Characterizing Java Streams in the Wild

Eduardo Rosales[1], Andrea Rosà[2], Matteo Basso[3], Alex Villazón[4], Adriana Orellana[5], Ángel Zenteno[6], Jhon Rivero[7], Walter Binder[8]
[1 2 3 8] Università della Svizzera italiana (USI), Switzerland
Email:[1]rosale@usi.ch, [2]andrea.rosa@usi.ch, [3]matteo.basso@usi.ch, [8]walter.binder@usi.ch
[4 5 6 7] Universidad Privada Boliviana, Bolivia
Email:[4]avillazon@upb.edu, [5]adrianaorellana1@upb.edu, [6]angelzenteno1@upb.edu, [7]jhonrivero1@upb.edu

***Abstract*—Since Java 8, streams ease the development of data transformations using a declarative style based on functional programming. Some recent studies aim at shedding light on how streams are used. However, they consider only small sets of applications and mainly apply static analysis techniques, leaving the large-scale analysis of dynamic metrics focusing on stream processing an open research question. In this paper, we present the first large-scale empirical study on the use of streams in Java. We present a novel dynamic analysis for collecting runtime information and key metrics that enable the fine-grained characterization of sequential and parallel stream processing. We massively apply our dynamic analysis using a fully automated approach, supported by a distributed infrastructure to mine public software projects hosted on GitHub. Our findings advance the understanding of the use of streams, both confirming some of the results of previous studies at a much larger scale, as well as revealing previously unobserved findings in the use of streams.**

***Index Terms*—Java streams, Stream processing, Code repositories, Empirical studies, Large-scale study, Characterization, Dynamic program analysis, GitHub, Java Virtual Machine**

I. Introduction

The *Stream API* [1] was added in Java 8 to allow developers perform data processing using a declarative and concise style based on functional programming [2]. The Stream API provides two main abstractions. First, the *stream*, which represents a sequence of elements that comes from a data source. Second, the stream *pipeline*, a sequence of operations that are applied to the elements in the stream upon execution. The operations in a pipeline allow performing actions such as mappings, searches, filtering, or data reductions. Streams can be created from diverse data sources, including collections, arrays, and files. Streams are versatile and are often a good option to implement algorithms that uniformly transform a sequence of elements to obtain a result [3]. Streams can be used to improve software design, making an application easier to extend and maintain. Moreover, a key feature of streams is that they can be processed in parallel just by calling a single method [4].

Recently, streams have received the attention of researchers studying their degree of adoption. At the time of writing, three studies analyzed the use of streams in the wild. Tanaka et al. [5] mine 100 software projects to study the use of *lambda expressions* [3], streams, and the `java.util.Optional` class. Khatchadourian et al. [6] examine 34 Java projects to study the use of streams in Java. This work represents the first attempt to specifically discover use cases of the Stream API. Finally, Nostas et al. [7] study the use of streams in 610 Java projects. This work is a partial replication of the study of Khatchadourian et al. considering a larger number of projects and validating most of the results previously obtained. Unfortunately, the aforementioned studies consider only a small number of projects. Furthermore, the above studies rely mainly on manual code inspection and static analysis techniques, which makes it impossible to analyze dynamic metrics unique to streams. Specifically, there is yet limited empirical evidence to make general and practical conclusions on how the Stream API is used.

Contributions. To bridge this gap, we first present Stream-Analyzer, a novel dynamic program analysis (DPA) specifically designed for the characterization of stream processing on the Java Virtual Machine (JVM). Stream-Analyzer accurately detects each stream used by a Java application, properly targeting all forms of stream creation and execution enabled by the Stream API. Conducting a large-scale study on the use of streams requires detecting stream processing exposed by a wide and diverse range of Java applications along with collecting representative information that best characterize streams. To this end, Stream-Analyzer specifically targets a careful selection of metrics suitable to be massively collected in the wild and whose analysis enables a fine-grained characterization of stream processing. Our tool is designed to run on top of a container-based, distributed infrastructure [8] for executing custom analyses on code repositories without requiring manual intervention or manual code inspection.

We use Stream-Analyzer to conduct the first large-scale empirical study on the use of streams in Java. Our study targets open-source software projects publicly available in GitHub [9], for a total of 1 808 415 projects analyzed, among which we find 132 211 projects that use the Stream API. Our work analyzes a total of 620 033 509 streams. On the one hand, we confirm that the findings of previous studies [5]–[7] conducted at a smaller scale also hold at a large scale. On the other hand, we provide new insights on how streams are used, thanks to the analysis of runtime information previously overlooked.

Outline. The rest of the paper is organized as follows. Sec. II provides background information. Sec. III describes the novel DPA Stream-Analyzer. Sec. IV presents our large-scale empirical study. Sec. V discusses the limitations of our work. Sec. VI reviews work related to our approach. Finally, Sec. VII presents our conclusions.

```
1  transactions.stream()
2    .parallel()
3    .filter(t -> t.getStatus() == Transaction.VALID)
4    .map(Transaction::getID)
5    .collect(Collectors.toSet());
```

Fig. 1: A simple stream example.

II. Background

This section introduces key concepts and the terminology used in our study.

Stream and pipeline. A *stream* is a sequence of data elements supporting either sequential or parallel operations that are structured in stages within an associated *pipeline*. The elements in a stream can come from multiple data sources including collections, generators of pseudo-random numbers, arrays, files, and strings [3].

Fig. 1 shows a simple Java code example of a stream. In line 1, a stream is generated from `transactions`, i.e., a list of objects of class `Transaction` (whose declaration and details are omitted here and are not relevant). The pipeline contains four operations: `parallel` which parallelizes the execution of the pipeline (line 2), `filter` that discards invalid transactions (line 3), `map` which extracts the identifiers from the remaining transactions (line 4), and `collect` that here accumulates the identifiers of valid transactions into a set (line 5). Therefore, the stream in the example is used to collect the set of IDs of valid transactions available in the list `transactions`.

Source method. *Stream creation* takes place upon the call to a *source method*, i.e., the method used to create a stream. In the example, `stream` (line 1) is the source method, which is defined in the interface `java.util.Collection` and here generates a stream from the data source `transactions`.

Source collection type. Many stream data sources implement the interface `Collection`[1]. We call *source collection type* the specific collection type from which a stream is created, if any. In the example, since `transactions` is a list, the source collection type is a concrete implementation of `java.util.List`, e.g., `java.util.ArrayList`.

Stream type. The Stream API supports four types of streams. The interface `java.util.stream.Stream` models a stream of objects, while the interfaces `java.util.stream.IntStream`, `java.util.stream.LongStream`, and `java.util.stream.DoubleStream` model streams of primitive types. In the example, we use an object-based stream type.

Characteristics of the data source. A stream has associated a *spliterator*, i.e., the parallel analogue of an iterator; it describes a (possibly infinite) collection of elements, with support for sequentially advancing, bulk traversal, and splitting off some portion of an input data for parallel computation [1]. Among others, the characteristics of the spliterator define whether the data source has an *encounter order* (i.e., the data source makes its elements available in a defined order), is *sorted* (i.e., the elements in the data source have a sort order), is *concurrent* (i.e., the data source is designed to handle concurrent modification), is *distinct* (i.e., the data source does not allow duplicates) or is *immutable* (i.e., data source elements cannot be added, replaced, or removed). The spliterator may also report the *data-source size*, i.e., the total number of elements in the data source, if known. Upon stream creation, this size is the total number of elements available in the data source. In the example, the characteristics of the data source upon stream creation depend on the concrete implementation of `List` used at runtime to create the stream. For instance, by default an `ArrayList` has an encounter order (the elements are encountered in index order) and is neither sorted, nor concurrent, nor immutable, nor distinct, while the data-source size would correspond to the number of elements in the collection.

Operations. Upon stream creation, a pipeline is generated and it may have associated operations. Operations are divided into *intermediate* (i.e., operations that produce other streams that can be further processed) and *terminal* (i.e., operations triggering the execution of the stream).

Intermediate operations. Intermediate operations are *lazy*, i.e., they do not perform any processing until a terminal operation is invoked [1]. Intermediate operations can be *stateless* or *stateful*. In stateless operations, each element can be processed independently from operations on other elements, while in stateful operations, the current state may depend on the state of previously seen elements [1]. An example of a stateful operation is `limit`, which truncates elements such that the size of the resulting stream is no longer than a given length [10]. In Fig. 1, `parallel`, `filter`, and `map` are all stateless intermediate operations.

Terminal operations. *Stream execution* takes place only if a stream has a terminal operation. The terminal operation triggers the *traversal* of the pipeline either to return a result (e.g., an array) or to produce side effects (e.g., to print all elements). A stream can have at most one terminal operation, which can be executed only once [1]. In the example, `collect` is the terminal operation that triggers stream execution.

Execution mode. A stream has an *execution mode* defining whether the stream is to be executed sequentially or in parallel. When a parallel stream is executed, typically pipeline traversal is performed by fork/join[2] tasks, all of which execute in a fork/join pool [12].

The execution mode of a stream is first set upon stream creation. In the example, method `stream` creates a sequential stream. Alternatively, calling method `parallelStream` would create a parallel stream. The execution mode can be switched by calling the `sequential` or `parallel` intermediate operations. In the example, the execution mode is parallel.

Length. A pipeline has a *length*, i.e., the total number of operations in the pipeline. In the example, the length is

[1]The fully qualified name of a class/interface appears upon first occurrence in the text; thereafter, we report only the class/interface name.

[2]Fork/join parallelism recursively splits (*fork*) work into tasks that are executed in parallel, waiting for them to complete, and then typically merging (*join*) the results produced by the forked tasks [11].

four, as the operations `parallel`, `filter`, `map`, and `collect` compose the pipeline.

Collector. In line 5, `collect` performs a *reduction* using a *collector*, i.e., an implementation of the interface `java.util.stream.Collector` that enables mutable reduction operations, such as accumulating elements into collections or summarizing elements according to various user-defined criteria [13]. In the example, the collector is the object returned by `Collectors.toSet()`, which accumulates input elements into a set.

Stream result type. After execution, the stream produces an output or a side effect. We call *stream result type* the type of the output produced after the stream is executed (typically a collection, an array, or a scalar). In the example, the stream result type is the concrete implementation of `java.util.Set` used at runtime by the collector, i.e., `java.util.HashSet`.

III. Stream-Analyzer

In this section, we introduce Stream-Analyzer, our DPA to collect dynamic metrics on streams. We first detail the events and entities targeted by Stream-Analyzer as well as the runtime information it collects (Sec. III-A). Then, we provide implementation details of Stream-Analyzer (Sec. III-B).

A. Metric Collection

Stream-Analyzer is a novel DPA specifically designed to characterize stream processing in the wild. Stream-Analyzer collects runtime information and metrics unique to streams, which enable a detailed characterization of both sequential and parallel stream processing on the JVM. To do so, Stream-Analyzer intercepts all forms of stream creation and execution available in the Stream API, accurately detecting each stream used in a Java application. Moreover, Stream-Analyzer targets a careful selection of information and metrics suitable to be massively collected in the wild. The rest of the section explains the data collected by our DPA, discriminating between the events related to stream creation and execution.

Stream creation. Upon stream creation, a pipeline associated to the new stream is created. We consider as a stream every instance of the interface `java.util.BaseStream`, the top-level interface of the Stream API [14]. We consider as a pipeline every subtype of the abstract class `java.util.stream.AbstractPipeline`.

In the Stream API, pipeline creation involves instantiating a new object that represents the first stage of the pipeline, i.e., the *head*. This object is created via the constructors of the subtypes of `AbstractPipeline`, i.e., `ReferencePipeline$Head`, `IntPipeline$Head`, `LongPipeline$Head`, or `DoublePipeline$Head`. All of these classes belong to the `java.util.stream` package and are the core implementations of the `Stream` interface and the primitive stream types (`IntStream`, `LongStream`, and `DoubleStream`, respectively). Stream-Analyzer uses the reference to the object representing the head of the pipeline (which is associated to a single stream) to produce a unique ID identifying the stream. This identifier is crucial to associate all collected information to the creation and execution of a specific stream.

The creation of a stream takes place in two ways, either by using the class `java.util.stream.StreamSupport`, which provides low-level methods to create streams using a spliterator, or by calling one of such methods through a wrapper method in other classes, e.g., `Collections.stream`. Stream-Analyzer detects both ways of creating streams to collect the source method. In addition, the stream data source may implement the interface `Collection`, which Stream-Analyzer instruments to collect the source collection type, if any. Stream-Analyzer also intercepts stream creation to query the characteristics of the spliterator associated to the new stream. Finally, Stream-Analyzer collects the data-source size as reported by the spliterator, a key metric that enables quantifying the number of elements in the data source upon stream creation.

Once a pipeline is created, operations can be appended to it. Stream-Analyzer intercepts calls to all methods enabling the appending of intermediate operations as defined in the interfaces `BaseStream`, `Stream`, `IntStream`, `LongStream` and `DoubleStream`. In the Stream API, appending an intermediate operation to a pipeline typically involves instantiating a new object representing the new stage. Stream-Analyzer uses the reference to this new stage of the pipeline to produce an ID which uniquely identifies the operation appended. Stream-Analyzer also captures a reference to the previous stage of the pipeline, which can be either the head or another operation. In such a way, having a unique identifier for each stage of the pipeline is crucial to subsequently attribute a newly detected operation to a specific pipeline. Stream-Analyzer collects the name of the method used to append an intermediate operation and its type. Overall, the data collected upon stream creation enables describing how a stream is generated, characterizing in detail the data source, and the set of transformations performed by intermediate operations in the pipeline.

Stream execution. Stream execution requires the invocation of a terminal operation which triggers the traversal of the pipeline. Stream-Analyzer tracks calls to all methods triggering stream execution that are defined in the interfaces `BaseStream`, `Stream`, `IntStream`, `LongStream`, and `DoubleStream`. Differently from intermediate operations, in the Stream API calling a terminal operation does not involve appending a new stage to the pipeline. As a result, Stream-Analyzer identifies and attributes a terminal operation to a pipeline by capturing the reference to the last stage of the pipeline before the terminal operation is called on it. Stream-Analyzer also collects the name of the method used to invoke the terminal operation and the stream result type. When the terminal operation performs a reduction via `collect`, Stream-Analyzer captures the name of the collector used. Finally, the length of the pipeline is computed along with the execution mode of the stream. The length is a key metric quantifying the set of data transformations done through a pipeline. Overall, the collected information is used to analyze how a pipeline is traversed, describing the kind of data processing that is performed, and the output produced by a stream.

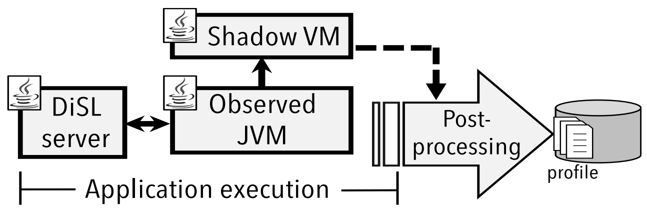

Fig. 2: High-level architecture of Stream-Analyzer.

B. Implementation

Fig. 2 shows the high-level architecture of Stream-Analyzer. Our DPA is built upon *DiSL* [15], a framework for the JVM to perform dynamic analysis via bytecode instrumentation. The DiSL weaver guarantees *full bytecode coverage*, i.e., DiSL instruments every Java method with a bytecode representation, enabling the complete instrumentation of the Java class library, which is notoriously hard to instrument [16]. This is important for our purposes as the Stream API is fully implemented within the Java class library, allowing Stream-Analyzer to accurately profile all streams used by an application, as well as all targeted events related to stream creation and execution.

A native agent attached to the JVM executing the application under analysis (hereafter called *observed JVM*) intercepts classloading, sending loaded classes to the *DiSL server*, i.e., a separate JVM where DiSL is deployed. In the DiSL server, the instrumentation logic determines the methods to be targeted and inserts the instrumentation code needed to collect the target metrics. Then, the instrumented classes are returned to the observed JVM where they are linked [17], allowing Stream-Analyzer to characterize stream processing.

To further isolate the analysis from the execution of the observed JVM, Stream-Analyzer uses *Shadow VM* [18], a deployment setting of DiSL which allows running analyses in a separate JVM process, mitigating the perturbations incurred by the inserted instrumentation code while also preventing known issues inherent to non-isolated approaches [19]. Upon collection, the information and metrics are sent to the Shadow VM, which contains most of the logic and data structures supporting the analysis of stream processing. This is possible thanks to a second native agent (sending data to the Shadow VM) attached to the observed JVM. Finally, upon application completion, Stream-Analyzer performs offline post-processing of the collected data, producing a single integrated *profile* containing all the information collected for each stream used in the observed application. The profile of each observed JVM is stored in a database, which can be queried later to produce aggregated data and statistics on all analyzed applications.

IV. Large-scale Analysis

In this section, we describe our approach to characterize streams in the wild as well as the findings of our study. We first detail the methodology used for our analysis (Sec. IV-A). Then, we present the results of our study (Sec. IV-B).

A. Methodology

All the analyzed projects are open source Java applications that are publicly available in GitHub. We considered only projects with at least one commit in year 2020 (until October). We analyzed the latest version (commit) of the projects in that period. Our choice is motivated by the need to complete our study in the wild in a reasonable time with the available resources. Furthermore, the chosen timeframe avoids old, obsolete, or inactive software projects that could bias the study.

For the purposes of mining GitHub to search for Java applications, we use *NAB* [8], a distributed infrastructure for automatically executing custom analyses on open-source projects hosted in public code repositories. A key feature of NAB is that is provides a plugin mechanism to integrate existing analyzes so that they can be applied to software repositories. We leverage this feature to run Stream-Analyzer on top of NAB to massively characterize stream processing in the wild. Another important feature of NAB is that it can be deployed using containers. Containerization is crucial to our purposes as it enables isolating the underlying execution environment and operating system, to prevent potential issues triggered while dynamically analyzing unverified software projects that may contain buggy or even harmful code. Moreover, the use of containers eases parallelizing analysis execution by leveraging multicores, along with simplifying the deployment of the analysis on multiple machines.

A key challenge in conducting our study is the need for automatically and massively analyzing the runtime behavior of software projects that were not designed or tested for this purpose, and hence may fail if DPAs are applied to them. A fully automatic approach should therefore first search for executable code available in a project and then attempt to execute it. NAB enables the analysis of code that can be executed through software tests, specifically *JUnit* [20] tests. In prior work [8], [21], we successfully located analyzable executable code in the wild, taking advantage of the fact that JUnit is a well-established and very popular testing framework for the JVM to implement software testing. We rely on NAB to analyze executable code exercised by JUnit tests, which can be built and run automatically from a build system. We target all projects using *maven* [22] as build system, which we use to download the project dependencies (as configured originally by the project developers), compile the sources, and run the available unit tests.

We use NAB to crawl GitHub looking for Java projects, clone each project found, and apply Stream-Analyzer to characterize the stream processing exposed by the application. While processing a candidate project, many kinds of failures can happen. At a very early stage, the cloning of the project can fail. In our approach, we retry cloning the project up to three times, after which we discard it from our analysis. Another challenge we face while analyzing projects in the wild is the need for handling failures both when attempting to build a project and when trying to execute the unit tests. In our approach, we discard the analysis of projects for which maven

TABLE I: Most popular source methods.

Source Method	Occurrences	%
`Collection.stream`	349 303 956	56.33
`IntStream.range`	87 677 566	14.14
`Arrays.stream`	58 690 188	9.46
`Collections$SynchronizedCollection.stream`	27 069 317	4.37
`Stream.of`	26 565 005	4.28
`Stream.concat`	22 869 945	3.69
`Random.ints`	12 020 051	1.94
`Collections$UnmodifiableCollection.stream`	11 795 112	1.90
`Stream.empty`	7 397 430	1.19
`CharSequence.chars`	5 001 524	0.81

TABLE II: Most popular source collection types.

Source Collection Type	Occurrences	%
`ArrayList`	258 713 462	41.72
`HashSet`	29 330 326	4.73
`Collections$SynchronizedSet`	25 569 216	4.12
`LinkedList`	12 246 909	1.98
`LinkedHashSet`	7 730 718	1.25
`Arrays$ArrayList`	7 480 889	1.21
`Collections$EmptyList`	5 759 041	0.93
`HashMap$EntrySet`	5 687 645	0.92
`Collections$SingletonList`	5 466 774	0.88
`Collections$UnmodifiableRandomAccessList`	4 806 994	0.78

fails either to build the sources or to locate and execute testing code. Moreover, even when the testing code can be executed, there is the need for dealing with buggy, malicious or non-terminating code. To this end, we set an *analysis timeout*, i.e., a maximum execution time for the analysis to complete. We set the analysis timeout to be one hour, after which the analysis of a project is halted to prevent the application from excessively consuming the available resources.

We initially analyzed a total of 1 808 415 Java projects. This extensive selection allows diversity in aspects such as size, domain, popularity, and testing practices, which is crucial to improve the representativeness of our study. From this initial set of projects, we discarded the ones that failed to clone, failed to build, lacked JUnit tests, failed to execute the unit tests, terminated due to an analysis timeout, or exposed no stream processing, resulting in 132 211 analyzed projects.

B. Results

In this subsection, we present the results of our study in the form of findings. When applicable, we compare them with the corresponding findings highlighted at a smaller scale by related work.

Source methods. We find a total of 47 different source methods among which we summarize the ten most popular ones in Table I. Our findings show that streams are created mainly from collections (62.60%). Other popular source methods include `IntStream.range` (returns a sequential `IntStream` of consecutive integers in a given range [23]), `java.util.Arrays.stream` (returns a sequential stream with the specified array as its data source [24]), `Stream.of` (returns a sequential stream generated from one or multiple data sources passed as arguments [10]), `Stream.concat` (returns a new stream containing all the elements of two streams passed as input [10]), `java.util.Random.ints` (returns an `IntStream` of pseudorandomly chosen integers [25]), `Stream.empty` (returns an empty sequential stream [10]) and `java.lang.CharSequence.chars` (returns an `IntStream` containing the integer representation of the char values from a source string [10]). We are not aware of any other study characterizing the source method from which streams generate.

Finding 1: *Streams are mainly created from collections, generators of integers, and arrays.*

Source collection types. We detect a total of 185 different source collection types, among which we summarize the ten most popular ones in Table II. Our findings show that streams are mainly generated from lists and sets. We are not aware of related work reporting the specific collection types from which streams are created. However, to some extent our results are similar to the ones reported by Costa et al. [26] while studying the use of collections in Java (10 986 projects analyzed). They find that `ArrayList`, `java.util.HashMap`, `HashSet`, `java.util.LinkedList`, `java.util.LinkedHashMap` and `java.util.LinkedHashSet`, among others, are the most popular Java collections.

Finding 2: *Streams generated from collections are mostly created from lists and sets.*

Stream types. We find that streams of objects are the most popular type of streams. 81.41% of the streams detected are of type `Stream`, 18.54% of type `IntStream`, 0.05% of type `LongStream`, and only 0.0011% are of type `DoubleStream`. Our findings show that the most popular streams are those whose elements are references to objects. As far as we are aware, previous studies do not report the popularity of the stream types used in the projects analyzed.

Finding 3: *Object-based stream types are far more popular than primitive-based stream types.*

Characteristics of the data source. We characterize the data source from which the stream is generated upon stream creation. We find that most of the streams are created from a data source having an encounter order (87.71%). This is expected as most streams generate from lists and arrays (data structures that typically preserve an index order). We find that most of the data sources allow duplicates (72.13%), which can also be explained by the fact that most of the streams are generated from lists and arrays. Lastly, we find that most of the streams are created from data sources not having a sort order (85.29%), which are mutable (68.09%), and which do not support concurrent modification (99.86%). While Khatchadourian et al. [6] report that streams in the analyzed projects are largely ordered, they mainly infer the ordering implemented by the stream data source via static analysis. We confirm this finding by detecting *ordering constraints* (i.e., the stream data source has an encounter and/or a sort order to be preserved) as reported at runtime by the spliterator upon stream creation.

Our findings indicate that most of the streams detected

TABLE III: Distribution of the stream data-source size.

Size Range	Occurrences	%
0	51 585 754	13.02
$[10^0, 10^1)$	309 639 158	78.15
$[10^1, 10^2)$	29 085 769	7.34
$[10^2, 10^3)$	4 620 480	1.17
$[10^3, 10^4)$	557 893	0.14
$[10^4, 10^5)$	1 692	$4 \cdot 10^{-4}$
$[10^5, 10^6)$	34	$1 \cdot 10^{-5}$
$[10^6, 10^7)$	5	$1 \cdot 10^{-6}$

TABLE IV: Most popular intermediate operations.

Intermediate Operation	Occurrences	%
`map`	427 009 693	51.57
`filter`	227 938 406	27.53
`mapToObj`	77 898 391	9.41
`flatMap`	34 512 362	4.17
`mapToInt`	27 327 892	3.30
`limit`	12 697 146	1.53
`mapToLong`	6 264 703	0.76
`sorted`	5 275 399	0.64
`skip`	4 041 440	0.49
`distinct`	2 877 764	0.35

may not be parallelized straightforwardly. As explained in the documentation of the Stream API, operations in the pipeline typically run in parallel more efficiently in the absence of ordering constraints [1].

Finding 4: *Stream data sources often allow duplicates, have ordering constraints, are mutable, and do not support concurrent modification.*

We were able to query the data-source size of 63.89% of all detected streams. For the remaining ones, the exact size was reported as unknown, which occurs when the data source is not sized or the size cannot be computed [27].

Table III reports the distribution of the data-source size (considering only data sources whose size could be successfully obtained). We find that most of the data sources used to generate streams contain a number of elements between 10^0 and 10^1. In this range, the most popular data-source sizes are 3 (23.88%), 2 (17.31%), 1 (13.13%), and 4 (10.21%). We also find that 13.02% of all the data sources analyzed contain zero elements upon creation. The maximum data-source size found is 1 114 112. Overall, we found that streams often generate from data sources containing few elements. To our knowledge, no other study analyzes the data-source size of streams or the characteristics reported at runtime by the spliterator associated to the stream.

Finding 5: *Stream data sources typically have few elements.*

Intermediate operations. We detect the occurrence of 827 977 847 intermediate operations in total. We show the ten most popular ones in Table IV. Our analysis shows that intermediate operations used for mapping (65.04%, considering operations `map`, `mapToObj`, `mapToInt`, `mapToLong`, and `mapToDouble`) and filtering (27.53%) are the most popular ones. This finding is consistent with the results by Tanaka et al. [5], Khatchadourian et al. [6], and Nostas et al. [7] (obtained at a much smaller scale), indicating that streams are mainly used to perform *MapReduce* [28] style processing.

TABLE V: Most popular terminal operations.

Terminal Operation	Occurrences	%
`collect`	197 338 242	31.83
`forEach`	145 774 791	23.51
`allMatch`	68 072 088	10.98
`findFirst`	51 274 578	8.27
`anyMatch`	43 892 114	7.08
`sum`	38 716 266	6.24
`findAny`	38 505 248	6.21
`reduce`	9 720 348	1.57
`toArray`	7 555 733	1.22
`count`	6 033 536	0.97

We find that both the `sorted` and the `unordered` intermediate operations (which respectively can introduce and remove ordering constraints among the elements of a stream) are barely used (0.64% and less than 0.001%, respectively). While considering these two operations, we confirm that most of the analyzed pipelines have ordering constraints due to the nature of the stream data source or due to transformations through the pipeline.

Khatchadourian et al. report that stateful operations are rarely used. We confirm this observation at a much larger scale, finding that stateless operations are by far more popular (96.99%). The exclusive use of stateless operations is recommended as according to the documentation of the Stream API, pipelines containing only stateless intermediate operations can be processed in a single pass, whether sequential or parallel, with minimal data buffering, potentially improving performance [1].

Finding 6: *Mapping and filtering are by far the most popular intermediate operations.*

Finding 7: *Stateful intermediate operations are barely used.*

Terminal operations. We detect the occurrence of 620 033 509 terminal operations in total and report the ten most popular ones in Table V. Our results show that the most used terminal operations are `collect` (31.83%) and `forEach` (i.e., applies a given function to each data element in a stream pipeline) (23.51%). While the former is key for map-reduce-like data processing, the latter indicates that streams are also used to process data iteratively and nondeterministically. This finding confirms the results by Tanaka et al., Khatchadourian et al, and Nostas et al. at a larger scale.

In addition, we find that deterministic terminal operations are the most used (70.28%). In this context, our results are consistent with the results reported by Khatchadourian et al. We find that `forEachOrdered` (i.e., the ordered and deterministic counterpart of `forEach`) is less popular (0.02%) than `forEach` (23.51%), and that `findFirst` (an operation returning the first element in the pipeline) is more popular (8.27%) than its nondeterministic counterpart `findAny` (6.21%), which returns any element in the pipeline.

TABLE VI: Distribution of the pipeline length.

Length	Occurrences	%
2	293 021 334	47.25
1	123 683 254	19.95
3	119 522 688	19.27
5	44 258 049	7.14
4	39 501 712	6.37
0	46 102	$7 \cdot 10^{-3}$
7	40 050	$6 \cdot 10^{-3}$
6	15 448	$2 \cdot 10^{-3}$
8	2 468	$4 \cdot 10^{-4}$
11	2 447	$4 \cdot 10^{-4}$
10	34	$5 \cdot 10^{-6}$
14	17	$3 \cdot 10^{-6}$

Finding 8: *Map-reduce-like data reductions via* `collect` *and iterative-style processing via* `forEach` *are the most common terminal operations.*

Finding 9: *Deterministic terminal operations are more popular than nondeterministic ones.*

Execution modes. Among a total of 620 093 603 streams detected, 620 033 509 were executed and 60 094 were not executed (i.e., streams that are created but lack a terminal operation). Among the streams executed, 99.66% were executed sequentially. This finding confirms the results obtained on a smaller scale by Khatchadourian et al. (34 projects and 1 038 streams analyzed, of which 13 were parallel) and Nostas et al. (610 projects analyzed, among which the authors report that in 23 at least a single parallel stream is created). Considering the low usage of stateful operations in the analyzed stream pipelines (as previously discussed), this finding may reveal potential missed speedups that could be obtained by parallelizing stream processing. Nonetheless, as pointed out by Lea et al. [4], when deciding whether to parallelize a stream, it is crucial to estimate if the sequential execution already exceeds a minimum threshold, which—as Lea et al. propose—could be measured in terms of execution time or the number of elements processed. The idea is finding whether, despite the presence of parallelization overheads, the parallel execution of a stream can result in performance gains. According to our findings, stream data sources typically contain few elements, therefore only a small selection of projects processing large amount of data may truly benefit from stream parallelization.

Finding 10: *Parallel streams are not popular.*

Lengths. Table VI reports, among all streams detected, the distribution of their pipeline length. Our results show that few operations are used in the projects analyzed, with an average length of only 2.34. We also find that 46 102 streams are created but no operation is called in their pipelines. This finding indicates that pipeline composition involves mostly few operations. To our knowledge, no previous study analyzes the pipeline length.

Finding 11: *Stream pipelines are typically composed of few operations.*

TABLE VII: Most popular collectors.

Collector	Occurrences	%
N/A	422 695 267	68.17
`Collectors.toList`	146 091 282	23.56
`Collectors.toSet`	15 855 229	2.56
`Collectors.toCollection`	15 550 665	2.51
`Collectors.toMap`	6 819 658	1.10
`Collectors.joining`	5 830 648	0.94
`Collectors.collectingAndThen`	5 495 747	0.88
`Collectors.groupingBy`	987 487	0.15
`Collectors.counting`	553 012	0.08
`Collectors.reducing`	121 687	0.02

Collectors. As shown in Table V, we detected a total of 197 338 242 invocations to `collect`, used to perform mutable reductions. Table VII shows the ten most popular collectors used during a mutable reduction (if any). Note that we identify the collector with the method used to obtain it (from class `Collector`). Among the executed streams, 68.17% do not perform a reduction using a collector (shown in Table VII as N/A). Our outcome is consistent with the results reported by Khatchadourian et al. and Nostas et al. in finding that reductions via `collect` mostly produce lists. Also, we confirm the observation done by both studies that concurrent reductions (e.g., `groupingByConcurrent`) are rarely used (less than 0.001%). This finding shows that despite the Stream API offers a variety of collectors, streams are mostly used to perform simple non-concurrent reductions whose results are mainly collected in lists and sets.

Finding 12: *Collectors that collect elements into lists and sets are the most used.*

Stream result types. We detected a total of 97 different stream result types. Table VIII summarizes the 10 most popular stream result types found. We find that `ArrayList` is the most popular data structure used to store the result of a stream (23.96%) while 23.53% of the executed streams do not return any result (e.g., streams performing the `forEach` and `forEachOrdered` terminal operations, which are `void` methods). This is expected, given the popularity of reductions collecting the results in lists as well as of `forEach`-like terminal operations. We are not aware of any related work characterizing stream result types.

Finding 13: *Lists are the most popular data structure used to store the output of a stream.*

Summary and discussion. Our findings can be summarized as follows. Streams are mostly used for basic data processing, mainly for the manipulation of lists containing few elements, using pipelines composed of few intermediate operations (mainly stateless ones), and typically performing simple data reductions whose output is also commonly stored in lists. We also find that parallel streams and concurrent data reductions are rarely used, that streams are often created from mutable data sources that do not support concurrent modification, and have ordering constraints. Finally, we find that map-reduce-style processing is popular via pipelines in the form of map-

TABLE VIII: Most popular stream result types.

Stream Result Type	Occurrences	%
`java.util.ArrayList`	148 567 911	23.96
`void`	145 918 994	23.53
`boolean`	112 159 247	18.09
`java.util.Optional`	99 032 395	15.97
`int`	29 961 760	4.83
`java.util.HashSet`	16 333 376	2.63
`java.util.HashMap`	15 464 414	2.49
`int[]`	9 576 981	1.54
`java.lang.String`	9 506 464	1.53
`Object[]`	8 745 210	1.41

filter-collect patterns. However, streams are also used for simple iterative-style collection processing via the execution of pipelines relying on `forEach`-like operations.

Our findings advance the understanding of stream processing on the JVM. First, our results can be used as a feedback to the community developing the Java class library to prioritize the optimization both of features of the Stream API according to their popularity and of related supporting features belonging to other Java APIs. Second, our results can be used by IDE and tool builders to understand which kind of support is required to help users make better decisions while using streams. In particular, our study highlights that parallel streams are not widely used. As a result, tools guiding an efficient parallelization of streams may help developers potentially improve the performance of their Java applications, particularly of those processing large datasets. Third, our findings can be used by developers in the search for missed opportunities to enhance data processing in Java. Indeed, our results show that the projects analyzed rarely make use of complex stream processing that truly benefits from the richness, versatility, and fluency of the Stream API. For instance, the use of parallel streams, the removal of ordering constraints (e.g., via `unordered`), as well as the use of concurrent collectors are not popular, but a careful selection and use of such features can make a difference in the goal of leveraging parallel data processing in Java. Finally, our findings can help educators training Java developers identify unexploited features of the Stream API that can be emphasized in learning processes, such that practitioners are aware of means to enable more efficient data processing on the JVM.

V. Limitations

In this section, we discuss the main limitations of our work and outline our future work.

Like any large-scale study, our findings depend on the analyzed projects, which may not be representative of the general use of Java streams. Nevertheless, the analyzed projects are diverse in aspects such as domain, size, and popularity.

Our study targets only GitHub. Nonetheless, it is currently the largest source-code-hosting facility, having more than 73 millions of users and hosting more than 200 million software projects [9]. We plan to expand the analysis to target other software repositories, such as GitLab [29] or BitBucket [30].

Our study uses Java version 8 (the version in which the Stream API was introduced). Projects relying on newer versions of Java are not included in our analysis and we do not consider the study of features of the Stream API introduced since Java 9. We plan to expand our study using the latest long long-term support release of Java.

Our study only considers projects that can be build via maven. We plan to use other build systems such as Ant [31] or Gradle [32] to target additional projects in further studies.

The timeframe selected for our study limits the number of projects that were analyzed. On the other hand, the chosen timeframe enables us to analyze current practices in the use of streams (excluding projects that are old, obsolete, or not maintained and which could bias the study), while still considering a large number of projects. As future work, we consider an extension of our analysis to cover more recently updated projects.

The analysis timeout may prevent some long-running projects to be analyzed. However, the choice of 1 hour as the analysis timeout is justified by the need for preventing non-verified software to uncontrollably consume our computing resources.

In comparison to previous studies [5]–[7] that mainly use manual code inspection or static analysis to analyze the code in a project, an important limitation of our study is that it only targets source code exercised by unit tests. Such code may not be representative for a real usage scenario of an application in production. However, previous work [8], [21] has shown that massively applying DPAs on workloads exercised by unit tests can provide useful information, highlight patterns, and derive statistics. We note that we execute testing code because we aim at running DPAs on a multitude of software projects automatically. As JUnit tests can be automatically executed by the build system via simple commands (e.g., `mvn test`), they make large-scale dynamic analysis possible.

Finally, our study considers streams that may be executed within test-harness classes. We plan to detect such kind of streams to avoid targeting stream executions related to warm-up phases. We also consider as future work the separate analysis of streams executed in application code, testing code, and in the Java class library.

VI. Related work

In this section, we review work related to our approach. First, we compare our work to other studies focused on the use of streams. Next, we review studies targeting functional programing in Java. Finally, we review work addressing the optimization of streams.

A. Studies on the Use of Java Streams

To the best of our knowledge, there are three studies examining the use of Java streams, all of which rely on static analysis and manual code inspection.

Tanaka et al. [5] mine 100 software projects to study the use of lambdas, streams, and the `Optional` class. They report that developers using such idioms mainly aim at improving performance and producing short, clear, and readable code. Regarding streams, they find that the most popular operations are `map`, `filter`, and `collect`. Khatchadourian et al. [6]

examine 34 projects to specifically study the use of streams in Java. The authors report several findings, including that parallel streams are not popular, that pipelines often have ordering constraints, and that streams are mostly used to iterate over collections and to perform data reductions. Finally, Nostas et al. [7] study the use of streams in 610 projects. Their work is a partial replication of the study of Khatchadourian et al. by considering a larger number of projects. The authors mainly confirm the results obtained by Khatchadourian et al. and find that streams are mostly used in frameworks, libraries, and tools.

In comparison to the aforementioned related work, our study is conducted at a much larger scale thanks to the use of a fully automated approach that avoids manual intervention. Moreover, we characterize runtime information that related work overlooks, since is mainly based on static analysis techniques that cannot collect dynamic metrics. In addition, our study is the first to report the popularity of source methods, source collection types, stream types, pipeline lengths, and stream result types. Lastly, as far as we are aware, we are the first ones to analyze the characteristics of stream data sources as collected at runtime, including their size, ordering constraints, duplicate allowance, mutability, and support for concurrent modification.

B. Studies on Functional Programming in Java

Some authors have studied functional programming in Java, in particular the use of lambdas.

Tsantalis et al. [33] study the use of lambdas to refactor duplicated code to benefit from *behavior parameterization* [34]. They find that lambdas are highly effective to avoid duplicated code. The study of Tsantalis et al. focuses solely on lambdas, disregarding streams. Mazinanian et al. [35] mine 241 software repositories containing over 100 000 lambdas, and survey 97 Java developers to understand how they are using lambdas. They find that developers are increasingly using lambdas to replace anonymous classes and for behavior parameterization. However, the study of Mazinanian et al. does not focus on streams. Nielebock et al. [36] study the use of lambdas in 2 923 projects implemented in C#, C++, and Java. They locate several lambdas in both application and testing code. Similarly to our finding showing that parallel streams are not popular, the authors find that developers tend to avoid using lambdas within concurrent code. Also the study of Nielebock et al. does not consider streams.

C. Optimization of Java Streams

Some authors have addressed the optimization of streams.

Ishizaki et al. [37] modify the IBM J9 JVM [38] and the Testarossa [39] compiler to translate invocations to the `IntStream.forEach` terminal operation of a parallel stream into optimized GPU code. Hayashi et al. [40] extend such work, proposing a supervised machine-learning approach generating heuristics that the runtime can use to select between a CPU or a GPU to efficiently execute parallel streams invoking `IntStream.forEach`. Khatchadourian et al. [41], [42] introduce an Eclipse plugin helping developers better code streams. The plugin evaluates whether it would be safe and potentially advantageous to restructure a stream pipeline to improve performance. To this end, the plugin infers how the execution of a stream would take place by analyzing it mainly via static analysis techniques [43], [44]. Finally, Møller et al. [45] present *StreamLiner*, a tool that translates the bytecode generated upon the compilation of sequential streams into bytecode representing more efficient imperative-style code.

While supporting the optimization of streams, the aforementioned tools are not designed for characterizing stream processing in the wild.

VII. Conclusions

In this paper, we present the first large-scale empirical study on the use of Java streams. In addition to the large scale, our study is the first one to consider dynamic characteristics of stream processing. We target 1 808 415 open-source software projects publicly available on GitHub, among which we find 132 211 projects that use the Stream API. Our study considers a total of 620 033 509 streams.

To conduct our study, we develop Stream-Analyzer, a novel dynamic analysis for collecting runtime information and key metrics that enable the fine-grained characterization of sequential and parallel stream processing on the JVM. We massively apply Stream-Analyzer using a fully automated approach, relying on a distributed infrastructure that enables mining applications hosted on GitHub.

The results of our study confirm the findings of recent efforts focused on studying the use of Java streams [5]–[7] at a much larger scale, and complementarily provide new insights about the use of the Stream API thanks to the collection of dynamic metrics unique to streams.

As part of our future work, we aim at extending our analysis by considering projects hosted in large-scale repositories other than GitHub, a larger timeframe, a newer Java version, and other build systems, as discussed in Sec. V. We also plan to identify projects showing suboptimal stream processing and find ways to optimize them. Finally, we plan to release Stream-Analyzer as open-source software to facilitate the replicability of our results.

Acknowledgments

This work has been supported by the Swiss National Science Foundation (project 200020_188688).

References

[1] Oracle, "Package java.util.stream," https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/Stream.html, 2021.

[2] Bird, Richard and Wadler, Philip, *An Introduction to Functional Programming*, 1st ed. GBR: Prentice Hall International (UK) Ltd., 1988.

[3] Bloch, Joshua, *Effective Java*, 3rd ed. Addison-Wesley, 2018.

[4] Lea, Doug (with the help of Goetz, Brian and Sandoz, Paul and Shipilev, Aleksey and Kabutz, Heinz and Bowbee, Joe), "When to Use Parallel Streams," http://gee.cs.oswego.edu/dl/html/StreamParallelGuidance.html, 2014.

[5] Hiroto, Tanaka and Shinsuke, Matsumoto and Shinji, Kusumoto, "A Study on the Current Status of Functional Idioms in Java," *IEICE Transactions on Information and Systems*, vol. E102.D, no. 12, pp. 2414–2422, 2019.

[6] Khatchadourian, Raffi and Tang, Yiming and Bagherzadeh, Mehdi and Ray, Baishakhi, "An Empirical Study on the Use and Misuse of Java 8 Streams," in *FASE*, 2020, pp. 97–118.

[7] Nostas, Joshua and Alcocer, Juan Pablo Sandoval and Costa, Diego Elias and Bergel, Alexandre", "How Do Developers Use the Java Stream API?" in *ICCSA*, 2021, pp. 323–335.

[8] Villazón, Alex and Sun, Haiyang and Rosà, Andrea and Rosales, Eduardo and Bonetta, Daniele and Defilippis, Isabella and Oporto, Sergio and Binder, Walter, "Automated Large-Scale Multi-Language Dynamic Program Analysis in the Wild," in *ECOOP*, vol. 134, 2019, pp. 20:1–20:27.

[9] GitHub, Inc., "GitHub," https://github.com/about, 2022.

[10] Oracle, "Interface Stream<V>," https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/Stream.html, 2021.

[11] Lea, Doug, "A Java Fork/Join Framework," in *JAVA*, 2000, pp. 36–43.

[12] Oracle, "Class ForkJoinPool," https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/ForkJoinPool.html, 2021.

[13] ——, "Class Collectors," https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/Collectors.html, 2021.

[14] ——, "Interface BaseStream<T,S extends BaseStream<T,S>>," https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/BaseStream.html, 2021.

[15] Marek, Lukáš and Villazón, Alex and Zheng, Yudi and Ansaloni, Danilo and Binder, Walter and Qi, Zhengwei, "DiSL: A Domain-specific Language for Bytecode Instrumentation," in *AOSD*, 2012, pp. 239–250.

[16] Binder, Walter and Hulaas, Jarle and Moret, Philippe, "Advanced Java Bytecode Instrumentation," in *PPPJ*, 2007, pp. 135–144.

[17] Oracle, "The Java Virtual Machine Specification - Chapter 5. Loading, Linking, and Initializing," https://docs.oracle.com/javase/specs/jvms/se17/html/jvms-5.html, 2021.

[18] Marek, Lukáš and Kell, Stephen and Zheng, Yudi and Bulej, Lubomír and Binder, Walter and Tůma, Petr and Ansaloni, Danilo and Sarimbekov, Aibek and Sewe, Andreas, "ShadowVM: Robust and Comprehensive Dynamic Program Analysis for the Java Platform," in *GPCE*, 2013, pp. 105–114.

[19] Kell, Stephen and Ansaloni, Danilo and Binder, Walter and Marek, Lukáš, "The JVM is Not Observable Enough (and What to Do About It)," in *VMIL*, 2012, pp. 33–38.

[20] The JUnit Team, "JUnit," https://junit.org, 2022.

[21] Yudi Zheng and Andrea Rosà and Luca Salucci and Yao Li and Haiyang Sun and Omar Javed and Lubomir Bulej and Lydia Y. Chen and Zhengwei Qi and Walter Binder, "AutoBench: Finding Workloads That You Need Using Pluggable Hybrid Analyses," in *SANER*, 2016, pp. 639–643.

[22] The Apache Software Foundation, "Apache Maven Project," https://maven.apache.org, 2022.

[23] Oracle, "Interface IntStream<T>," https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/stream/IntStream.html, 2021.

[24] ——, "Class Arrays," https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Arrays.html, 2021.

[25] ——, "Class Random," https://docs.oracle.com/javase/8/docs/api/java/util/Random.html, 2021.

[26] Costa, Diego and Andrzejak, Artur and Seboek, Janos and Lo, David, "Empirical Study of Usage and Performance of Java Collections," in *ICPE*, 2017, p. 389–400.

[27] Oracle, "Interface Spliterator<T>," https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Spliterator.html, 2021.

[28] Dean, Jeffrey and Ghemawat, Sanjay, "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, no. 1, p. 107–113, 2008.

[29] GitLab, "GitLab," https://www.gitlab.com, 2022.

[30] Atlassian, "BitBucket," https://bitbucket.org, 2022.

[31] The Apache Software Foundation, "The Apache Ant Project," https://ant.apache.org, 2022.

[32] Gradle Inc., "Gradle Build Tool," https://gradle.org, 2022.

[33] Nikolaos, Tsantalis and Davood, Mazinanian and Shahriar Rostami, "Clone Refactoring with Lambda Expressions," in *ICSE*, 2017, pp. 60–70.

[34] Urma, Raoul-Gabriel and Fusco, Mario and Mycroft, Alan, *Java 8 in Action: Lambdas, Streams, and Functional-Style Programming*, 1st ed. Manning Publications Co., 2014.

[35] Mazinanian, Davood and Ketkar, Ameya and Tsantalis, Nikolaos and Dig, Danny, "Understanding the Use of Lambda Expressions in Java," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 1–31, Oct. 2017.

[36] Nielebock, Sebastian and Heumüller, Robert and Ortmeier, Frank, "Programmers Do Not Favor Lambda Expressions for Concurrent Object-Oriented Code," *Empirical Softw. Eng.*, vol. 24, no. 1, pp. 103–138, Feb. 2019.

[37] K. Ishizaki and A. Hayashi and G. Koblents and V. Sarkar, "Compiling and Optimizing Java 8 Programs for GPU Execution," in *PACT*, 2015, pp. 419–431.

[38] Renouf, Colin, *The IBM J9 Java Virtual Machine for Java 6*. Apress, 2009, pp. 15–34.

[39] Grcevski, Nikola and Kielstra, Allan and Stoodley, Kevin and Stoodley, Mark and Sundaresan, Vijay, "JavaTM Just-in-time Compiler and Virtual Machine Improvements for Server and Middleware Applications," in *VM*, 2004, pp. 12–12.

[40] Hayashi, Akihiro and Ishizaki, Kazuaki and Koblents, Gita and Sarkar, Vivek, "Machine-Learning-based Performance Heuristics for Runtime CPU/GPU Selection," in *PPPJ*, 2015, pp. 27–36.

[41] Khatchadourian, Raffi and Tang, Yiming and Bagherzadeh, Mehdi and Ahmed, Syed, "A Tool for Optimizing Java 8 Stream Software via Automated Refactoring," in *SCAM*, 2018, pp. 34–39.

[42] ——, "Safe Automated Refactoring for Intelligent Parallelization of Java 8 Streams," in *ICSE*, 2019, pp. 619–630.

[43] Fink, Stephen J. and Yahav, Eran and Dor, Nurit and Ramalingam, G. and Geay, Emmanuel, "Effective Typestate Verification in the Presence of Aliasing," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 2, May 2008.

[44] WALA Team, "WALA," http://wala.sourceforge.net, 2019.

[45] Møller, Anders and Veileborg, Oskar Haarklou, "Eliminating Abstraction Overhead of Java Stream Pipelines Using Ahead-of-Time Program Optimization," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020.