A Tool for Optimizing Java 8 Stream Software via Automated Refactoring

Raffi Khatchadourian
City University of New York (CUNY) Hunter College
raffi.khatchadourian@hunter.cuny.edu

Yiming Tang
City University of New York (CUNY) Graduate Center
ytang3@gradcenter.cuny.edu

Mehdi Bagherzadeh
Oakland University
mbagherzadeh@oakland.edu

Syed Ahmed
Oakland University
sfahmed@oakland.edu

***Abstract*—Streaming APIs are pervasive in mainstream Object-Oriented languages and platforms. For example, the Java 8 Stream API allows for functional-like, MapReduce-style operations in processing both finite, e.g., collections, and infinite data structures. However, using this API efficiently involves subtle considerations like determining when it is best for stream operations to run in parallel, when running operations in parallel can actually be less efficient, and when it is safe to run in parallel due to possible lambda expression side-effects. In this paper, we describe the engineering aspects of an open source automated refactoring tool called OPTIMIZE STREAMS that assists developers in writing optimal stream software in a semantics-preserving fashion. Based on a novel ordering and typestate analysis, the tool is implemented as a plug-in to the popular Eclipse IDE, using both the WALA and SAFE frameworks. The tool was evaluated on 11 Java projects consisting of ∼642 thousand lines of code, where we found that 36.31% of candidate streams we refactorable, and an average speedup of 1.55 on a performance suite was observed. We also describe experiences gained from integrating three very different static analysis frameworks to provide developers with an easy-to-use interface for optimizing their stream code to its full potential.**

***Index Terms*—refactoring, automatic parallelization, typestate analysis, ordering, Java 8, streams, eclipse, WALA, SAFE**

I. INTRODUCTION

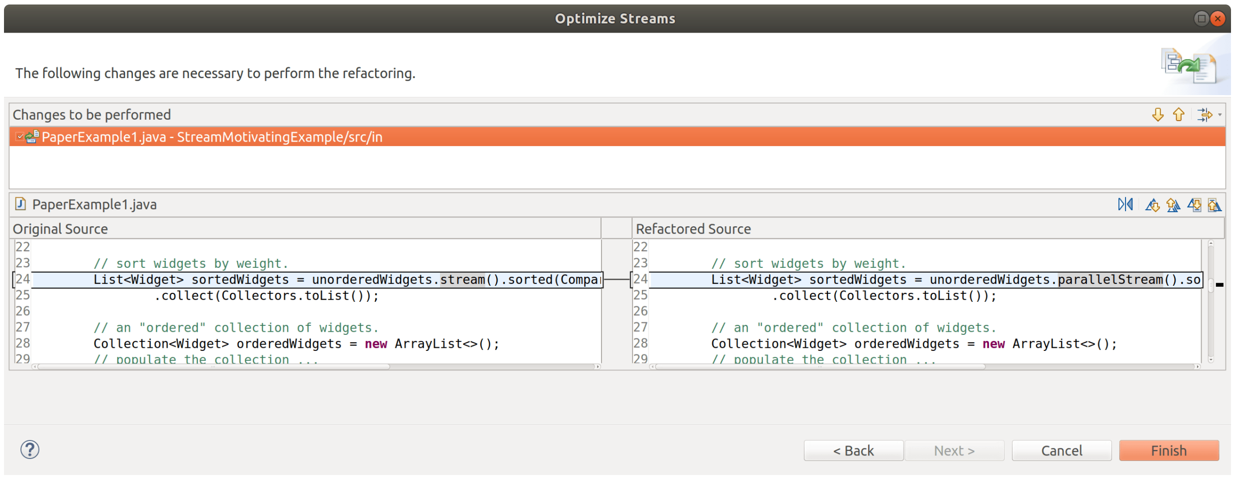
Streaming APIs are widely-available in today's mainstream, Object-Oriented programming languages and platforms [1], including Scala [2], JavaScript [3], C# [4], Java [5], and Android [6]. They incorporate MapReduce-like [7] operations on native data structures like collections. MapReduce abstracts away much of the complexity of writing parallel programs by facilitating big data processing on multiple nodes using succinct functional-like programming constructs. It is a popular programming paradigm for writing a certain class of parallel programs, making writing parallel code in these languages easier. Particularly, such streaming APIs can make writing parallel programs less error-prone by allowing developers to avoid possible data races, thread interference and/or contention, and other problems commonly associated with parallel programs [8]. For example, as illustrated in fig. 1, Java 8 streams can execute in parallel simply by adding `parallel()` call to the operation pipeline.

However, MapReduce traditionally runs in a highly-distributed environment in the absence of shared memory. On the other hand, Java 8 stream, for example, typically execute on a single node under multiple threads or cores in a shared memory space. In this case, because collections reside on the local machine's memory, issues may arise from the close ties between shared memory and the operations. Thus, developers must *manually* determine whether running stream code in parallel results in an efficient yet interference-free program [9], ensuring that no operations on different threads interleave [10].

Though there are many benefits [11, Ch. 1] to using streams, efficient stream computation necessitates some careful thought, such as determining whether executing streams in parallel is more optimal than running it sequentially due to potential side-effects, buffering, etc. Passing *stateful* expressions to stream operations may also be problematic as the results of such expressions may depend on state that may change, which can undermine performance. These problems may not be immediately evident to developers, possibly requiring complex interprocedural analysis, understanding the particulars of stream implementations, and knowing which API to use in the best situations. Manual analysis and/or refactoring, i.e., semantics-preserving, source-to-source transformation, for optimal stream code can be overwhelming and error- (developers may mistakenly alter or misuse code), and omission-prone (developers may miss refactoring opportunities).

In fact, during ongoing experiments based on our previous work [9], we found 157 total streams across 11 open source subject projects with a 34 subject maximum, which can increase over time with a rise in stream popularity. Also, the number of operations issued per stream may be many; we found that there were 4.14 operations per stream on average. This warrants manual determination and compacting of operation insertion locations when manually optimizing streams. Lastly, (manual) interprocedural and type hierarchy analysis may be needed to discover ways to use streams in a particular context. Permutating through operation combinations and subsequently assessing performance, for which dedicated performance tests may be absent, can be burdensome.

In this paper, we report on the design and implementation of a fully-automated refactoring tool named OPTIMIZE STREAMS that transforms Java 8 stream code for improved performance.



The following changes are necessary to perform the refactoring.

PaperExample1.java - StreamMotivatingExample/src/in

PaperExample1.java

Original Source

```
        // sort widgets by weight.
        List<Widget> sortedWidgets = unorderedWidgets.stream().sorted(Compa
                .collect(Collectors.toList());

        // an "ordered" collection of widgets.
        Collection<Widget> orderedWidgets = new ArrayList<>();
        // populate the collection ...
```

Refactored Source

```
        // sort widgets by weight.
        List<Widget> sortedWidgets = unorderedWidgets.parallelStream().so
                .collect(Collectors.toList());

        // an "ordered" collection of widgets.
        Collection<Widget> orderedWidgets = new ArrayList<>();
        // populate the collection ...
```

Fig. 1: Screenshot of the OPTIMIZE JAVA 8 STREAM REFACTORING preview wizard.

The tool is used in assessing our ongoing work [9] but is also publicly available an open source Eclipse (http://eclipse.org) plug-in (available at http://git.io/vpTLk) built atop of the Java Development Tools (JDT) (http://eclip.se/ed) refactoring infrastructure [12] with a fully-functional UI, preview pane, and unit tests. The approach at the tool's foundation is based on a novel ordering analysis, which infers when maintaining the order of a data sequence in a particular expression is necessary for semantics preservation, and typestate analysis [13], [14], which augments the type system with "state" and has been traditionally used for preventing resource usage errors (e.g., trying to read from a closed file, not closing a socket prior to program termination). Our tool uses typestate, along with interprocedurally analyzing relationships between types, to identify stream usages that can possibly execute more efficiently in parallel and which, in fact, can be *hindered* by parallelism. It also discovers possible side-effects in λ-expressions, i.e., units of computation to be executed in a deferred fashion, to safely transform streams to either execute sequentially or in parallel.

To the best of our knowledge, OPTIMIZE STREAMS is the first tool to integrate automated refactoring with typestate analysis. It uses both the WALA static analysis framework (http://wala.sf.net) and the SAFE typestate analysis engine (http://git.io/vxwBs). Integrating such complex static analyses is an engineering challenge as these analyses normally involve an instruction-based Intermediate Representation (IR), while refactorings work on Abstract Syntax Trees (ASTs) to facilitate source-to-source transformation. It is convenient for such analyses to operate on instruction-based IR as they can be encoded in a way that simplifies the analysis (e.g., Static Single Assignment; SSA [15]). However, since refactorings involve source-to-source transformations, it is convenient to work directly on the AST. Relating complex static analysis results to the original IR is an engineering challenge of this work, and we discuss our experiences in integrating typestate into our refactoring tool.

The ongoing evaluation currently involves studying our plug-in's performance on 11 Java projects of varying size and domain with a total of ~642 thousand lines of code. In this paper, we discuss the engineering challenges faced in the study, as well as those faced in compiling our data set [16].

We make the following specific contributions:

Implementation and motivation details. Our tool's novel engineering aspects are detailed with a focus on its integration of typestate analysis, instruction-based IR static analysis, and abstract syntax-based analysis. Also, architecture, API usage, data representations, algorithms, implementation issues, and a more comprehensive motivation are outlined.

Real-world study engineering. To ensure real-world applicability, our tool enabled the study of 11 Java programs that use streams, where we found that 36.31% of candidate streams we refactorable, with an observed average speedup of 1.55 during performance testing. Engineering challenges faced in this large-scale study, the experiences gained in developing this contribution, and user feedback is described.

II. MOTIVATION

In this section, use cases that have motivated the existence of our tool are portrayed. Using a simplified example, we highlight some of the challenges associated with the automated analysis and refactoring of Java 8 streams for greater parallelism and/or increased efficiency.

Listing 1 portrays code that uses the Java 8 Stream API to process collections of `Widgets` with `weights`. Listing 1a shows the original version, while Listing 1b is the improved (but semantically equivalent) version as a result of our refactoring tool. In listing 1a, a `Collection` of `Widgets` is declared (line 1) and instantiated (line 1) that does not maintain element ordering as `HashSet` does not support it [17]. Note that ordering is dependent on the run time type (`HashSet`) rather than the compile-time type (`Collection`).

A `stream`, i.e., a data source view representing an element sequence supporting MapReduce-style operations, of `unorderedWidgets` is created on line 4 via the `stream()` method as invoked on the collection. It is a

Listing 1 Snippet of `Widget` collection processing using Java 8 streams based on [5], [9].

(a) Stream code snippet prior to refactoring.

```
1  Collection<Widget> unorderedWidgets = new HashSet<>();
2
3  List<Widget> sortedWidgets = unorderedWidgets
4    .stream()
5    .sorted(Comparator.comparing(Widget::getWeight))
6    .collect(Collectors.toList());
7
8  Collection<Widget> orderedWidgets = new ArrayList<>();
9
10 // collect distinct widget weights into a TreeSet.
11 Set<Double> distinctWeightSet = orderedWidgets
12   .stream().parallel()
13   .map(Widget::getWeight).distinct()
14   .collect(Collectors.toCollection(TreeSet::new));
15
16 // collect distinct widget colors into a HashSet.
17 Set<Color> distinctColorSet = orderedWidgets
18   .parallelStream().map(Widget::getColor)
19   .distinct()
20   .collect(HashSet::new, Set::add, Set::addAll);
21
22 // collect widget colors matching a regex.
23 Pattern pattern = Pattern.compile(".*e[a-z]");
24 ArrayList<String> results = new ArrayList<>();
25 orderedWidgets.stream().map(w -> w.getColor())
26   .map(c -> c.toString())
27   .filter(s -> pattern.matcher(s).matches())
28   .forEach(s -> results.add(s));
```

(b) Improved stream code via refactoring.

```
1  Collection<Widget> unorderedWidgets = new HashSet<>();
2
3  List<Widget> sortedWidgets = unorderedWidgets
4    .~~stream()~~parallelStream()
5    .sorted(Comparator.comparing(Widget::getWeight))
6    .collect(Collectors.toList());
7
8  Collection<Widget> orderedWidgets = new ArrayList<>();
9
10 // collect distinct widget weights into a TreeSet.
11 Set<Double> distinctWeightSet = orderedWidgets
12   .stream().~~parallel()~~
13   .map(Widget::getWeight).distinct()
14   .collect(Collectors.toCollection(TreeSet::new));
15
16 // collect distinct widget colors into a HashSet.
17 Set<Color> distinctColorSet = orderedWidgets
18   .parallelStream().map(Widget::getColor)
19   .unordered().distinct()
20   .collect(HashSet::new, Set::add, Set::addAll);
21
22 // collect widget colors matching a regex.
23 Pattern pattern = Pattern.compile(".*e[a-z]");
24 ArrayList<String> results = new ArrayList<>();
25 orderedWidgets.stream().map(w -> w.getColor())
26   .map(c -> c.toString())
27   .filter(s -> pattern.matcher(s).matches())
28   .forEach(s -> results.add(s));
```

sequential stream, meaning the operations will execute serially due to the particular API called. Streams may also be associated with an *encounter order*, i.e., is the order the elements will be visited by the operations. The encounter order is derived from the steam *ordered* attribute, which can be dependent on whether the stream's source supports ordering of its elements. For example, the stream on line 4 will be unordered since it's source (line 1) `HashSets` are unordered. As such, the order in which may stream operations traverse the elements is nondeterministic, a characteristic that can have significant impact on efficient parallel computation.

On line 5, elements of the stream are `sorted()` by the corresponding *intermediate* operation, the result of which is a (possibly) new stream with the encounter order rearranged accordingly. The operation has an optional parameter, namely, a `Comparator`, dictating the sorting criteria. In this case, `Widgets` are to be sorted by their weight in non-decreasing order. The syntax `Widget::getWeight` is a method *reference* denoting the method that should be used for the comparison. Intermediate operations like `sorted()` have their execution deferred, i.e., they are "lazily" executed, are deferred until a so-called *terminal* operation is executed like `collect()` (line 6). This is a special kind of (mutable) reduction, aggregating results of prior intermediate operations into a given `Collector`, in this case, one that yields a `List`. The combination of the stream data source, any (queued) intermediate operations, and a terminal operation such as `collect()` form a stream *pipeline*. This execution of this pipeline results in a `List` of `Widgets` sorted by `weight`.

It may be possible to increase performance by running this stream's pipeline in parallel. Listing 1b, line 4 displays the corresponding refactoring with the stream pipeline execution in parallel (removed code is ~~struck through~~, while the added code is underlined). Note, however, that had the stream been *ordered*, running the pipeline in parallel may actually result in worse performance due to the multiple passes and/or data buffering required by so-called *stateful* intermediate operations (SIOs) like `sorted()`. Because the stream is *unordered*, the mutable reduction can be done more efficiently [5].

A distinct widget weight `Set` is created on lines 11–14. Unlike the previous example, this reduction *already* takes place in `parallel` due to the corresponding call at line 12. Note though that there is a possible performance degradation here as the SIO `distinct` may require multiple passes, the computation takes place in *parallel*, *and* the stream it operates on is *ordered* as dictated by its source (i.e., `orderedWidgets` is an instance of an `ArrayList`). Keeping the parallel computation but unordering the stream may improve performance, but we would need to determine whether doing so is safe. In other words, we would need to know whether it is safe to unorder the stream prior to invoking the `distinct()` operation. To determine this automatically without developer input can be difficult, however. Furthermore, it can be error-prone if done manually, especially on large and complex projects.

Our insight includes that by analyzing the type of the resulting reduction, we may be able to determine if unordering a stream is safe. In this case, it is a (mutable) reduction (i.e., `collect()` operation on line 14) to a `Set`, of which subclasses that do not preserve ordering exist. If we could determine that the resulting `Set` is one of the unordered `Sets`, unordering the stream would be safe since such an operation would not preserve ordering. The type of the resulting `Set` returned by `collect()`, though, is determined by the passed `Collector`, in this case, the return value of `Collectors.toCollection(TreeSet::new)`.[1] Unfortunately, since `TreeSets` preserve ordering, we must keep the stream ordered. Here, to improve performance, it may be advantageous to run this pipeline, perhaps surprisingly, *sequentially*, the transformation of which takes place on line 12 of listing 1b. Note that removing `parallel()` is not the only option; it can also be replaced with `sequential()`, but, doing so would be redundant since `stream()` returns a stream that is *already* sequential.

In contrast, lines 17–20 map, in parallel, each `Widget` to its `Color`, filter those that are `distinct`, and `collect` them into a `Set`. To portray a variety of ways mutable reductions can occur, a more direct form of `collect()` is used rather than a `Collector`, and the collection is to a `HashSet`, which does *not* maintain element ordering. As such, and unlike the previous example, though the stream is originally ordered, since the (mutable) reduction is to an *unordered* destination, we can infer that the stream can be safely unordered to improve performance. Thus, line 19 in listing 1b shows the inserted call to `unordered()` immediately prior to the `distinct()` operation call. This allows `distinct()` to work more efficiently under parallel computation [5].

Lastly, on lines 23–28, `Widget` colors matching a regular expression are sequentially collected into an `ArrayList`. The code proceeds by `map`ping each widget to its `Color`, each `Color` to its `String` representation, filtering matching strings, and `forEach`, `add`ing them to the resulting `ArrayList` via the behavioral parameter (λ-expression) `s->results.add(s)`. The stream is not refactored to parallel because of the side-effects produced by the λ-expression. If executed in parallel, the unsynchronized `ArrayList` could cause incorrect results due to thread scheduling, altering original program semantics. Adding synchronization to the `ArrayList` would solve that problem but cause thread contention, undermining the benefit of parallelism [5].[2]

While the above example has been simplified, manual analysis of stream code can be complex, especially in large programs, necessitating a thorough understanding of API intricacies as seen in listing 1, possible alias analysis, knowledge of type ordering attributes, etc. Henceforth, it would be extremely valuable to developers if automation is available to assist them in writing their stream code. In the following sections, we discuss detail the engineering of our tool to automatically assist developers to refactor their streaming code to take full advantage of comprehensive streaming APIs.

III. Implementation

The Optimize Streams refactoring tool, available at http://git.io/vpTLk, is implemented as an open-source Eclipse IDE plug-in and built upon WALA and SAFE. Eclipse is leveraged for its existing, well-documented, and well-integrated refactoring framework and test engine [12], including static analysis and transformation APIs (e.g., `ASTRewrite`), refactoring preview pane (as shown in fig. 1), precondition checking (e.g., `Refactoring.checkInitialConditions()`, `Refactoring.checkFinalPreconditions()`), and refactoring testing (e.g., `RefactoringTest`). It serves as a front-end to our refactoring, and due to plug-ins such as m2e (http://eclip.se/eb) and Buildship (http://eclip.se/3T), it may be utilized by any project that takes advantage of popular build systems like Maven and Gradle. What is more is that Eclipse is completely open source for all Java development (see http://jetbrains.com/idea/#chooseYourEdition) thus possibly impacting more Java developers. For the initial entry point into the tool, as well as the transformation portion, Eclipse ASTs with source symbol bindings are used as an IR.

A. Architecture and Dependencies

WALA is used for static analyses such as side-effect analysis (ModRef), and SAFE, which depends on WALA, for its typestate analysis, which is depicted in fig. 2. The right-hand side of fig. 2 portrays the internal architecture of Optimize Streams, while the left-hand side depicts its external dependencies. The internal architecture, for the most part, follows in line with that described in Khatchadourian and Masuhara [18]. It is listed here for self-containment, but further details are not included. However, the relationship between the internal plug-in architecture and that of the external dependencies are described more fully here.

1) Entry Points Selection: As shown in fig. 2, the core internal plug-ins consist of `edu.cuny.hunter.streamrefactoring.core` and `core.analysis`. The former is mainly responsible for dealing with the Eclipse ASTs, however, it does use WALA to process entry points and relate them to the call graph produced by WALA. Our tool accepts two kinds of entry points, explicit and implicit. Explicit entry points can be specified by the developer using annotations found in our accompanying annotation library. Developers may also elect, via a wizard option, to have our tool automatically discover different kinds of "standard" entry points, including main methods, JUnit test cases, and microbenchmarking methods (JMH). Our tool unions explicit and implicit entry points.

2) Static Analysis Integration: The `core.analysis` package is mainly responsible for bridging the Eclipse representation with that of the results of the static analysis employed by WALA and SAFE. This includes using the `com.ibm.wala.ide.util.JavaEclipseProjectPath` to properly initiate the analysis path used by WALA to perform the SSA transformation. Some changes were necessary to this class in order to support refactoring test suites, including dealing with artificial JDK classes (e.g., `rtstubs.jar`).

A call graph is built using WALA, which is needed for interprocedural type inference (using pointer analysis) for determining stream source types, the ModRef analysis for discovering possible λ-expression side-effects, and the typestate analysis, for determining stream state, e.g., parallel, unordered. Our tool uses a k-CFA call graph construction algorithm, as stream *client* code is the focus of the analysis. The k

[1] `TreeSet::new` is a method reference to the `TreeSet` default ctor.

[2] Fixing this problem could also involve refactoring `forEach()` to a mutable reduction but is currently outside the scope of our tool.

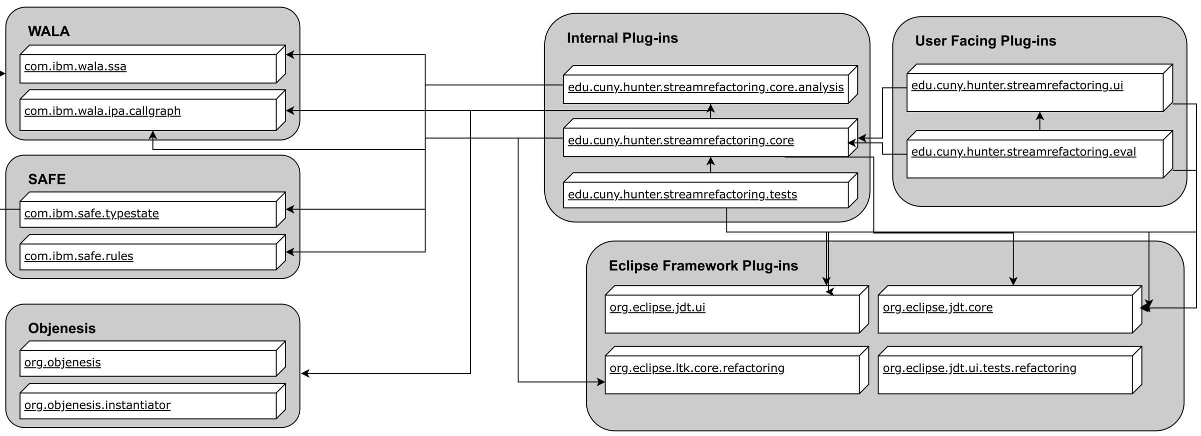

WALA
com.ibm.wala.ssa
com.ibm.wala.ipa.callgraph
SAFE
com.ibm.safe.typestate
com.ibm.safe.rules
Objenesis
org.objenesis
org.objenesis.instantiator
Internal Plug-ins
edu.cuny.hunter.streamrefactoring.core.analysis
edu.cuny.hunter.streamrefactoring.core
edu.cuny.hunter.streamrefactoring.tests
User Facing Plug-ins
edu.cuny.hunter.streamrefactoring.ui
edu.cuny.hunter.streamrefactoring.eval
Eclipse Framework Plug-ins
org.eclipse.jdt.ui
org.eclipse.jdt.core
org.eclipse.ltk.core.refactoring
org.eclipse.jdt.ui.tests.refactoring

Fig. 2: Architecture and dependency diagram.

parameter is input to our tool (with $k = 2$ being the default as it is the minimum k value to consider client-code) for methods returning streams and $k = 1$ elsewhere (for tractability). SAFE also utilizes the call graph, as depicted in fig. 2.

3) Stream Ordering Analysis: While WALA is used to approximate possible stream source types (e.g., types of collections for which streams derived), reflection is used to determine the type's "ordering" attribute. Doing so is possible as a type's ordering does not typically change throughout the lifetime of the associated object. Built-in reflection mechanisms are used to reflectively instantiate the type and retrieve its ordering characteristics by calling the `characteristics()` method on an associated stream's `Spliterator`. When types have no-arg constructors, Objenesis (http://objenesis.org), a tool normally used for Mock Objects, is used to bypass constructor calls.

B. Relating Intermediate Representations

As previously discussed, for the refactoring portion, we utilize Eclipse ASTs as an IR, while the WALA-based static analysis consumes instruction-based IR in SSA form. Our tool maps the different IRs when necessary, e.g., to identify transformation locations and to utilize generic information, which is only available at the source (AST) level due to type erasure, to improve the precision of the type analysis. To relate SSA-based IR to Eclipse ASTs, a combination of line number (retrieved via an option in WALA and available in Eclipse AST bindings) and method signatures is used.

C. Typestate Analysis Integration

Note that each intermediate operation may result in a new stream instance being created. SAFE tracks the state of instances using a unique identifier representing the approximated object instance at run time. This identifier is correlated with signatures in the call string. Call strings are available due to the k-CFA call graph construction algorithm. The call string entry is then related to a corresponding object creation instruction in the SSA, which in turn is mapped to the corresponding AST node in the manner mentioned in section III-B.

Typestate analysis is traditionally used to validate *complete* sequences of methods called on objects. Traditionally, this ensures that objects are not in a nonsensical state when particular methods are called on them and that no resources have been leaked (e.g., a missing call to `close()` on a file). In our case, however, we are interested in determining stream at the point of the reduction, i.e., when a terminal operation is called, which may not represent the end of the program. In our implementation, this required "dissecting" the internal details of the SAFE analysis engine and extracting state details at the appropriate times. This is mainly enabled by the `com.ibm.safe.Factoid` type, which relates individual object instances to state at a particular instruction.

SAFE was originally designed to be used by developers as end-users and not for programmatic consumption as is the case with OPTIMIZE STREAMS. In other words, SAFE requires developers to specify automata to be used in the typestate analysis. It is comprehensive with many options that are entered in text-based configuration files. As such, some engineering challenges involving utilizing SAFE in a programmatic fashion including building APIs to create automata were necessary.

IV. EVALUATION ENGINEERING CHALLENGES

While our study is currently in progress, details of which may be found on our project website (http://cuny.is/streams), here, we highlight some of the empirical findings of our tool, as well as discuss the engineering challenges faced in its assessment. OPTIMIZE STREAMS was applied to 11 Java programs of varying size and domain that use Java 8 streams. In these projects, our tool was able to refactor 36.31% of candidate streams it encountered despite its conservative nature. We proceeded to assess the impact of our tool by comparing the results of performance tests before and after the refactoring. Performance tests, in particular, microbenchmarks, are highly desirable for this kind of assessment as there are many factors that can externally influence the efficiency of parallel programs. Microbenchmarking tests, like those written on the Java Microbenchmarking Harness (JMH), offer various important features in this domain including process isolation, warm-up routines, etc. Moreover, to truly benefit from parallelism, programs must process an amount of data over a particular epsilon, i.e., the point in which the overhead of running in parallel meets the performance of serial execution. As such, true performance tests must typically process large data sets so that parallelism can be exhibited.

Despite the benefits of microbenchmarking, however, of the 11 projects studied, only one, `htm.java`, included a proper JMH test suite. Even though this open source project went above and beyond to include such a test suite, the amount of data being processed by the test suite was minimal. By direction of the project developers [19], we expanded the amount of data and were able to observe an average speedup of 1.55. This is particularly encouraging as it matches the speedup observed by a similar *manual* refactoring by Naftalin [20, Ch. 6]. We also received encouraging positive feedback from developers after submitting a patch including the refactoring as a pull request to the project. The authors were even invited to join the project as regular contributors, but we are still awaiting the results of the pull request.

In the absence of true performance tests, which seemed to be rare in the open source projects we explored, regular JUnit tests were used to obtain more performance metrics. Though these tests are not true indicators of performance due to the lack of process isolation, data set size, etc., we were at least able to establish that, under such conditions, our refactoring, on average, did not produce worse performance than the original. To help mitigate the absence of performance tests in this situation, we ran entire test suites up to 100 times and averaged the result, with the hope of eliminating some of the warmup factors. However, data set size in this scenario was not controlled and most likely small as unit tests normally execute upon each commit and need to be fast. The sheer number of unit tests in our subject projects was too large to individually increase data set sizes.

Another engineering challenge encountered was specifying entry points for a large project corpus, many of which were frameworks. This eventually lead to the automatic discovery of common entry points feature discussed in section III-A1.

V. CONCLUSION & FUTURE WORK

We have described the engineering aspects of an automated refactoring tool called OPTIMIZE STREAMS that assists developers with writing optimal Java 8 Stream code. It is open source and widely available to Java developers as an Eclipse plug-in. OPTIMIZE STREAMS integrates an Eclipse refactoring with the advanced static analyses offered by WALA and SAFE. 11 Java projects totaling $\sim$642 thousands of lines of code were used in the tools assessment, engineering challenges faced by the evaluation were discussed, and a speedup of 1.55 on the refactored code was observed. Several options for customizing the behavior of the tool are available to developers.

In the future, we plan to handle more advanced ways of relating ASTs to SSA-based IR, as well as incorporate more kinds of (complex) reductions like those involving maps. We also plan to explore applicability to other streaming APIs/languages and explore the possibility of refactoring side-effect producing code so that it is amenable to our refactoring.

REFERENCES

[1] A. Biboudis, N. Palladinos, G. Fourtounis, and Y. Smaragdakis, “Streams à la carte: Extensible pipelines with object algebras,” in *ECOOP*, 2015, pp. 591–613. DOI: 10.4230/LIPIcs.ECOOP.2015.591.

[2] EPFL. (2017). Collections–mutable and immutable collections, [Online]. Available: https://www.scala-lang.org/api/2.12.3/scala/collection.

[3] Refsnes Data. (2015). JavaScript array map() method, [Online]. Available: https://www.w3schools.com/jsref/jsref_map.asp.

[4] Microsoft. (2018). LINQ: .NET language integrated query, [Online]. Available: http://msdn.microsoft.com/en-us/library/bb308959.aspx.

[5] Oracle. (2017). java.util.stream, [Online]. Available: http://docs.oracle.com/javase/9/docs/api/java/util/stream/package-summary.html.

[6] J. Lau. (2017). Future of Java 8 language feature support on Android, [Online]. Available: http://android-developers.googleblog.com/2017/03/future-of-java-8-language-feature.html.

[7] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008. DOI: 10.1145/1327452.1327492.

[8] S. Lu, S. Park, E. Seo, and Y. Zhou, “Learning from mistakes: A comprehensive study on real world concurrency bug characteristics,” in *ASPLOS*, ACM, 2008, pp. 329–339. DOI: 10.1145/1346281.1346323.

[9] Y. Tang, R. Khatchadourian, M. Bagherzadeh, and S. Ahmed, “Poster: Towards safe refactoring for intelligent parallelization of java 8 streams,” in *ICSE Companion*, 2018. DOI: 10.1145/3183440.3195098.

[10] Oracle. (2017). Thread interference, [Online]. Available: http://docs.oracle.com/javase/tutorial/essential/concurrency/interfere.html.

[11] R. Warburton, *Java 8 Lambdas: Pragmatic Functional Programming*. O’Reilly Media, Apr. 7, 2014.

[12] D. Bäumer, E. Gamma, and A. Kiezun, “Integrating refactoring support into a java development tool,” Oct. 2001, [Online]. Available: http://people.csail.mit.edu/akiezun/companion.pdf.

[13] R. E. Strom and S. Yemini, “Typestate: A programming language concept for enhancing software reliability,” *IEEE TSE*, vol. SE-12, no. 1, pp. 157–171, Jan. 1986. DOI: 10.1109/tse.1986.6312929.

[14] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay, “Effective typestate verification in the presence of aliasing,” *ACM TOSEM*, vol. 17, no. 2, pp. 91–934, May 2008. DOI: 10.1145/1348250.1348255.

[15] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Global value numbers and redundant computations,” in *POPL*, ACM SIGPLAN-SIGACT, ACM, 1988, pp. 12–27. DOI: 10.1145/73560.73562.

[16] R. Khatchadourian, Y. Tang, M. Bagherzadeh, and S. Ahmed, *Raw results for the Optimize Java 8 Stream refactoring evaluation*, Apr. 2018. DOI: 10.5281/zenodo.1219882.

[17] Oracle. (2017). HashSet (Java SE 9) & JDK 9, [Online]. Available: http://docs.oracle.com/javase/9/docs/api/java/util/HashSet.html.

[18] R. Khatchadourian and H. Masuhara, “Defaultification refactoring: A tool for automatically converting Java methods to default,” in *ASE*, Oct. 2017, pp. 984–989. DOI: 10.1109/ASE.2017.8115716.

[19] D. Ray. (2018). Pull request 539, [Online]. Available: http://github.com/numenta/htm.java/pull/539#issuecomment-375146963.

[20] M. Naftalin, *Mastering Lambdas: Java Programming in a Multicore World*, 1st ed. McGraw-Hill, Sep. 11, 2014.