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

Presented by: Kyle Wynne and Alex Mozeak

# Introduction

- Functional language features are being integrated into mainstream languages, most notably Java.
- Features such as lambda functions and the Stream API allow for functional-style processing of data.
- Functional programming benefits are well known. For example, it is easier to read and write than the imperative style, particularly for more complex computations.

# The Issue

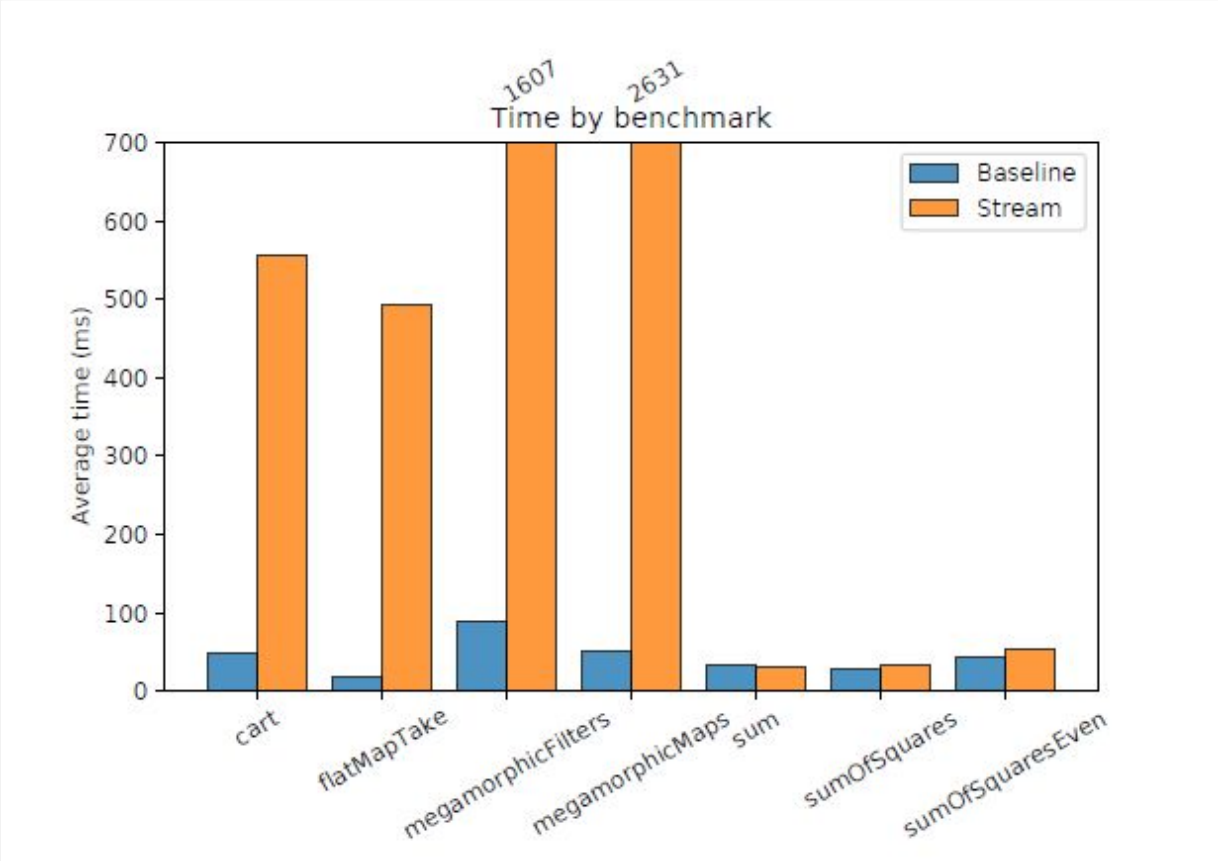
- Functional style programming using streams has performance issues.
- Imperative-style alternatives prove to be much faster.
- Stream pipelines strength, easability to switch to parallel processing, is not an ideal situation either.

*“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.”*

- *Project Cassandra Developer*

# Example

Performance of baseline imperative implementation versus sequential Java streams.



# The Typical Approach to Java Code Optimization

“In Java, the main approach to code optimization has been part of just-in-time (JIT) compilation in the virtual machine.”

It is believed the JIT optimizer is able to make better decisions at run-time.

JIT compiler must make fast decisions about what and when to optimize while running the program.

# Ahead-of-Time Optimizations

Ahead-of-Time (AOT) optimization techniques have advantages over JIT optimizations.

An AOT optimizer can have time to perform a more precise analysis of the program.

AOT allows the developer to know before execution if an optimization was successful.

AOT optimizations can be deployed without modifications to the JVM installation.

# Stream Pipelines

```
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  }
```

# Stream Pipeline Inefficiency

The main reason for stream pipelines being less efficient is due to the virtual calls.

A Java stream pipeline with N elements and depth K will require up to N x K virtual calls in order to push the elements through the pipeline.

In order to eliminate this inefficiency the number of calls must be addressed.

# Eliminating the Overhead

- The optimization builds on this idea of eliminating the constant virtual calls.
- Focus on optimizing sequential stream pipelines, as these are 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.”*

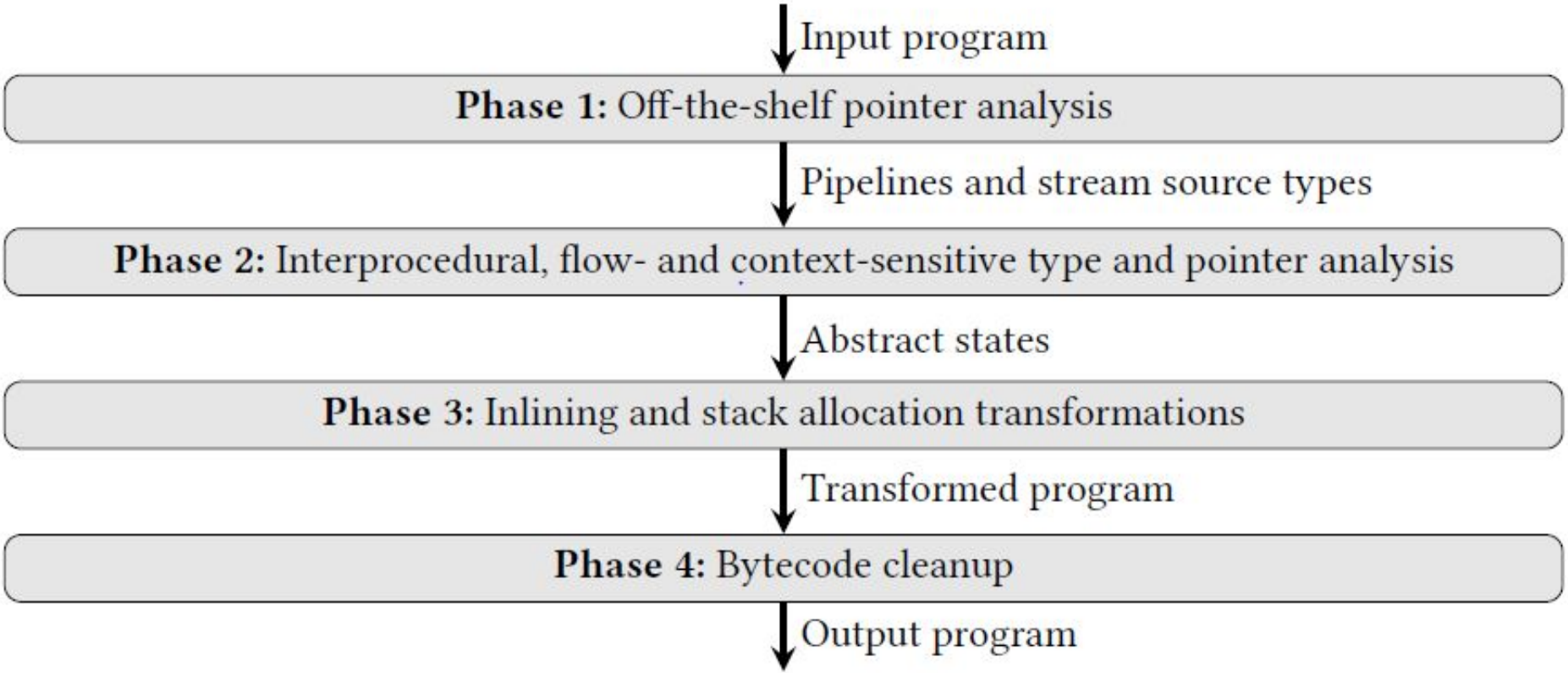
# Optimization Approach

**Phase 1: Pre-analysis**
The compiled program is analyzed with an off-the-shelf pointer analysis. Stream pipeline segments are located within the program.

**Phase 2: Interprocedural Analysis**
Interprocedural Analysis of the pipelines. The result consists of an abstract state for every analyzed flow.

**Phase 3: Inlining and Stack Allocation**
Method inlining and stack allocation transformations.

**Phase 4: Cleanup**
This phase involves the cleanup of redundancies which resulted from the transformations in phase 3.




Flow diagram of the approach.

# Phase 1: Pre-analysis

The first phase is a preliminary analysis meant to identify stream pipelines within the analyzed program.

- Usually trivial to find all local variables of type *Stream*.
- The authors leave room to improve on identifying stream objects that are not local (passed as parameters, stored in objects, etc.)

The first phase also restricts the set of possible concrete types for stream sources.

- Trivial to identify in cases such as in Slide 7 (ArrayList is the concrete type).
- AOT compilation affords time where a dedicated pointer analysis tool can be used to identify less trivial sources.

This allows the optimization to focus on only the parts of the program where it is required.
The concrete types of the stream sources are used in the interprocedural analysis in Phase 2.

# Phase 2: Interprocedural (IP) Analysis

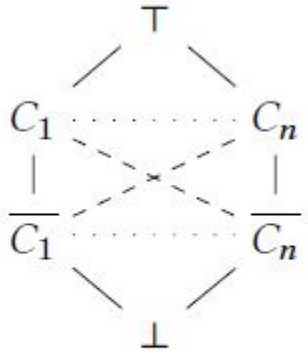
The goal here is to map out the method calls of the program and determine an abstract heap of program values.

- *“After finishing the analysis, we have the information necessary to unambiguously resolve the calls at every analyzed call site.”*

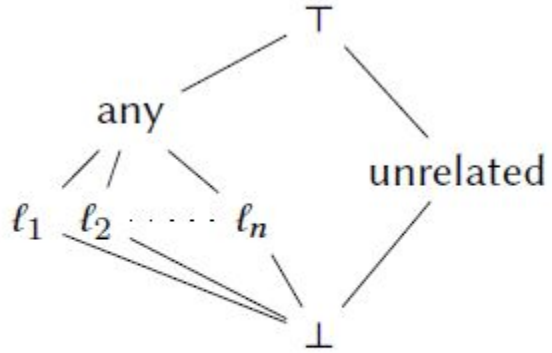
A lattice of abstract state is used. Values = Type x Pointer.

- Type lattice is used for all classes, interfaces, primitive types, and array types.
- Pointer lattice is used for allocation sites.

# Phase 2: Interprocedural Analysis Cont.




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




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

# Phase 2: Interprocedural Analysis Cont.

Transfer functions are generated for each method call.

- Functions that model the output for each possible input.
- If the source type from Phase 1 is still kind of fuzzy, the transfer function can over-approximate the potential outputs.

The analysis can fail, causing optimization to be aborted for a given stream pipeline.

- If the abstract state becomes too imprecise.
- If a transfer function has to over-approximate too much.

# Phase 3: Inlining and Stack Allocation

**Inlining:**

- Transformation at method call instruction is similar to the IP analysis.
  - *Callee* gets expanded into a method with **n** arguments and **m** local variables.
  - *Caller* gets **m** local variables to be passed into the *callee*.
- If the *callee* cannot be uniquely determined, inlining will not be applied at this call.
  - May be called by other *callers* that do not have the same abstract representation.
  - Remember, analysis does not cover the whole program.
  - For this reason, inlining is not directly suitable for optimizing parallel streams.

**Stack Allocation:**

- All instructions that access the object's fields must be able to be inlined.
- If this condition is unable to be met, the object is ineligible for stack allocation.

Optimization using this technique is “all or nothing”.

# Phase 4: Cleanup

A lot of redundancies are added to the bytecode by Phase 3. A cleanup process is thus invoked, reducing local variables by a factor of 10 to 40 and bytecode instructions by a factor of 10 in the transformed code.

```
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  }
```

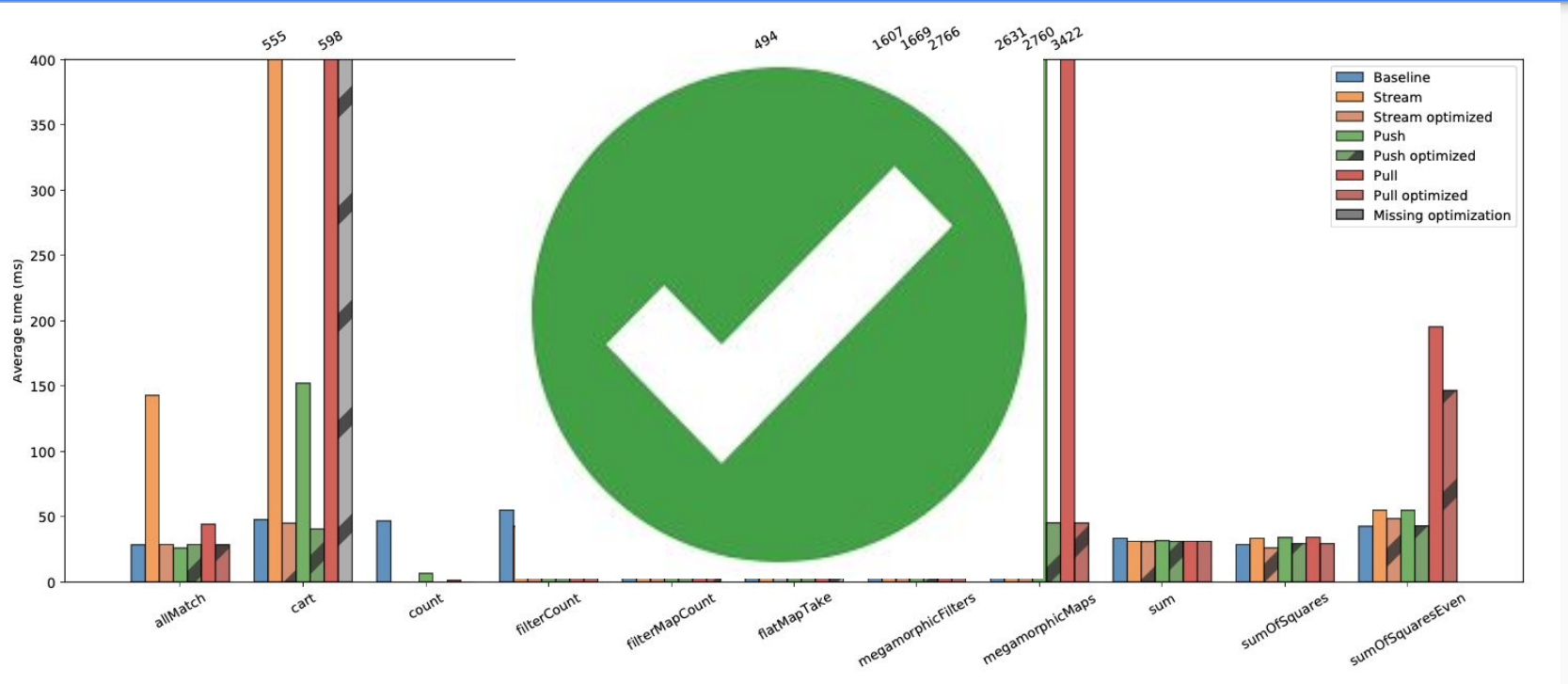
```
173  int f_opt(int i) {
174    ILOAD 0
175    ISTORE 1    (removed)
176    ILOAD 1     (removed)
177    ICONST_2
178    IMUL
179    ...
180  }
```

# Conclusion - Overview

The authors name their implementation of the approach *Streamliner*. They attempt to answer two research questions:

1. *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?*
2. *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?*

# Conclusion - 1



# Conclusion - 2

| 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% |

# Implications for Reactive Programming

*Streamliner* makes Java stream pipelines run faster.

Reactive Programming processes event streams in an, ideally, purely functional manner.

Thus, compiling Java code ahead-of-time using the proposed techniques can make programs more *reactive*, a net benefit.

# References

Anders Mùller 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

# Thanks!