information on `StreamOpFlag` and `StreamOpFlag$Type` enums needed for control sensitivity.

This small extension of the analysis relies on two assumptions. The first is that final fields of pre-analyzed classes are not modified by the client at run-time. This assumption could be violated by clients that use reflection,[7] or by bytecode that is not emitted by the Java compiler. Even though the Java compiler does not allow multiple writes to `final` fields, it is possible to load classes into the JVM that violate this constraint.[8] We also assume that the values of `static final` fields involved in the stream pipeline do not rely on the run-time environment in which they are initialized.

A final trick necessary to enable useful analysis of the Java stream implementation is a model for the standard library method `java.util.stream.AbstractPipeline.wrapSink`. This method is responsible for traversing the stream pipeline from back to front, chaining together consumers (called `Sinks` in Java stream terminology) along the way. This consumer is what the spliterator will send elements into when the pipeline executes, and analyzing the chaining precisely is therefore critical. This method is implemented with a loop instead of with recursion and thus loop unrolling

[7]However, since Java 9 the JVM can disallow all reflective accesses to JDK internal API's. See Relaxed-strong-encapsulation.
[8]See https://hg.openjdk.java.net/jdk/jdk12/file/06222165c35f/src/hotspot/share/interpreter/rewriter.cpp#l435

is necessary to analyze the behavior precisely. We have precise enough information to unroll the loop, but we have not extended the analysis to support this in the proof-of-concept implementation, so instead we replace that method with a model that has the loop manually unrolled.

6 PHASE 3: INLINING AND STACK ALLOCATION

Both kinds of transformations we apply, inlining and stack allocation, are classic compiler optimizations used for decades [Choi et al. 1999; Detlefs and Agesen 1999]. In this section we briefly describe how they work, in particular how they depend on each other, how they use the information from the main analysis phase, and what their limitations are. The transformation starts at the entry point of the pipeline and considers each bytecode instruction one-by-one.

6.1 Inlining

At a method call instruction the transformation tries to resolve the callee in the same way as the analysis, using the abstract state for this program point. If the callee cannot be uniquely determined, inlining is not applied for the call. Otherwise the callee is resolved to some method with n arguments and m local variables. At the call instruction the Java operand stack must contain at least n values where the top n values will be consumed by the call. At method entry, the callee expects the parameter values to be placed in the local variables numbered 0 to $n-1$. In the caller method, we allocate m new locals for the inlined method. For each argument in reverse order, a `store` instruction is inserted to the appropriate newly allocated local variable before the call instruction. The callee is then recursively transformed where care is taken to remap variable accesses to the allocated variables in the caller. Return instructions are handled by replacing them with an unconditional jump to a fresh label placed at the end of the inlined method.[9] The list of bytecode instructions in the transformed method is then spliced into the caller in place of the call instruction, and the maximum stack size of the caller is adjusted accordingly.

The transformation is easy to apply, but the ability to apply it in the AOT setting can be hindered by Java's access control mechanisms. If the callee is in a different class and/or package than the caller, the callee might be able to access fields, methods, and classes that the caller cannot, for example if they are declared **`private`**. In this case, inlining the callee would produce code that does not pass Java's runtime encapsulation checks [Budimlic and Kennedy 1997, 1998]. We return to this issue at the end of the section.

Since the analysis does not cover the whole program, the inlining transformation is only successful if callees can be inlined all the way into the body of the method containing the stream pipeline. If the analysis starts in method f_1 and analyzes a call to f_2 that further calls f_3, only inlining f_3 into f_2 using the abstract states from the analysis would be unsound, as f_2 could have other callers than f_1 where the abstract states do not match the ones we used for the transformation. This implies that the technique is not directly suitable to optimize parallel stream pipelines. In such pipelines, the work of executing the pipeline is delegated to multiple threads, and can therefore not be inlined into the method containing the stream pipeline.

6.2 Stack Allocation

Stack allocation can only be done for objects that do not escape their method [Choi et al. 1999]; the analysis has already checked that property as explained in Section 5. To be able to perform the transformation in a way that preserves the program semantics, it is also necessary that sufficiently

[9] The semantics of a JVM call instruction specify that (at most) one value is placed on the operand stack after execution. While the stack is not required to contain only one value at a return instruction, this is the case for all bytecode generated by the Java compiler. Additional measures can be taken to allow for full return semantics.

precise pointer information is available from the preceding phase. For this reason, we use the following concept of *stack allocation eligibility*. When an object is allocated in the stack, and its fields are stored in local variables in the call frame instead of on the heap, all instructions that access the object's fields must be appropriately transformed. If this condition cannot be satisfied for a given object (identified by an object label), then the object is ineligible for stack allocation. Eligibility is determined by examining all field access instructions in the analyzed code. At a field access instruction the abstract state can be queried for the abstract value of the object reference, denoted (τ, p). If $p \notin \mathit{ObjectLabels}$ then all object labels ℓ where $\ell \sqsubseteq p$ and $\mathit{filter}(\ell, \tau)$ holds are made ineligible for stack allocation. For such an ℓ the abstract state is not precise enough to ensure that this field access instruction can be redirected to the corresponding local variable. Notice that stack allocation eligibility relies on inlining – all method calls that the potentially stack allocated object flows into must be inlined to ensure that we can translate load and store instructions and inline virtual calls.

After stack allocation eligibility is determined, the transformation starts. It operates on `new`, `putfield`, and `getfield` instructions. Whenever `new` is encountered, the transformation checks whether the allocation site defined by this bytecode instruction is eligible for stack allocation. If this is the case, local variables are allocated for all of the object's fields, and they are associated with the object label. To keep the operand stack layout valid, the `new` instruction is temporarily replaced with an instruction that loads the `null` constant.

At a `getfield` instruction, the abstract state is queried for the abstract value of the object reference, denoted (τ, p) as above. If $p \in \mathit{ObjectLabels}$ and p is eligible for stack allocation, the instruction is transformed into a read to the local variable that was previously allocated for the field. The transformation handles `putfield` instructions similarly.

As mentioned in Section 3, stack allocation can enable more inlining optimizations. If a candidate method for inlining contains field accesses that would violate Java's access control, inlining can only take place if the object that is accessed is allocated on the stack, such that its fields can be accessed as local variables instead.

In practice, because of the interdependencies between the two transformations, optimizing a given stream pipeline with our technique is usually "all or nothing" – either inlining succeeds for all the methods involved in the pipeline and all the objects created in the process are stack allocated, or the optimization fails entirely.

6.3 Handling Private Fields

In Section 1 we mentioned how spliterator implementations for Java's standard library contain accesses to package-private fields (see Figure 5). These accesses are directed at a `Collection` object that should not be stack allocated, either because the object is not allocated within the analyzed method, or because the scope of the analysis would have to be broadened to not only include the stream pipeline but also parts of the application code relevant to the collection object, to be able to carry out the necessary transformations. The consequence is that inlining the spliterator methods will always be prohibited by the rules described above. Not being able to inline the spliterator methods produces a cascade of other optimizations that cannot take place, due to the interplay between them. When this happens, different courses of action are possible:

Inline as much as possible: We can choose to apply only the optimizations that are possible. This is, however, undesirable, as the bulk of the performance benefits of the optimization comes from fully inlining the call to `spliterator.forEachRemaining` and eliminating chains of virtual calls to push elements through the pipeline. This chain starts in the `forEachRemaining` method, and when this method cannot be inlined it disallows inlining of further calls.

Use reflection: Violating field accesses can be circumvented with the use of Java's Reflection API. The use of reflection can have unfortunate performance drawbacks, and is only possible when the application is run in an appropriate Java security context. Since reflection allows us to expose encapsulated fields that are not part of the object's interface, using reflection only works as long as the internal implementation of the object does not change, which could happen between different Java releases.

Exploit the Java module system: Since Java 9 introduced the Platform Module System,[10] it is possible, with appropriate JVM settings, to inject a class into the same module and package as the collection class, to expose private members of the class in a public interface. This does not suffer the same performance drawback as reflection, but still depends on the internal implementation of the class. In addition the JVM must be run with special settings.

Copy the class: We can make a copy of the collection class that publicly exposes its members. This class must be used in place of the original class throughout the application code. This way the application will work no matter the environment in which it runs, and will not suffer any performance drawbacks.

None of these solutions are ideal, but allow the transformation to optimize stream pipelines with collection sources. For our experiments we chose the last course of action as the lesser evil.

7 PHASE 4: CLEANUP

The above transformations, while general, typically introduce a lot of redundant bytecode instructions. The goal of the cleanup phase is to remove some of these redundancies from the transformed method. The `null` values introduced temporarily during the stack allocation transformation (see Section 6.2) are also eliminated in this phase. As input, the phase receives the transformed method from the previous phase. It then applies a few simple intraprocedural analyses and transformations described below. We motivate the cleanup techniques with two examples of method bodies that can be shortened. In the first example, method `f` invokes `twice` in line 164:

```
162  int f(int i) {
163    ILOAD 0
164    INVOKE int twice(int)
165    ...
166  }
```

```
167  int twice(int x) {
168    ILOAD 0
169    ICONST_2
170    IMUL
171    IRETURN
172  }
```

```
173  int f_opt(int i) {
174    ILOAD 0
175    ISTORE 1      (removed)
176    ILOAD 1       (removed)
177    ICONST_2
178    IMUL
179    ...
180  }
```

When `twice` has been inlined into `f` as shown on the right, the argument loaded for `twice` will be stored into a fresh local variable that is immediately reloaded and never reassigned. In this case we can remove the store and load instructions (indicated by 'removed' above). In general, such redundancies occur whenever we inline a method that is called with variables as arguments, and the method never assigns to the local variables for those parameters.

Another type of redundancy is introduced in the stack allocation transformation. Consider the following example on the left. A `State` object is created, then `i` is written to its `value` field, and finally the value is read from the field and returned. In this example, the `State` object can be stack allocated.

[10] https://www.oracle.com/corporate/features/understanding-java-9-modules.html

```
181 class State { public int value; }
182
183 int m(int i) {
184   NEW State
185   DUP
186   INVOKE void State.<init>()
187   ASTORE 1
188   ALOAD 1
189   ILOAD 0
190   PUTFIELD State.value int
191   ALOAD 1
192   GETFIELD State.value int
193   IRETURN
194 }
```

```
195 int m_opt(int i) {
196   ACONST_NULL (removed)
197   DUP         (removed)
198   POP         (removed)
199   ASTORE 1    (removed)
200   ALOAD 1     (removed)
201   ILOAD 0
202   ISTORE 2    (removed)
203   POP         (removed)
204   ALOAD 1     (removed)
205   POP         (removed)
206   ILOAD 2     (removed)
207   IRETURN
208 }
```

On the right, the method is shown after applying the stack allocation transformation, but before cleanup. The `State` class has one field, so a fresh local variable is allocated for it in the method, in this case it gets the index 2. The `putfield` instruction is replaced with a store to the allocated local variable followed by a pop. This pop is necessary to preserve the operand stack layout. A similar transformation is applied for the `getfield` instruction. With further intraprocedural simplifications, the body of the method can now be reduced to only a `load` of the argument followed by a `return` instruction, making all the other instructions redundant.

Figure 9b shows a transformed stream pipeline with redundancies from both the inlining and stack allocation transformations, in Java source code form.

Both inlining and stack allocation introduce a lot of redundant local variables in the transformed method. To eliminate such redundancy, we incorporate a flow-sensitive must-alias analysis similar to the one used in the Scala compiler.[11] This analysis determines which values are guaranteed to be equal for each program point. For instance it can determine that the local variables 0 and 1 must alias at line 176. With this information we can redirect the load instruction to the first local variable. This in turn makes the instructions in lines 174 and 175 dead, so they can safely be removed. We additionally employ other well-known intraprocedural analyses: strongly live variables analysis, nullness analysis, reachability analysis, and sign analysis, and the optimizations they enable, together with a suite of peephole optimizations.

In our benchmarks, the cleanup optimizations reduce the number of local variables in the transformed stream pipeline by a factor of 10 to 40 and the number of bytecode instructions by a factor 10, and thereby enable further optimizations by the JIT. Of course these numbers vary a lot depending on the pipeline in question.

8 EVALUATION

Our proof-of-concept implementation of the approach, named Streamliner, consists of approximately 8 KLOC Java code, building on ASM[12] for bytecode manipulation and analysis.

We evaluate our approach by answering the following research questions:

RQ1: Is the performance of the optimized code comparable to that of hand-optimized code, when applied to micro-benchmarks and using either push- or pull-style libraries?

RQ2: To what extent is the technique able to optimize stream pipelines in real-world Java applications? In cases where it fails, what are the reasons?

The Streamliner implementation and experimental data are available at https://brics.dk/streamliner/.

[11] https://github.com/scala/scala/blob/2.13.x/src/compiler/scala/tools/nsc/backend/jvm/analysis/AliasingAnalyzer.scala
[12] https://asm.ow2.io/

Table 1. Java Virtual Machines used in the performance evaluation.

Name	Java Version	Build number
Oracle HotSpot VM	8	1.8.0_241
OpenJDK HotSpot VM	13	13+13
GraalVM CE	11	11.0.6+9-jvmci-20.0
Eclipse OpenJ9	13	0.18.0

8.1 RQ1: Performance Evaluation

To answer the first research question we wish to compare the performance of programs before and after optimization. Our approach mainly targets stream pipelines which usually are components of larger programs. To isolate the performance impact of our optimization we evaluate the approach on a suite of 11 micro-benchmarks that consist only of stream pipelines. This suite builds upon micro-benchmarks from previous work [Biboudis et al. 2015] and includes a new benchmark that uses the `allMatch` terminal operation. This operation terminates the execution of the pipeline as soon as an element that does not satisfy the supplied predicate is found (it is a short-circuiting operation), and therefore follows an alternative code path in the Java stream library.

We do not include real-world Java programs in the performance evaluation, as measuring the impact of the optimization would be extremely difficult to do in a fair manner. Stream pipelines are used for different reasons and with different workloads, as small parts of bigger applications. Many stream pipelines in existing code are not performance critical; conversely, programmers sometimes avoid using streams exactly for performance reasons, as discussed in the introduction.

The performance measurements are made using the Java Microbenchmarking Harness (JMH) tool [Oracle 2014a], a benchmarking tool designed for JVM-based languages included in the OpenJDK project. It performs a series of iterations to warm up the JIT before doing proper testing iterations. In our experiments we perform 5 warm-up iterations and 10 normal iterations, and the presented number is the average over those 10 iterations. We omit confidence intervals, as fluctuations between runs are negligible compared to the differences resulting from the use of optimization and the choices of library and VM [Georges et al. 2007].

We have performed experiments on the four different Java VMs and versions shown in Table 1. For each VM we measure the performance of each micro-benchmark before and after optimization. For each benchmark we have four groups. The *Baseline* group constitutes the benchmark implemented with Java `for`-loops, while the *Pull* and *Push* groups use the simple library implementation described in Section 2. We include our own stream library implementations to show that they suffer from the same performance deficiencies as the Java stream implementation compared to the baseline, and that the optimization can yield performance improvements for both pull- and push style stream APIs. Finally, the *Stream* group uses the stream implementation of the Java standard library. The results can be found in Figures 12 to 15. The `sum` and `sumOfSquaresEven` benchmarks are shown in Figure 9a and Figure 1, respectively.

The results show that, when using Java's stream library, after optimization 10 of the 11 benchmarks have comparable performance to that of the baseline implementation.

The technique fails to optimize the stream pipeline for `flatMapTake` in the Stream group on all VMs except Oracle HotSpot VM 8 (indicated by gray bars). This benchmark features a short-circuiting pipeline that includes a `flatMap` operator, which uses a lazily-initialized buffer to hold elements from its generated streams.[13] The lazy initialization pattern results in too much imprecision causing the analysis to abort. This can be remedied in future work by more precise analysis.

[13] All the VMs use the OpenJDK implementation of the standard library for streams. The Oracle Hotspot VM uses an earlier version that does not use lazy initialization.

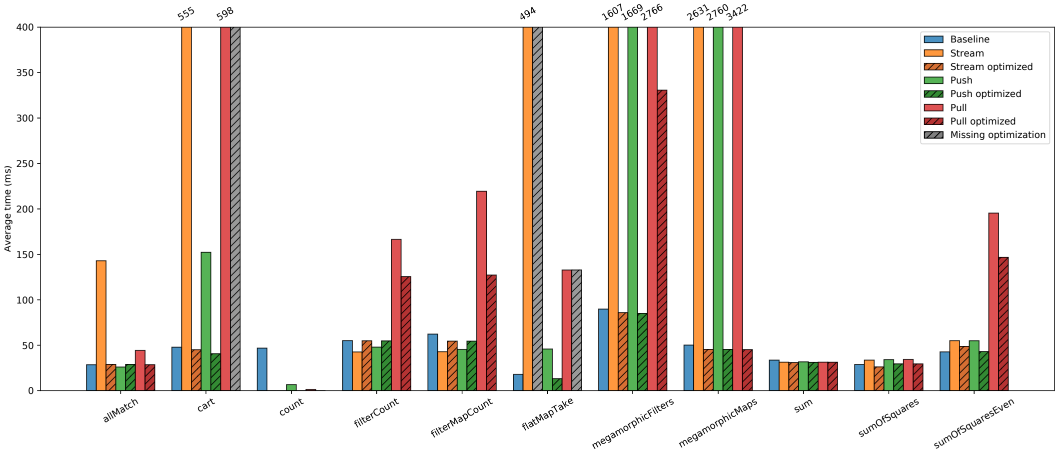


Fig. 12. Micro-benchmarks run on OpenJDK 13.

Across all implementations we experience a massive speedup for the (pathological) `cart`, `megamorphicFilters`, and `megamorphicMaps` benchmarks. The `count` benchmarks (which count the number of elements in a stream) experience drastic speedups after optimization, because the JIT can determine that the result is equivalent to the length of the supplied array, effectively making the run-time negligible.

Although our focus is on Java's stream library, we also test the applicability of our approach for pull-style streams. Across all tests, the analysis is too imprecise to optimize the `cart` and `flatMapTake` benchmarks in the Pull group. These benchmarks all use the `flatMap` stream operator. In the pull stream implementation, this operator assumes the iterator protocol in that calls to `get` are preceded by a call to `hasNext` returning **`true`**. A relational analysis is required to separate the abstract states for when `hasNext` returns `true` or `false` respectively. In many cases the performance of the optimized pull-style stream pipelines does not match that of the baseline, nor the performance of the optimized Java pipelines. This is due to a suboptimal structure of the optimized bytecode, which results in the JIT compiler generating performance-wise worse machine code. The same reason explains how optimized code in some cases performs marginally worse than the unoptimized version, as seen in the `filterCount` and `filterMapCount` benchmarks with OpenJDK 13 in Figure 12. Further cleanup transformations are needed to make the bytecode as efficient as the baseline.

There are some differences between the results on the various VMs. The Oracle HotSpot VM is slower for some benchmarks in the Baseline group compared to OpenJDK 13, which is not surprising as the OpenJDK VM has experienced five more years of development. However, we still experience the same relative speedup when the optimization is applied.

For the OpenJ9 VM, we also experience significant speedups for the optimized code, although OpenJ9's absolute performance seems to be below that of the other VMs. In some of the benchmarks, in particular the `sum` benchmarks, the Stream optimized code seems to be twice as fast as the baseline, which is suspicious. A plausible explanation for this is that the Java Microbenchmarking Harness is geared towards performance evaluation of HotSpot-based virtual machines (and OpenJ9 is not based on the HotSpot VM), which may lead to inaccurate measurements.

The `megamorphicMaps` benchmark that was highlighted in Section 1 for its exceptionally poor performance when executed with Java's streams is presented in Figure 16. Figures 16a and 16b show

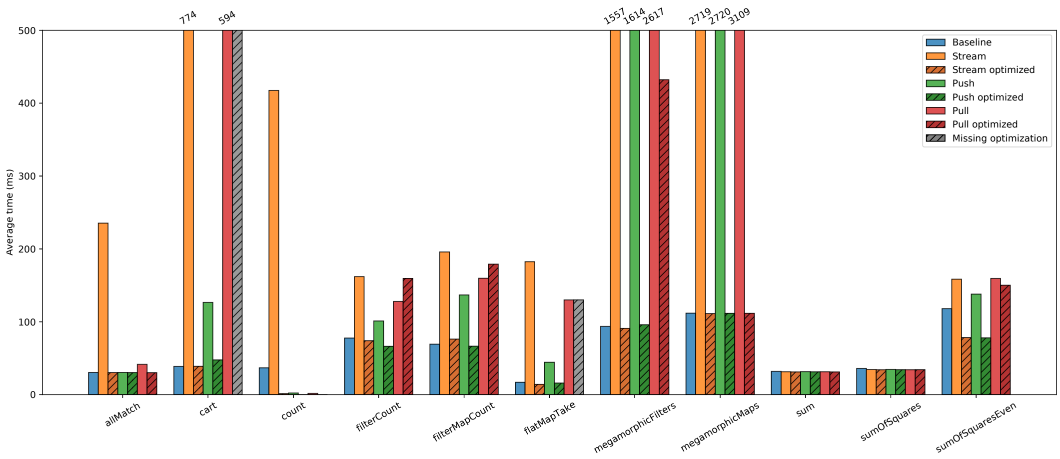


Fig. 13. Micro-benchmarks run on Oracle's JDK 8.

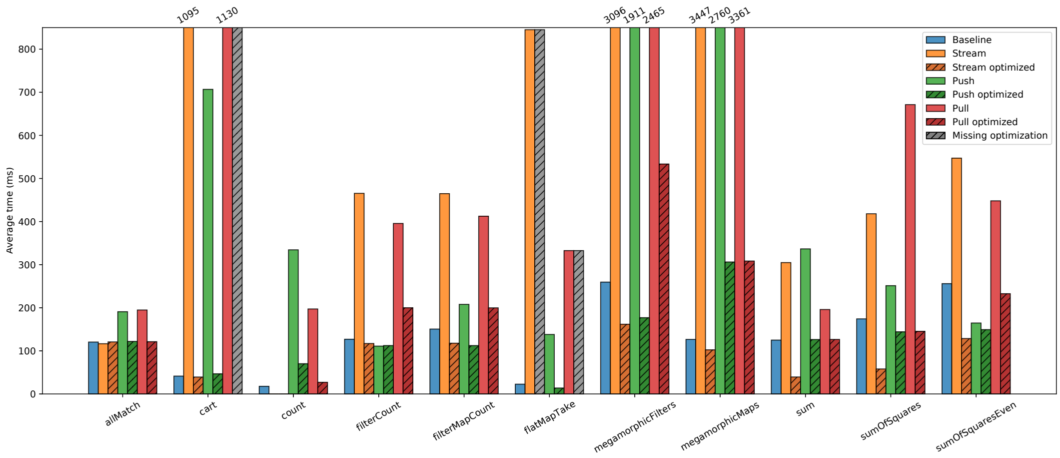


Fig. 14. Micro-benchmarks run on OpenJ9 13.

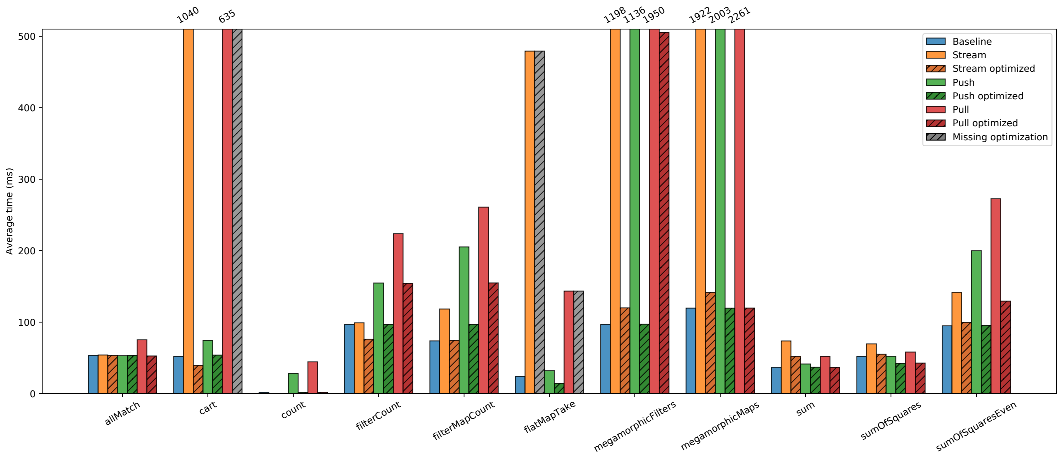


Fig. 15. Micro-benchmarks run on GraalVM 11.

```
209  public int megamorphicMaps() {
210    return IntStream.of(v)
211              .map(d -> d * 1)
212              .map(d -> d * 2)
213              .map(d -> d * 3)
214              .map(d -> d * 4)
215              .map(d -> d * 5)
216              .map(d -> d * 6)
217              .map(d -> d * 7)
218              .sum();
219  }
```

(a) Benchmark implemented with Java streams.

```
220  public int megamorphicMaps() {
221    int acc = 0;
222    for(int i = 0; i < v.length; i++)
223      acc += v[i]*1*2*3*4*5*6*7;
224    return acc;
225  }
```

(b) Benchmark implemented with a for loop.

```
226  public int megamorphicMaps() {
227    int[] values = v;
228    int endExclusive = values.length;
229    // bounds and null checking omitted
230    int state = 0;
231    if(values.length >= endExclusive) {
232      int i = 0;
233      if (0 < endExclusive) {
234        do {
235          int t = values[i];
236          int t2 = t * 1;
237          int t3 = t2 * 2;
238          int t4 = t3 * 3;
239          int t5 = t4 * 4;
240          int t6 = t5 * 5;
241          int t7 = t6 * 6;
242          int t8 = t7 * 7;
243          state += t8;
244        } while (++i < endExclusive);
245      }
246    }
247    return state;
248  }
```

(c) Decompiled benchmark after optimization.

Fig. 16. The megamorphicMaps benchmark.

the code that is executed in the Stream and Baseline group, respectively. In Figure 16c we show the optimized code after decompilation. The JIT compiler is able to transform both the baseline and optimized code into equally efficient machine code, but it is perhaps not immediately clear to the programmer that the code in Figure 16a is semantically equivalent to the code in Figure 16c. The optimized code shows similar structure to that of the baseline implemented with a for loop, but includes several transformation artifacts. One such artifact is the check of `values.length >= endExclusive`, which is always true. This could be removed with additional simple intraprocedural cleanup transformations, although it does not affect the performance.

8.2 RQ2: Evaluation on General Programs

To answer the second research question, we run the analysis and transformation on a suite of 28 different Java projects that use streams, to evaluate how many stream pipelines the analysis is able to optimize. Projects are randomly chosen from the RepoReapers dataset [Munaiah et al. 2017] under the criteria that we can build the project and that the project contains uses of streams.

Since we are not interested in evaluating the quality of the exact choice of pre-analysis, in this experiment we use a simple alternative to dataflow analysis to decide the stream source types. The ability to optimize a pipeline does not hinge on the concrete type of the stream, only that we know which one it is. For this reason, for streams created from collections (i.e., using the `stream` method in a sub-class of `java.util.Collection`), we simply choose a specific concrete collection type, such as `ArrayList`. For other kinds of stream sources, this pre-analysis simply aborts.

Table 2. Results of the evaluation on optimization of stream pipelines in general Java programs.

Category	Count	%
Successful optimization	5 293	77%
Imprecise resolution of call target	985	14%
Use of advanced stream operators	260	4%
Escaping pipeline object	121	2%
Infinite recursion	34	<1%
Other	186	3%
Total	6 879	100%

We identified 6 879 sequential stream pipelines in the chosen projects. As explained in Section 6, whether a stream pipeline can be optimized with our approach is "all or nothing". This gives us a simple way to classify each attempt to optimize a pipeline as being successful or not: If the optimizer succeeds in inlining *all* the stream library code used in the pipeline into the method containing the pipeline, then the optimization is successful. For each pipeline, we invoked the combined analysis and transformation, and recorded whether the pipeline was successfully optimized or not.

In the cases where the optimization is not successful, we have attempted to identify the most likely cause. The results of the experiment are presented in Table 2. Out of 6 879 pipelines, 5 293 (77%) are successfully optimized. This leaves 1 586 pipelines that fail to optimize for different reasons. The most prominent reason is that the analysis aborts due to imprecise type information at call sites, making it impossible to statically track the interprocedural control flow of the stream pipeline. This imprecision can arise from different sources as described in Section 5. One is that the simple pre-analysis implementation fails to deliver the type information needed to analyze the construction of the stream source, which accounts for about half of the 985 cases. (That may happen if the stream is created neither from a collection nor from static methods such as `IntStream.of`.) Incorporating a full-fledged pointer analysis, such as Boomerang, [Späth et al. 2016] can likely help the analysis in these cases. The analysis also experiences imprecision when the pipeline structure depends on branching (for example when an intermediate operation is applied to a stream only under some conditions, as in line 160). More advanced techniques could insert optimized code for both cases and branch on the original condition. The next most common cause of inability to optimize is the use of stream operators that the analysis is not precise enough to handle. This includes the `LongStream.range` source, which leads to infinite recursion, the `toArray` and `concat` operators, and the `flatMap` operator when involved in a short-circuiting pipeline as outlined in Section 8.1. These operators include some complex state that is initialized during pipeline execution which the analysis is unable to follow. This can cause the analysis to abort due to an imprecise resolution of a call target, as described in Section 5, or make the analysis result too imprecise to allow meaningful optimization, as described in Section 6. In 121 cases, the main analysis aborts due to an object escaping the analyzed part of the code, and in 34 cases the analysis aborts due to uncontrolled growth of call strings. Both of these conditions are described in Section 5. The remaining unsuccessful cases are harder to classify. In most of these cases the analysis succeeds but Java's access control mechanisms prevent optimization, as discussed in Section 6.

In summary, the results from this experiment show that the relatively simple static analysis presented in Section 5 can produce the information needed to optimize stream pipelines in a variety of programs. Moreover, the technique is quite cheap to apply. In our experiments, the analysis and transformation take approximately one second to apply for each pipeline on average.

9 RELATED WORK

Most work on compiler optimization for Java focuses on JIT optimizations [Arnold et al. 2005; Aycock 2003], and there is (surprisingly) little work on AOT optimizations in general for Java and related languages. To our knowledge, we are the first to investigate the use of AOT optimization for eliminating the massive abstraction overhead of Java stream pipelines.

The bytecode-to-bytecode optimizer described by Budimlic and Kennedy [1997, 1998] applies a transformation called object inlining, which inlines all data and code from selected objects, much like our use of method inlining and stack allocation, however, they do not present any strategies for when to apply the transformation. They also encounter the limitation of inlining methods that access private members. The term 'object inlining' has also been used for another kind of optimization that fuses together objects to reduce the number of object allocations, without method inlining or stack allocation [Dolby and Chien 1998].

Another related technique is the interprocedural escape analysis for guiding stack allocation optimization for Java by Choi et al. [1999]. Our dataflow analysis (Section 5) performs a variant of escape analysis by the use of the unrelated lattice element, to determine which objects may escape the stream pipeline code.

The Interflow optimizer [Shabalin and Odersky 2018] for Scala Native uses a combination of flow-sensitive type inference, method duplication, partial evaluation, partial escape analysis, and inlining. It focuses on optimizations for Scala's collection library, not for stream pipelines, and is designed for native code generation instead of bytecode-to-bytecode transformation. By targeting native code, they avoid the problems with Java's access modifiers discussed in Section 6. On the other hand, by choosing bytecode-to-bytecode transformation, our approach is easier to incorporate into existing build processes and execution platforms.

Our approach builds on ideas from the techniques mentioned above, and applies them to optimize stream pipelines. By focusing analysis and transformation on stream pipeline code that has large potential for optimization, we can afford more expensive analysis than the general purpose optimization techniques.

Our optimization technique can also be viewed as a form of program specialization [Schultz et al. 2003], where we specialize the stream library code to each individual stream pipeline. Instead of using a binding-time analysis as in traditional partial evaluation, we use a specialized analysis that simultaneously infers types and points-to information to guide the transformations.

Khatchadourian et al. [2020a] have developed a tool for optimizing Java streams that uses a static typestate analysis to determine whether it is advantageous to convert a sequential stream to a parallel one or vice versa. Parallel computation is a natural source of performance improvement, so their goal is to determine preconditions for when it is safe to execute pipelines concurrently. While parallel streams can offer better performance, it does not address the inherent overhead that is currently present when using Java's streams sequentially, as discussed in the introduction.

Declarative data processing has close ties to functional programming. Deforestation [Wadler 1990] is a technique that transforms functional programs that operate on trees (in particular lists) into equivalent programs without allocating intermediate results in new trees, thereby improving run-time performance. Many variants of deforestation exist, but mostly for functional programming languages. These techniques are difficult to adapt to optimize code that uses Java's stream library, in particular because of its advanced features described in Section 2.

For programming languages with advanced meta-programming capabilities, such as staging, efficient stream implementations can be obtained by implementing stream fusion and other optimizations within the libraries themselves. The *strymonas* library for Scala and OCaml [Kiselyov et al. 2017], ScalaBlitz for Scala [Prokopec and Petrashko 2013], the fold-based fusion technique

for Scala by Jonnalagedda and Stucki [2015], and LinqOptimizer for C# and F# [Palladinos and Rontogiannis 2014] follow that approach. These techniques cannot be adapted to Java, because it lacks the necessary language features. Also, our goal is to enable optimization for Java's existing stream library, not to replace it.

The stream library for Java by Biboudis et al. [2015] aims for extensibility, not to reach the performance of imperative code.

C# supports declarative data processing in the form of Language-Integrated Query (LINQ), which suffers from similar performance problems as Java streams compared to hand-optimized code. The Steno tool [Murray et al. 2011] makes it possible to translate declarative LINQ queries into imperative code, using iterator fusion and nested loop generation optimizations. Earlier work has applied similar approaches as Steno for Common Lisp and Pascal programs [Waters 1991]. The key difference to our technique is that Steno relies on hardwired knowledge about the semantics of all the available LINQ operators and thus does not need to look at their implementations; in contrast, our approach is not limited to a specific API but instead relies on static analysis of the stream library implementation.

Also for C#, Adamus et al. [2015] have developed a technique for optimizing LINQ queries by identifying free expressions in nested queries. By lifting these expressions out of the nested query they can avoid redundant re-computation at run-time, thus improving performance. Their technique builds on the idea of rewriting stream pipelines, and is not concerned with the performance overhead of using LINQ queries compared to hand-optimized code.

10 CONCLUSION

Streams are a powerful abstraction mechanism in Java programming, but they incur a large performance overhead, which JIT optimization has been unable to mitigate. In this work we exploit the fact that stream pipelines are relatively small pieces of code, which makes them amenable to high-precision interprocedural analysis and optimization. We have demonstrated the feasibility of AOT optimization of Java stream pipelines. By aggressively applying method inlining and stack allocation transformations driven by a static type/pointer analysis, our experimental results show that a variety of stream pipelines can be automatically transformed into efficient imperative-style code that has much better performance characteristics. For 10 of 11 micro-benchmarks, the resulting bytecode is as effective as hand-written imperative-style code, and 77% of 6 879 stream pipelines found in real-world Java programs are optimized successfully. Since the optimizer is fast (even for a prototype implementation) and structured as a bytecode-to-bytecode transformer, it is easily deployed in ordinary build processes. Moreover, the approach is not restricted to Java's push-style stream implementation but also produces good results for a simple pull-style library.

The experimental results also identify opportunities for future work. Most importantly, more pipelines could be optimized if the analysis is improved to be able to reason more accurately about short-circuiting operations. Also, incorporating relational analysis can lead to improved precision necessary for optimizing certain operations when using a pull-style stream library.

ACKNOWLEDGMENTS

This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 647544).

REFERENCES

Radoslaw Adamus, Tomasz Marek Kowalski, and Jacek Wislicki. 2015. A step towards genuine declarative language-integrated queries. In *2015 Federated Conference on Computer Science and Information Systems, FedCSIS 2015, Lódz, Poland, September 13-16, 2015*, Vol. 5. IEEE, 935–946. https://doi.org/10.15439/2015F156

Ole Agesen. 1995. The Cartesian Product Algorithm: Simple and Precise Type Inference Of Parametric Polymorphism. In *ECOOP'95 - Object-Oriented Programming, 9th European Conference, Århus, Denmark, August 7-11, 1995, Proceedings (Lecture Notes in Computer Science)*, Vol. 952. Springer, 2–26. https://doi.org/10.1007/3-540-49538-X_2

Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. 2005. A Survey of Adaptive Optimization in Virtual Machines. *Proc. IEEE* 93, 2 (2005), 449–466. https://doi.org/10.1109/JPROC.2004.840305

Matthew Arnold, Stephen J. Fink, Vivek Sarkar, and Peter F. Sweeney. 2000. A comparative study of static and profile-based heuristics for inlining. In *Proceedings of ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo 2000), Boston, MA, USA, January 18, 2000*. ACM, 52–64. https://doi.org/10.1145/351397.351416

John Aycock. 2003. A brief history of just-in-time. *ACM Comput. Surv.* 35, 2 (2003), 97–113. https://doi.org/10.1145/857076.857077

Aggelos Biboudis, Nick Palladinos, George Fourtounis, and Yannis Smaragdakis. 2015. Streams a la carte: Extensible Pipelines with Object Algebras. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic (LIPIcs)*, Vol. 37. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 591–613. https://doi.org/10.4230/LIPIcs.ECOOP.2015.591

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

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*. ACM, 243–262. https://doi.org/10.1145/1640089.1640108

Zoran Budimlic and Ken Kennedy. 1997. Optimizing Java: theory and practice. *Concurrency - Practice and Experience* 9, 6 (1997), 445–463.

Zoran Budimlic and Ken Kennedy. 1998. *Static interprocedural optimizations in Java.* Technical Report. Center for Research on Parallel Computation, Rice University, Technical Report CRPC-TR98746.

David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. 1986. Interprocedural constant propagation. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction, Palo Alto, California, USA, June 25-27, 1986*. ACM, 152–161. https://doi.org/10.1145/12276.13327

David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. 1990. Analysis of Pointers and Structures. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*. ACM, 296–310. https://doi.org/10.1145/93542.93585

Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. 1999. Escape Analysis for Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, Colorado, USA, November 1-5, 1999*. ACM, 1–19. https://doi.org/10.1145/320384.320386

Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP'95 - Object-Oriented Programming, 9th European Conference, Århus, Denmark, August 7-11, 1995, Proceedings (Lecture Notes in Computer Science)*, Vol. 952. Springer, 77–101. https://doi.org/10.1007/3-540-49538-X_5

David Detlefs and Ole Agesen. 1999. Inlining of Virtual Methods. In *ECOOP'99 - Object-Oriented Programming, 13th European Conference, Lisbon, Portugal, June 14-18, 1999, Proceedings (Lecture Notes in Computer Science)*, Vol. 1628. Springer, 258–278. https://doi.org/10.1007/3-540-48743-3_12

Julian Dolby and Andrew A. Chien. 1998. An Evaluation of Automatic Object Inline Allocation Techniques. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98), Vancouver, British Columbia, Canada, October 18-22, 1998*. ACM, 1–20. https://doi.org/10.1145/286936.286943

Julian Dolby, Stephen J. Fink, and Manu Sridharan. 2010. T.J. Watson Libraries for Analysis. http://wala.sourceforge.net/

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, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. ACM, 57–76. https://doi.org/10.1145/1297027.1297033

Manohar Jonnalagedda and Sandro Stucki. 2015. Fold-based fusion as a library: a generative programming pearl. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala, Scala@PLDI 2015, Portland, OR, USA, June 15-17, 2015*. ACM, 41–50. https://doi.org/10.1145/2774975.2774981

John B. Kam and Jeffrey D. Ullman. 1977. Monotone Data Flow Analysis Frameworks. *Acta Inf.* 7 (1977), 305–317. https://doi.org/10.1007/BF00290339

Raffi Khatchadourian, Yiming Tang, and Mehdi Bagherzadeh. 2020a. Safe Automated Refactoring for Intelligent Parallelization of Java 8 Streams. *Science of Computer Programming* (2020), 102476. https://doi.org/10.1016/j.scico.2020.102476

Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Baishakhi Ray. 2020b. An Empirical Study on the Use and Misuse of Java 8 Streams. In *Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science)*, Vol. 12076. Springer, 97–118. https://doi.org/10.1007/978-3-030-45234-6_5

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. ACM, 285–299. https://doi.org/10.1145/3093333.3009880

Ondrej Lhoták and Laurie J. Hendren. 2003. Scaling Java Points-to Analysis Using SPARK. In *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings (Lecture Notes in Computer Science)*, Vol. 2622. Springer, 153–169. https://doi.org/10.1007/3-540-36579-6_12

Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. 2017. Understanding the use of lambda expressions in Java. *PACMPL* 1, OOPSLA (2017), 85:1–85:31. https://doi.org/10.1145/3133909

Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empirical Software Engineering* 22, 6 (2017), 3219–3253. https://doi.org/10.1007/s10664-017-9512-6

Derek Gordon Murray, Michael Isard, and Yuan Yu. 2011. Steno: automatic optimization of declarative queries. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. ACM, 121–131. https://doi.org/10.1145/1993498.1993513

Erik M. Nystrom, Hong-Seok Kim, and Wen-mei W. Hwu. 2004. Importance of heap specialization in pointer analysis. In *Proceedings of the 2004 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'04, Washington, DC, USA, June 7-8, 2004*. ACM, 43–48. https://doi.org/10.1145/996821.996836

Oracle. 2014a. Java Microbenchmarking Harness. http://openjdk.java.net/projects/code-tools/jmh/

Oracle. 2014b. java.util.stream documentation for JDK 8. https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html

Oracle. 2014c. JDK 8. https://openjdk.java.net/projects/jdk8/

Nick Palladinos and Kostas Rontogiannis. 2014. LinqOptimizer: An automatic query optimizer for LINQ to Objects and PLINQ. http://nessos.github.io/LinqOptimizer/

Young Gil Park and Benjamin Goldberg. 1992. Escape Analysis on Lists. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI), San Francisco, California, USA, June 17-19, 1992*. ACM, 116–127. https://doi.org/10.1145/143095.143125

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, SCALA@SPLASH 2017, Vancouver, BC, Canada, October 22-23, 2017*. ACM, 29–40. https://doi.org/10.1145/3136000.3136002

Aleksandar Prokopec and Dmitry Petrashko. 2013. ScalaBlitz: Lightning-fast Scala collections framework. https://scala-blitz.github.io/

John Rose. 2015. Hotspot-dev mailing list: Perspectives on Streams Performance. http://mail.openjdk.java.net/pipermail/hotspot-compiler-dev/2015-March/017278.html

Ulrik Pagh Schultz, Julia L. Lawall, and Charles Consel. 2003. Automatic program specialization for Java. *ACM Trans. Program. Lang. Syst.* 25, 4 (2003), 452–499. https://doi.org/10.1145/778559.778561

Denys Shabalin and Martin Odersky. 2018. Interflow: interprocedural flow-sensitive type inference and method duplication. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala, SCALA@ICFP 2018, St. Louis, MO, USA, September 28, 2018*. ACM, 61–71. https://doi.org/10.1145/3241653.3241660

Micha Sharir and Amir Pnueli. 1981. *Two approaches to interprocedural data flow analysis.* Prentice-Hall, Chapter 7, 189–234.

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. ACM, 17–30. https://doi.org/10.1145/1926385.1926390

Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs)*, Vol. 56. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.22

Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*. ACM, 387–400. https://doi.org/10.1145/1133981.1134027

Philip Wadler. 1990. Deforestation: Transforming Programs to Eliminate Trees. *Theor. Comput. Sci.* 73, 2 (1990), 231–248. https://doi.org/10.1016/0304-3975(90)90147-A

Richard C. Waters. 1991. Automatic Transformation of Series Expressions into Loops. *ACM Trans. Program. Lang. Syst.* 13, 1 (1991), 52–98. https://doi.org/10.1145/114005.102806