Summer 7-23-2019

Safe Automated Refactoring for Intelligent Parallelization of Java 8 Streams

Raffi T. Khatchadourian
CUNY Hunter College

Yiming Tang
CUNY Graduate Center

Mehdi Bagherzadeh
Oakland University

Syed Ahmed
Oakland University

Safe Automated Refactoring for Intelligent Parallelization of Java 8 Streams

Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Syed Ahmed

Abstract

Streaming APIs are becoming more pervasive in mainstream Object-Oriented programming languages and platforms. For example, the Stream API introduced in Java 8 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 such as determining when it is best for stream operations to run in parallel, when running operations in parallel can be less efficient, and when it is safe to run in parallel due to possible lambda expression side-effects. Also, streams may not run all operations in parallel depending on particular collectors used in reductions. In this paper, we present an automated refactoring approach that assists developers in writing efficient stream code in a semantics-preserving fashion. The approach, based on a novel data ordering and typestate analysis, consists of preconditions and transformations for automatically determining when it is safe and possibly advantageous to convert sequential streams to parallel, unorder or de-parallelize already parallel streams, and optimize streams involving complex reductions. The approach was implemented as a plug-in to the popular Eclipse IDE, uses the WALA and SAFE analysis frameworks, and was evaluated on 11 Java projects consisting of ∼642K lines of code. We found that 57 of 157 candidate streams (36.31%) were refactorable, and an average speedup of 3.49 on performance tests was observed. The results indicate that the approach is useful in optimizing stream code to their full potential.

Index Terms

refactoring, static analysis, automatic parallelization, typestate analysis, Java 8, streams

1 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]. These APIs incorporate MapReduce-like [7] operations on native data structures such as collections. Below is a "sum of even squares" example in Java [1], where accepts a λ-expression (unit of computation) and results in the list element's square. The λ-expression argument to `filter()` evaluates to true iff the element is even:

```
list.stream().filter(x->x%2==0).map(x->x*x).sum();
```

MapReduce, which helps reduce the complexity of writing parallel programs by facilitating big data processing on multiple nodes using succinct functional-like programming constructs, is a popular programming paradigm for writing a specific class of parallel programs. It makes writing parallel code easier, as writing such code can be difficult due to possible data races, thread interference, and contention [8]–[10]. For instance, the code above can execute in parallel simply by replacing `stream()` with `parallelStream()`.

MapReduce, though, traditionally operates in a highly-distributed environment with no concept of shared memory, while Java 8 Stream processing operates in a single node under multiple threads or cores in a shared memory space. In the latter case, because the data structures for which the MapReduce-like operations execute are on the local machine, problems may arise from the close intimacy between shared memory and the operations being performed. Developers, thus, must *manually* determine whether running stream code in parallel results in an efficient yet interference-free program [11] and ensure that no operations on different threads interleave [12].

Despite the benefits [13, Ch. 1], using streams efficiently requires many subtle considerations. For example, it is often not straight-forward if running a particular operation in parallel is more optimal than running it sequentially due to potential side-effects of λ-expressions, buffering, etc. Other times, using *stateful* λ-expressions, i.e., those whose results depend on any state that may change during execution, can undermine performance due to possible thread contention. In fact, ∼4K stream questions have been posted on Stack Overflow [14], of which ∼5% remain unanswered, suggesting that there is developer confusion surrounding this topic.

In general, these kinds of errors can lead to programs that undermine concurrency, underperform, and are inefficient. Moreover, these problems may not be immediately evident to developers and may require complex interprocedural analysis, a thorough understanding of the intricacies of a particular stream implementation, and knowledge of situational API replacements. Manual analysis and/or refactoring (semantics-preserving, source-to-source transformation) to achieve optimal results can be overwhelming and error- and omission-prone. This problem is exacerbated by the fact that 157 total candidate streams[1] across 11 projects with a 34 project maximum[2] were found during our experiments (section 4), a number that can increase over time as streams rise in popularity. In fact, Mazinanian *et al.* [15] found an increasing trend in the adoption of λ-expressions, an essential part of using the Java 8 stream API, with the number of λ-expressions being

- *R. Khatchadourian and Y. Tang are with the Departments of Computer Science at City University of New York (CUNY) Hunter College and Graduate Center, New York, NY, {10065,10016}, USA.*
  E-mail: {raffi.khatchadourian,ytang3}@{hunter,gradcenter}.cuny.edu
- *M. Bagherzadeh and S. Ahmed are with the Department of Computer Science & Engineering, Oakland University, Rochester, MI, 48309, USA.*
  E-mail: {mbagherzadeh,sfahmed}@oakland.edu

1. Stream candidacy is determined by several analysis parameters that involve performance trade-offs as described in sections 4.2 and 4.3.
2. A stream instance approximation is defined as an invocation to a stream API returning a stream object, e.g., `stream()`, `parallelStream()`.

introduced increasing by two-fold between 2015 and 2016. And, a recent GitHub search by the authors yielded 350K classes importing the `java.util.stream` package.

The operations issued per stream may be many; we found an average of 4.14 operations per stream. Permutating through operation combinations and subsequently assessing performance, if such dedicated tests even exist, can be burdensome. (Manual) interprocedural and type hierarchy analysis may be needed to discover ways to use streams in a particular context optimally.

Recently, attention has been given to retrofitting concurrency on to existing sequential (imperative) programs [16]–[18], translating imperative code to MapReduce [19], verifying and validating correctness of MapReduce-style programs [20]–[23], and improving performance of the underlying MapReduce framework implementation [24]–[27]. Little attention, though, has been paid to mainstream languages utilizing functional-style APIs that facilitate MapReduce-style operations over native data structures like collections. Furthermore, improving imperative-style MapReduce code that has either been handwritten or produced by one the approaches above has, to the best of our knowledge, not been thoroughly considered. Tang *et al.* [11] only briefly present preliminary progress towards this end, while Khatchadourian *et al.* [28] discuss engineering aspects.

The problem may also be handled by compilers or run times, however, refactoring has several benefits, including giving developers more control over where the optimizations take place and making parallel processing explicit. Refactorings can also be issued multiple times, e.g., prior to major releases, and, unlike static checkers, refactorings transform source code, a task that can be otherwise error-prone and involve nuances.

We propose a fully-automated, semantics-preserving refactoring approach that transforms Java 8 stream code for improved performance. The approach is based on a novel data ordering and typestate analysis. The ordering analysis involves inferring when maintaining the order of a data sequence in a particular expression is necessary for semantics preservation. Typestate analysis is a program analysis that augments the type system with "state" and has been traditionally used for preventing resource errors [29], [30]. Here, it is used to identify stream usages that can benefit from "intelligent" parallelization, resulting in more efficient, semantically-equivalent code.

Typestate was chosen to track state changes of streams that may be aliased and to determine the final state following a terminal (reduction) operation. Non-terminal (intermediate) operations may return the receiver stream, in which case traditional typestate applies. However, we augmented typestate to apply when a *new* stream is returned in such situations (cf. sections 3.2 and 3.4). Our approach interprocedurally analyzes relationships between types. It also discovers possible side-effects in arbitrarily complex λ-expressions to safely transform streams to either execute sequentially or in parallel, depending on which refactoring preconditions, which we define, pass. Furthermore, to the best of our knowledge, it is the first automated refactoring technique to integrate typestate.

The refactoring approach was implemented as an open-source Eclipse [31] plug-in that integrates analyses from WALA [32] and SAFE [33]. The evaluation involved studying the effects of our plug-in on 11 Java projects of varying size and domain with a total of $\sim$642K lines of code. Our study indicates that (i) given its interprocedural nature, the (fully automated) analysis cost is reasonable, with an average running time of 0.45 minutes per candidate stream and 6.602 seconds per thousand lines of code, (ii) despite their ease-of-use, parallel streams are not commonly (manually) used in modern Java software, motivating an automated approach, and (iii) the proposed approach is useful in refactoring stream code for greater efficiency despite its conservative nature. This work makes the following contributions:

Precondition formulation and algorithm design. We present a novel refactoring approach for maximizing the efficiency of their Java 8 stream code by automatically determining when it is safe and possibly advantageous to execute streams in parallel, when running streams in parallel can be counterproductive, and when ordering is unnecessarily depriving streams of optimal performance. Our minimally invasive transformation algorithm approach refactors streams for greater parallelism while maintaining original semantics.

Generalized typestate analysis. Streams necessitate several generalizations of typestate analysis, including determining object state at arbitrary points and support for immutable object call chains. Reflection is also combined with (hybrid) typestate analysis to identify initial states.

Implementation and experimental evaluation. To ensure real-world applicability, the approach was implemented as an Eclipse plug-in built on WALA and SAFE and was used to study 11 Java programs that use streams. Our technique successfully refactored 36.31% of candidate streams, and we observed an average speedup of 3.49 during performance testing. The experimentation also gives insights into how streams are used in real-world applications, which can motivate future language and/or API design. These results advance the state of the art in automated tool support for stream code to perform to their full potential.

2 Motivation, Background, and Insight

We present a running example that highlights some of the challenges associated with analyzing and refactoring streams for greater parallelism and increased efficiency. Listing 1 depicts a simplified, hypothetical widget class [5]. `Widget`s have a `Color` (lines 2–3) and a real `weight` (line 4). A constructor is provided (line 6), as well as accessor methods (lines 7–8). `Object` methods `equals()` and `hashCode()` are appropriately overridden (not shown).

Listing 2 portrays code that uses the Java 8 Stream API to process collections of `Widget`s with `weight`s. Listing 2a is the original version, while listing 2b is the improved (but semantically-equivalent) version after our refactoring. In listing 2a, a

Listing 1 A hypothetical widget class.

```
1  class Widget {
2    enum Color {RED, BLUE, GREEN, /*...*/};
3    private Color color;
4    private double weight;
5    public Widget(Color color, double weight)
6      {this.color = color; this.weight = weight;}
7    public Color getColor() {return this.color;}
8    public double getWeight() {return this.weight;}
9    /* override equals() and hashCode() ... */}
```

Listing 2 Snippet of `Widget` collection processing using Java 8 streams based on *java.util.stream (Java SE 9 & JDK 9)* [5].

(a) Stream code snippet prior to refactoring.

```
1  Collection<Widget> unorderedWidgets = new HashSet<>();
2  Collection<Widget> orderedWidgets = new ArrayList<>();
3
4  List<Widget> sortedWidgets = unorderedWidgets
5    .stream()
6    .sorted(Comparator.comparing(Widget::getWeight))
7    .collect(Collectors.toList());
8
9  // collect weights over 43.2 into a set in parallel.
10 Set<Double> heavyWidgetWeightSet = orderedWidgets
11   .parallelStream().map(Widget::getWeight)
12   .filter(w -> w > 43.2).collect(Collectors.toSet());
13
14 // sequentially collect into a list, skipping first 1000.
15 List<Widget> skippedWidgetList = orderedWidgets
16   .stream().skip(1000).collect(Collectors.toList());
17
18 // collect the first green widgets into a list.
19 List<Widget> firstGreenList = orderedWidgets
20   .stream()
21   .filter(w -> w.getColor() == Color.GREEN)
22   .unordered().limit(5).collect(Collectors.toList());
23
24 // collect distinct widget weights into a TreeSet.
25 Set<Double> distinctWeightSet = orderedWidgets
26   .stream().parallel()
27   .map(Widget::getWeight).distinct()
28   .collect(Collectors.toCollection(TreeSet::new));
29
30 // collect distinct widget colors into a HashSet.
31 Set<Color> distinctColorSet = orderedWidgets
32   .parallelStream().map(Widget::getColor)
33   .distinct()
34   .collect(HashSet::new, Set::add, Set::addAll);
35
36 // get the total weight of all distinct widgets.
37 Stream<Widget> unorderedStream =
38   unorderedWidgets.stream();
39 Stream<Widget> orderedStream =
40   orderedWidgets.parallelStream();
41 Stream<Widget> concatStream =
42   Stream.concat(unorderedStream, orderedStream);
43 double distinctWeightSum = concatStream.distinct()
44   .mapToDouble(w -> w.getWeight()).sum();
45
46 // collect widget colors matching a regex.
47 Pattern pattern = Pattern.compile(".*e[a-z]");
48 ArrayList<String> results = new ArrayList<>();
49 orderedWidgets.stream().map(w -> w.getColor())
50   .map(c -> c.toString())
51   .filter(s -> pattern.matcher(s).matches())
52   .forEach(s -> results.add(s));
```

(b) Improved stream client code via refactoring.

```
1  Collection<Widget> unorderedWidgets = new HashSet<>();
2  Collection<Widget> orderedWidgets = new ArrayList<>();
3
4  List<Widget> sortedWidgets = unorderedWidgets
5    .~~stream()~~parallelStream()
6    .sorted(Comparator.comparing(Widget::getWeight))
7    .collect(Collectors.toList());
8
9  // collect weights over 43.2 into a set in parallel.
10 Set<Double> heavyWidgetWeightSet = orderedWidgets
11   .parallelStream().map(Widget::getWeight)
12   .filter(w -> w > 43.2).collect(Collectors.toSet());
13
14 // sequentially collect into a list, skipping first 1000.
15 List<Widget> skippedWidgetList = orderedWidgets
16   .stream().skip(1000).collect(Collectors.toList());
17
18 // collect the first green widgets into a list.
19 List<Widget> firstGreenList = orderedWidgets
20   .~~stream()~~parallelStream()
21   .filter(w -> w.getColor() == Color.GREEN)
22   .unordered().limit(5).collect(Collectors.toList());
23
24 // collect distinct widget weights into a TreeSet.
25 Set<Double> distinctWeightSet = orderedWidgets
26   .stream().~~parallel()~~
27   .map(Widget::getWeight).distinct()
28   .collect(Collectors.toCollection(TreeSet::new));
29
30 // collect distinct widget colors into a HashSet.
31 Set<Color> distinctColorSet = orderedWidgets
32   .parallelStream().map(Widget::getColor)
33   .unordered().distinct()
34   .collect(HashSet::new, Set::add, Set::addAll);
35
36 // get the total weight of all distinct widgets.
37 Stream<Widget> unorderedStream =
38   unorderedWidgets.stream();
39 Stream<Widget> orderedStream =
40   orderedWidgets.parallelStream();
41 Stream<Widget> concatStream =
42   Stream.concat(unorderedStream, orderedStream);
43 double distinctWeightSum = concatStream.distinct()
44   .mapToDouble(w -> w.getWeight()).sum();
45
46 // collect widget colors matching a regex.
47 Pattern pattern = Pattern.compile(".*e[a-z]");
48 ArrayList<String> results = new ArrayList<>();
49 orderedWidgets.stream().map(w -> w.getColor())
50   .map(c -> c.toString())
51   .filter(s -> pattern.matcher(s).matches())
52   .forEach(s -> results.add(s));
```

`Collection` of `Widgets` is declared (line 1) that does not maintain element ordering as `HashSet` does not support it [34]. Note that ordering is dependent on the run time type.

A `stream` (a view representing element sequences supporting MapReduce-style operations) of `unorderedWidgets` is created on line 5. It is sequential, meaning its operations will execute serially. Streams may also have an *encounter order*, which can be dependent on the stream's source. In this case, it will be unordered since `HashSets` are unordered.

On line 6, the stream is `sorted` by the corresponding *intermediate* operation, the result of which is a (possibly) new stream with the encounter order rearranged accordingly. `Widget::getWeight` is a method *reference* denoting the method that should be used for the comparison. Intermediate operations are deferred until a *terminal* operation is executed like `collect()` (line 7). `collect()` is a special kind of (mutable) reduction that aggregates results of prior intermediate operations into a given `Collector`. In this case, it is one that yields a `List`. The result is a `Widget List` sorted by `weight`.

It may be possible to increase performance by running this stream's "pipeline" (i.e., its sequence of operations) in parallel. Listing 2b, line 5 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 result in worse performance due to the multiple passes and/or data buffering required by *stateful* intermediate operations (SIOs) like `sorted()`. Because the stream is *unordered*, the reduction can be done more efficiently as the framework can employ a divide-and-conquer strategy [5].

In contrast, line 2 instantiates an `ArrayList`, which maintains element ordering. Furthermore, a parallel stream is derived from this collection (line 11), with each `Widget` mapped to its weight, each weighted filtered (line 12), and the results `collect`ed into a `Set`. Unlike the previous example, however, no optimizations are available here as an SIO is not included in the pipeline and, as such, the parallel computation does not incur the aforementioned possible performance degradation.

Lines 15–16 create a list of `Widget`s gathered by (sequentially) skipping the first thousand from `orderedWidgets`. Like `sorted()`, `skip()` is also an SIO. Unlike the previous example, though, executing this pipeline in parallel could be counterproductive because, as it is derived from an *ordered* collection, the stream is ordered. It may be possible to unorder the stream (via `unordered()`) so that its pipeline would be more amenable to parallelization. In this situation, however, unordering could alter semantics as the data is assembled into a structure maintaining ordering. As such, the stream remains sequential as element ordering must be preserved.

On lines 19–22, the first five green `Widget`s of `orderedWidgets` are sequentially `collect`ed into a `List` As `limit()` is an SIO, performing this computation in parallel could have adverse effects as the stream is ordered (with the source being `orderedWidgets`). Yet, on line 22, the stream is `unordered`[3] *before* the `limit()` operation. Because the SIO is applied to an unordered stream, to improve performance, the pipeline is refactored to parallel on line 20 in listing 2b. Although similar to the refactoring on line 5, it demonstrates that stream ordering does not solely depend on its source.

A distinct widget weight `Set` is created on lines 25–28. Unlike the previous example, this collection *already* takes place in `parallel`. 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 is *ordered*. Keeping the parallel computation but unordering the stream may improve performance but we would need to determine whether doing so is safe, which can be error-prone if done manually, especially on large and complex projects.

Our insight is 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()` on line 28) to a `Set`, of which subclasses that do not preserve ordering exist. If we could determine that the resulting `Set` is unordered, unordering the stream would be safe since the collection operation would not preserve ordering. The type of the resulting `Set` returned here is determined by the passed `Collector`, in this case, `Collectors.toCollection(TreeSet::new)`, the argument to which is a reference to the default constructor. Unfortunately, since `TreeSet`s preserve ordering, we must keep the stream ordered. Here, to improve performance, it may be advantageous to run this pipeline, perhaps surprisingly, *sequentially* (line 26, listing 2b).

Lines 31–34 map, in parallel, each `Widget` to its `Color`, filter those that are `distinct`, and `collect`ing them into a `Set`. To demonstrate the 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, 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 33 in listing 2b shows the inserted call to `unordered()` immediately prior to `distinct()`. This allows `distinct()` to work more efficiently under parallel computation [5].

Streams can also be stored in variables. Lines 43–44 sum the weight of *all* distinct `Widget`s. Two streams are created from each of the `Widget` collections (lines 38–40), with the former being unordered and the latter ordered (due to their sources) and parallel. The streams are composed via a concatenation operation on line 42, which produces an ordered stream iff *both* of the constituent streams are ordered and a parallel stream if *either* of the streams are parallel [35]. Here, the resulting stream is unordered and parallel, and the computation (lines 43–44) needs no further optimization.

Lastly, on lines 47–52, `Widget` colors matching a particular regular expression are sequentially accumulated 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 λ-expression `s -> results.add(s)`. The stream is not refactored to parallel because of the λ-expression's possible side-effects. Otherwise, the unsynchronized `ArrayList` could cause incorrect results due to thread scheduling, possibly altering semantics. Adding synchronization would solve that problem but cause thread contention, undermining the benefit of parallelism [5].

Manual analysis of stream client code can be complicated, even as seen in this simplified example. It necessitates a thorough understanding of the intricacies of the underlying computational model, a problem which can be compounded in more extensive programs. As streaming APIs become more pervasive, it would be extremely valuable to developers, particularly those not previously familiar with functional programming, if automation can assist them in writing efficient stream code.

2.1 Concurrent Reductions

3. The use of `unordered()` is deliberate despite nondeterminism.

Listing 3 Complex reduction operation refactoring example based on *java.util.stream (Java SE 9 & JDK 9)* [5].

(a) Stream code snippet of complex reductions before refactoring.

```
53 Map<Color, List<Widget>> widgetsByColor =
54   orderedWidgets
55     .parallelStream()
56     .collect(Collectors.groupingBy(
57       Widget::getColor));
58
59 widgetsByColor = unorderedWidgets
60   .stream()
61   .collect(Collectors.groupingBy(
62     Widget::getColor));
63
64 Map<Color, Set<Widget>> widgetsByColor2 =
65   orderedWidgets
66     .stream()
67     .collect(Collectors.groupingBy(
68       Widget::getColor, HashMap::new,
69         Collectors.toSet()));
70
71 widgetsByColor2 = orderedWidgets
72   .stream()
73   .collect(Collectors.groupingByConcurrent(
74     Widget::getColor, ConcurrentHashMap::new,
75       Collectors.toCollection(LinkedHashSet::new)));
```

(b) Improved complex reduction stream code via refactoring.

```
53 Map<Color, List<Widget>> widgetsByColor =
54   orderedWidgets
55     .~~parallelStream()~~stream()
56     .collect(Collectors.groupingBy(
57       Widget::getColor));
58
59 widgetsByColor = unorderedWidgets
60   .~~stream()~~parallelStream()
61   .collect(Collectors.groupingByConcurrent(
62     Widget::getColor));
63
64 Map<Color, Set<Widget>> widgetsByColor2 =
65   orderedWidgets
66     .~~stream()~~parallelStream()
67     .collect(Collectors.groupingByConcurrent(
68       Widget::getColor, ConcurrentHashMap::new,
69         Collectors.toSet()));
70
71 widgetsByColor2 = orderedWidgets
72   .stream()
73   .collect(Collectors.groupingBy~~Concurrent~~(
74     Widget::getColor, ~~Concurrent~~HashMap::new,
75       Collectors.toCollection(LinkedHashSet::new)));
```

Listing 3 portrays several, more complex reduction operations that produce `Map`s (i.e., `groupingBy` operations). The statement beginning on line 53, listing 3a groups `Widget`s by `Color` into a `Map` in parallel. However, such operations may be counterproductive if performed in parallel, particularly those where the combining operation is expensive, especially as is the case with certain `Map` implementations [5]. For example, on lines 56–57, intermediate map results must *merged* together because the container being used for accumulation cannot be concurrently modified, i.e., the entire `Map` must be operated on as a whole [17].

This problem can be mitigated by using a *concurrent* reduction. In this situation, a container type supporting *concurrent* modification, e.g., a `ConcurrentHashMap` [36] is used instead, allowing for safe shared, concurrent manipulation via multiple threads. This eliminates the need for `Map` merging, possibly improving performance under parallel computations [5]. However, such a reduction can *only* be used if ordering of the value `Collection` is unimportant. In other words, because the `Map` will be concurrently modified, the order in which the `Widget`s are placed into the value container is nondeterministic and dependent on thread scheduling. As the value container being utilized in this example if of type `List`, a data structure *maintaining* ordering, *and* that the stream is *ordered*, a concurrent reduction *cannot* be used. In other words, the lack of importance of ordering is a precondition for a refactoring to a concurrent reduction. In this case, it is advantageous to execute this pipeline *sequentially* instead to avoid the merge-based accumulation overhead under parallel computation, as shown in listing 3b.

In contrast to the statement at line 53, the statement beginning at line 59 performs a similar operation but sequentially and on an *unordered* stream. As such, to begin with, the data is unordered, thus, using a `Collector` that produces a value `Collection` with ordering is inconsequential; we would be imposing an ordering that is arbitrary. In other words, despite the value `Collection` being a `List` (which maintains ordering), this code can be refactored to execute in a *parallel* stream pipeline, as shown in listing 3b, line 60. However, this is *not* sufficient to utilize a concurrent reduction. The `Collector` must also utilize a *concurrent* `Collection` to retain intermediate results, i.e., a *concurrent* `Collector`. This can be accomplished by replacing `groupingBy()` with `groupingByConcurrent()` (line 61).

Unlike line 60, the (sequential) `Stream` created on line 66 is *ordered* due to its source, namely, `orderedWidgets` (line 65). And, unlike lines 56–57, the `Collector` used, i.e., the reference of which is returned by the call to `Collectors.toSet()`, on line 69 is one that uses a `Set` rather than a `List` as a value `Collection` in the `groupingBy()` operation. In this case, `Collectors.toSet()` returns a `Collector` *known* to be *unordered*, i.e., it does not maintain the encounter order of its input. As such, despite the stream being ordered, since the `Collector` is unordered, this computation can be refactored to utilize a concurrent reduction (listing 3b, line 67), which also involves refactoring the resulting `Map` type to its *concurrent* version (line 68).[4]

The stream created on line 72 is also ordered, however, in contrast to the `Collector` used on line 69, the `Collector` utilized on line 75 is *ordered* because it uses a value `Collection`, in this case, a `LinkedHashSet`, that maintains element ordering. As such, a concurrent reduction cannot be utilized here. In listing 3b, the stream execution remains sequential but the developer inadvertently used a concurrent `groupingBy` operation. Leaving it as such in the sequential case would cause unnecessary synchronization code to execute. Thus, the `groupingBy` operation has been refactored to a non-concurrent operation (listing 3b, line 73), which also necessitates refactoring the resulting `Map` type to the non-concurrent version (line 74).

4. The current API does not allow a particular `Collector` to be specified without also specifying the resulting `Map` type (cf. listing 3, lines 56–57).

TABLE 1
CONVERT SEQUENTIAL STREAM TO PARALLEL preconditions.

	exe	ord	se	SIO	ROM	transformation
P1	seq	unord	*F*	N/A	N/A	Convert to para.
P2	seq	ord	*F*	*F*	N/A	Convert to para.
P3	seq	ord	*F*	*T*	*F*	Unorder, convert to para.

TABLE 2
OPTIMIZE PARALLEL STREAM preconditions.

	exe	ord	SIO	ROM	transformation
P4	para	ord	*T*	*F*	Unorder.
P5	para	ord	*T*	*T*	Convert to seq.

3 OPTIMIZATION APPROACH

3.1 Intelligent Parallelization Refactorings

We propose three new refactorings, i.e., CONVERT SEQUENTIAL STREAM TO PARALLEL, OPTIMIZE PARALLEL STREAM, and OPTIMIZE COMPLEX MUTABLE REDUCTION. The first deals with determining if it is possibly advantageous (performance-wise, based on type analysis) and safe (e.g., no race conditions, semantics alterations) to transform a sequential stream to parallel. The second deals with a stream that is *already* parallel and ascertains the steps (transformations) necessary to possibly improve its performance, including *unordering* and *converting* the stream to sequential. The third deals with mutable reductions involving map merging (e.g., `groupingBy()`), optimizing certain kinds of mutable reductions that can be inefficient if executed in parallel in particular cases. Here, `Collector`s may be changed from non-concurrent to concurrent or vice versa.

3.1.1 Converting Sequential Streams to Parallel

Table 1 portrays the preconditions for our proposed CONVERT SEQUENTIAL STREAM TO PARALLEL refactoring. It lists the conditions that must hold for the transformation to be both semantics-preserving as well as possibly advantageous, i.e., resulting in a possible performance gain. Column **exe** denotes the stream's execution mode, i.e., whether, upon the execution of a terminal operation, its associated pipeline will execute sequentially or in parallel ("seq" is sequential and "para" is parallel). Column **ord** denotes whether the stream is associated with an encounter order, i.e., whether elements of the stream *must* be visited in a particular order ("ord" is ordered and "unord" is unordered). Column **se** represents whether any behavioral parameters (λ-expressions) that will execute during the stream's pipeline have possible side-effects. Column **SIO** constitutes whether the pipeline has any stateful intermediate operations. Column **ROM** represents whether the encounter order must be preserved by the result of the terminal reduction operation. A *T* denotes that the reduction result depends on the encounter order of a previous (intermediate) operation. Conversely, an *F* signifies that any ordering of the input operation to the reduction need not be preserved. Column **transformation** characterizes the transformation actions to take when the corresponding precondition passes (note the conditions are mutually exclusive). *N/A* is either *T* or *F*.

A stream passing P1 is one that is sequential, unordered, and has no side-effects. Because this stream is already unordered, whether or not its pipeline contains an SIO is inconsequential. Since the stream is unordered, any SIOs can run efficiently in parallel. Moreover, preserving the ordering of the reduction is also inconsequential as no original ordering exists. Here, it is both safe and possibly advantageous to run the stream pipeline in parallel. The stream derived from `unorderedWidgets` on line 5, listing 2 is an example of a stream passing P1. A stream passing P2 is also sequential and free of λ-expressions containing side-effects. However, such streams are ordered, meaning that the refactoring only takes place if no SIOs exist. P3, on the other hand, will allow such a refactoring to occur, i.e., if an SIO exists, *only* if the ordering of the reduction's result is inconsequential, i.e., the reduction ordering need not be maintained. As such, the stream can be *unordered* immediately before the (first) SIO (as performed on line 33, listing 2b). The stream created on line 16, listing 2 is an example of a stream failing this precondition.

3.1.2 Optimizing Parallel Streams

Table 2 depicts the preconditions for the OPTIMIZE PARALLEL STREAM refactoring. Here, the stream in question is *already* parallel. A stream passing either precondition is one that is ordered and whose pipeline contains an SIO. Streams passing P4 are ones where the reduction does not need to preserve the stream's encounter order, i.e., **ROM** is *F*. An example is depicted on line 32, listing 2. Under these circumstances, the stream can be explicitly unordered immediately before the (first) SIO, as done on line 33 of listing 2b. Streams passing P5, on the other hand, are ones that the reduction ordering *does*

TABLE 3
OPTIMIZE COMPLEX MUTABLE REDUCTION preconditions.

	exe	ord	col	ROM	transformation
P6	seq	ord	concur	*T*	Convert col to nconcur.
P7	para	ord	nconcur	*T*	Convert stream to seq.
P8	para	ord	concur	*T*	Convert col to nconcur and stream to seq.
P9	seq	ord	nconcur	*F*	Convert col to concur and stream to para.
P10	seq	ord	concur	*F*	Convert stream to para.
P11	para	N/A	nconcur	*F*	Convert col to concur.
P12	seq	unord	nconcur	N/A	Convert col to concur and stream to para.
P13	seq	unord	concur	N/A	Convert stream to para.

matter, e.g., the stream created on line 26. To possibly improve performance, such streams are transformed to *sequential* (line 26, listing 2b).[5]

3.1.3 Optimizing Complex Mutable Reductions

Table 3 presents the preconditions for the OPTIMIZE COMPLEX MUTABLE REDUCTION refactoring. Column **col** is collector kind ("concur" is concurrent and "nconcur" is non-concurrent). Any transformation involving converting a collector from or to a concurrent collector necessitates also transforming the associated `Map`, if provided, to match as shown in listing 3b. Furthermore, any stream being transformed as part of this refactoring must also pass preconditions for the associated refactoring, e.g., a stream being transformed from sequential to parallel must not include side-effects in its pipeline.

Preconditions 6–8 focus on situations where ordered streams must have their ordering maintained. As detailed in section 2.1, streams passing these conditions should run sequentially and have matching (non-concurrent) collectors. Preconditions 9–11, on the other hand, deal with situations where streams are initially ordered but the reduction order does not matter. These streams can be transformed to run in parallel (given they pass the conditions in table 1) and should have matching (concurrent) collectors. Lastly, P12 and P13 deal with streams that are initially sequential and unordered. Regardless of whether the reduction order matters, w.r.t. mutable reduction, it is possibly advantagoues to run such streams in parallel with concurrent collectors (lines 59–62, listing 3b).

3.2 Identifying Stream Creation

Identifying where in the code streams are created is imperative for several reasons. First, streams are typically derived from a source (e.g., a collection) and take on its characteristics (e.g., ordering). This is used in tracking stream attributes across their pipeline (section 3.3). Second, for streams passing preconditions, the creation site serves a significant role in the transformation (section 3.7). In other words, it helps locate where the transformation should take place.

There are several ways to create streams, including being derived from `Collection`s, being created from arrays (e.g., `Arrays.stream()`), and via static factory methods (e.g., `IntStream.range()`). Streams may also be directly created via constructors, but it is not typical of clients, which are our focus. We consider stream creation point approximations as any expression evaluating to a type implementing the `java.util.stream.BaseStream` interface, which is the top-level stream interface. We exclude, however, streams emanating from intermediate operations, i.e., instance methods whose receiver *and* return types implement the stream interface, as such methods are not likely to produce new streams but rather ones derived from the receiver but with different attributes.

3.3 Tracking Streams and Their Attributes

We discuss our approach to tracking streams and their attributes (i.e., state) using a series of labeled transition systems (LTSs). The LTSs are used in in the typestate analysis (section 3.4).

3.3.1 Execution Mode

Definition 1. The LTS E is a tuple $E=(E_S,E_\Lambda,E_\rightarrow)$ where $E_S=\{\bot,seq,para\}$ is the set of states, E_Λ is a set of labels, and $E_\rightarrow$ is a set of labeled transitions.

The labels E_Λ corresponds to method calls that either create or transform the execution mode of streams. We denote the initial stream ("phantom") state as $\bot$. Different stream creation methods may transition the newly created stream to one that is either sequential or parallel. Figure 1 diagrammatically portrays a proper subset of the relation $E_\rightarrow$ (`Col` is `Collection`). Transitions stemming from the $\bot$ state represent stream creation methods (section 3.2). As an example, the stream on line 5, listing 2a would transition from $\bot$ to seq, while the stream at line 26 would transition from seq to $para$ as a result of the corresponding call.

5. Unlike table 1, side-effects are not considered here as our approach is a *performance*-based refactoring. De-parallelizing streams with possible side-effects would be considered a *correctness*-based transformation and is out of scope w.r.t. this work.

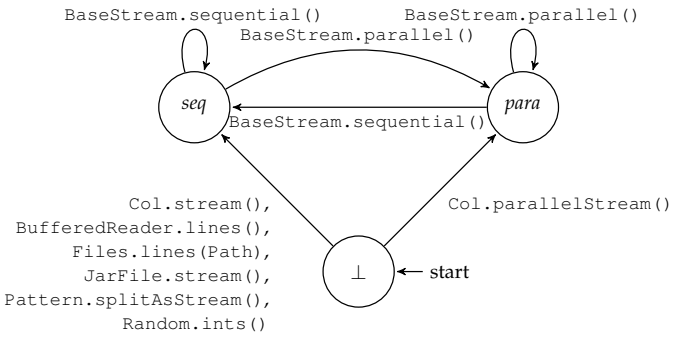


Fig. 1. A proper subset of the relation $E_{\rightarrow}$ in $E=(E_S,E_\Lambda,E_{\rightarrow})$.

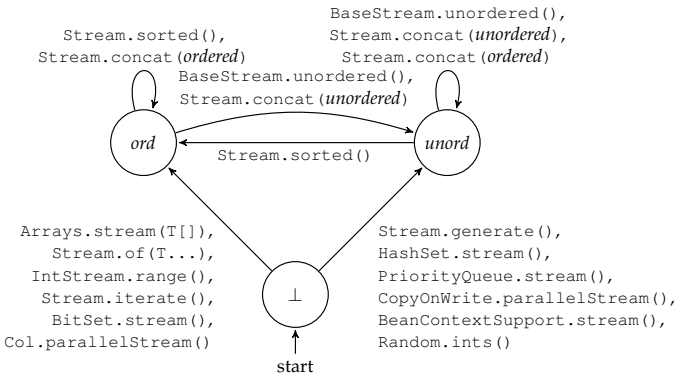


Fig. 2. A proper subset of the relation $O_{\rightarrow}$ in $O=(O_S,O_\Lambda,O_{\rightarrow})$.

3.3.2 Ordering

Whether a stream has an encounter order depends on the stream source (run time) type and the intermediate operations. Certain sources (e.g., `List`) are intrinsically ordered, whereas others (e.g., `HashSet`) are not. Some intermediate operations (e.g., `sorted()`) may impose an encounter order on an otherwise unordered stream, and others may render an ordered stream unordered (e.g., `unordered()`). Further, some terminal operations may ignore encounter order (e.g., `forEach()`) while others (e.g., `forEachOrderer()`) abide by it [5].

Definition 2. The LTS O for tracking stream ordering is the tuple $O=(O_S,O_\Lambda,O_{\rightarrow})$ where $O_S=\{\bot,ord,unord\}$ and other components are in line with definition 1.

Figure 2 portrays a proper subset of the relation $O_{\rightarrow}$, which depicts valid transitions between stream ordering modes. As with E_S, $\bot$ is a phantom initial state immediately before stream creation. For presentation, the static method `Stream.concat(Stream,Stream)` is modeled as an instance method where the receiver represents the first parameter, i.e., the origin state is that of the first parameter, and the *state* of the second parameter is the sole explicit parameter.

For instance, the stream on line 5, listing 2a would transition from $\bot$ to $unord$ due to `HashSet.stream()`. Although the compile-time type of `unorderedWidgets` is `Collection` (line 1), we use an interprocedural type inference algorithm (explained next) to approximate `HashSet`. The stream at line 26 would transition from $\bot$ to ord state as a result of `orderedWidgets` having the type `ArrayList` (line 2).

Approximating Stream Source Types and Characteristics: The fact that stream ordering can depend on the run time type of its source necessitates that its type be approximated. For this, we use an interprocedural type inference algorithm via points-to analysis [37] that computes the possible run time types of the receiver from which the stream is created. Once the type is obtained, whether source types produce ordered or unordered streams is determined via reflection. While details are in section 4.1, briefly, the type is reflectively instantiated and its `Spliterator` [38] extracted. Then, stream characteristics, e.g., ordering, are queried [38]. This is enabled by the fact that collections and other types supporting streams do not typically change their ordering characteristics dynamically. `Collector` attributes are determined in a similar fashion.

Using reflection in this way amounts to a kind of *hybrid* typestate analysis where initial states are determined via dynamic analysis. If reflection fails, e.g., an abstract type is inferred, the default is to ordered and sequential. This choice is safe considering that there is no net effect caused by our proposed transformations, thus preserving semantics. Furthermore, to prevent ambiguity in state transitions, it is required that each inferred type have the same attributes.

3.4 Tracking Stream Pipelines

Tracking stream pipelines is essential in determining satisfied preconditions. Pipelines can arbitrarily involve multiple methods and classes as well as be data-dependent (i.e., spanning multiple branches). In fact, during our evaluation (section 4), we found many real-world examples that use streams interprocedurally.

Our automated refactoring approach involves developing a variant of typestate analysis [29], [30] to track stream pipelines and determine stream state when a terminal operation is issued. Typestate analysis is a program analysis that augments the type system with "state" information and has been traditionally used for prevention of program errors such as those related to resource usage. It works by assigning each variable an initial ($\perp$) state. Then, (mutating) method calls transition the object's state. States are represented by a lattice and possible transitions are represented by LTSs. If each method call sequence on the receiver does not eventually transition the object back to the $\perp$ state, the object may be left in a nonsensical state, indicating the potential presence of a bug.

Our typestate analysis makes use of a call graph, which is created via a k-CFA call graph construction algorithm [39], making our analysis both object and context sensitive (the context being the k-length call string). In other words, it adds context so that method calls to an object creation site (**`new`** operator) can be distinguished from one another [40, Ch. 3.6]. It is used here to consider client-side invocations of API calls as object creations. Setting $k = 1$ would not suffice as the analysis would not consider the client contexts as stream creations. As such, at least for streams, k must be $>=2$. Although k is flexible in our approach, we use $k=2$ as the default for streams and $k=1$ elsewhere. Section 4.2.1 discusses how k was set during our experiments, as well as a heuristic to help guide developers in choosing a sufficient k.

We formulate a variant of typestate since operations like `sorted()` return (possibly) new streams derived from the receiver stream with their attributes altered. Definition 3 portrays the formalism capturing the concept of typestate analysis used in the remainder of this section. Several generalizations are made to extract typestate at a particular program point.

Definition 3 (Typestate Analysis)**.** Define $TState_{LTS}(i_s, exp) = S$ where LTS is a labeled transition system, i_s a stream instance, exp an expression, and to be the possible states S of i_s at exp according to LTS.

In definition 3, exp, an expression in the Abstract Syntax Tree (AST), is used to expose the internal details of the analysis. Typically, typestate is used to validate complete statement sequences. Regarding definition 3, this would be analogous to exp corresponding to a node associated with the last statement of the program. In our case, we are interested in typestates at particular program points; otherwise, we may not be able to depict typestate at the execution of the terminal operation accurately.

As an example, let i_s be the stream on line 5, listing 2a and exp the expression `collect()` at line 7. Then, $TState_O(i_s$,`collect(..)`$)$ $\{ord\}$.

Traditional typestate analysis is used with (mutating) methods that alter object state. The Stream API, though, is written in an immutable style where each operation returns a stream reference that may refer to a new object. A naïve approach may involve tracking the typestates of the returned references from intermediate operations. Doing so, however, would produce an undesirable result as each stream object would be at the starting state.

Section 3.3 treats intermediate operations as being (perhaps void returning) methods that mutate the state of the receiver. This makes the formalism concise. However, in actuality, intermediate operations are *value returning methods*, returning a reference to the same (general) type as the receiver. As such, the style of this API is that of immutability, i.e., "manipulating" a stream involves creating a new stream based on an existing one. In such cases, the receiver is then considered *consumed*, i.e., any additional operations on the receiver would result in a run time exception, similar to linear type systems [41].

Our generalized typestate analysis works by tracking the state of stream instances as follows. For a given expression, the analysis yields a set of possible states for a given instance following the evaluation of the expression. Due to the API style, a typestate analysis that has a notion of instances that are *based on* other instances is needed. As such, we compute the typestate of individual streams and proceed to *merge* the typestates to obtain the *final* typestate after the expression of where a terminal operation consumes the stream. The final typestate is derived at this point because that is when all of the (queued) intermediate operations will execute. Moreover, the final typestate is a *set* due to dataflow analysis of possible branching.

3.4.1 Intermediate Streams

A stream is created via APIs calls stemming from the $\perp$ state as discussed in section 3.3. Recall that intermediate operations may or may not also create streams based on the receiver. We coin such streams as *intermediate* streams as they are used to progress the computation to a final result. Moreover, intermediate streams cannot be instantiated alone; they must be based on (or derived from) existing ones. If an intermediate stream is derived from another intermediate stream, then, there must exist a chain of intermediate stream creations that starts at a non-intermediate stream. Due to conditional branching and polymorphism, there may be multiple such (possible) chains. Intermediate streams must be appropriately arranged so that the correct final state may be computed.

To sequence stream instances, we require a "predecessor" function $Pred(i_s) = \{i_{s_1}, ..., i_{s_n}\}$ that maps a stream i_s to a set of streams that may have been used to create i_s. $Pred(i_s)$ is computed by using the points-to set of the reference used as the receiver when i_s was instantiated.

We now demonstrate the predecessor function using the code in listing 4a. Suppose we would like to know the state of the stream referred to by `s2` before the commencement of the terminal operation `count()` on line 10. The points-to set

Listing 4 Sequencing stream instance derivations.

(a) Before refactoring.

```
1  void m(int x) {
2    Stream s1 =
3      o.stream();//1
4    Stream s2 = null;
5    if (x > 0)
6      s2 = s1.filter(..);//2
7    else
8      s2 = s1.parallel()
9        .filter(..);//3
10   int c = s2.count();}
```

(b) After refactoring.

```
1  void m(int x) {
2    Stream s1 =
3      o.stream()parallelStream();
4    Stream s2 = null;
5    if (x > 0)
6      s2 = s1.filter(..);
7    else
8      s2 = s1.parallel()
9        .filter(..);
10   int c = s2.count();}
```

of `s2` consists of the objects created by each of the `filter()` operations on lines 6 and 9, respectively. These allocation sites have been numbered in comments in the source code using comments.[6] As such, we have that $PointsTo(\texttt{s2}) = \{\texttt{filter()}_2, \texttt{filter()}_3\}$.[7] For the first call to `filter()`, `s1` refers to the receiver. Because $PointsTo(\texttt{s1}) = \{\texttt{stream()}_1\}$ (from line 3), we have that $Pred(\texttt{filter()}_2) = \texttt{stream()}_1$. Finally, because `stream()` is *not* an intermediate operation, we have that $Pred(\texttt{stream()}_1) = \varnothing$.

Conversely, for the call to `filter()` on line 9, the receiver is the result of `s1.parallel()`. Interestingly, no allocation takes place here as `parallel()` simply sets a field value in the receiver and returns its reference, i.e., `s1`. Since $PointsTo(\texttt{s1}) = \{\texttt{stream()}_1\}$, we also have that $Pred(\texttt{filter()}_3) = \{\texttt{stream()}_1\}$. Definition 4 describes this function more generally.

Definition 4 (Predecessor Objects)**.** Define $Pred(\texttt{o.m()}) = \{i_1, i_2, ..., i_n\}$ where `o` is an object reference, `m` a method, `o.m()` results in an object reference, and $i_k \in \{i_1, i_2, ..., i_n\}$ for $1 \leq k \leq n$ an abstract heap object identifier:

$$Pred(\texttt{o.m()}) = \begin{cases} \varnothing & \text{if } \texttt{m()} \text{ is not intermediate.} \\ PointsTo(\texttt{o}) & \text{o.w.} \end{cases}$$

3.4.2 Typestate Merging

Since intermediate operations possibly create new streams based on the receiver, the typestate analysis will generate different states for any stream produced by an intermediate operation. We are interested in, however, the final state just before the commencement of the terminal operation, which results in stream consumption. Recall from section 3.3.1 that $\bot$ models an initial state. As such, $\bot$ will symbolize the initial state of intermediate streams. In other words, although an intermediate stream may "inherit" state from the stream from which it is derived, in our formalism, we use $\bot$ as a placeholder until we can derive what exactly the state should be. To this end, we introduce the concept of *typestate merging*.

First, we define a state selection function that results in the first state if it is not $\bot$ and the second state otherwise:

Definition 5 (State Selection)**.** Define $Select\colon\ S \times S \rightarrow S$ to be the state selection function:

$$Select(s_i, s_j) = \begin{cases} s_j & \text{if } s_i = \bot \\ s_i & \text{o.w.} \end{cases}$$

Definition 5 "selects" the "most recent" state in the case that the typestate analysis determines it for the instance under question and a previous state otherwise. For example, let $s_i = \bot$ and $s_j = para$. Then, $Select(s_i, s_j) = para$. Likewise, let $s_i = unord$ and $s_j = ord$. Then, $Select(s_i, s_j) = unord$.

Next, we define the state merging function, which allows us to merge two sets of states, as follows:

Definition 6 (State Merging)**.** Define $Merge(S_i, S_j) = S$ to be the typestate merging function:

$$Merge(S_i, S_j) = \begin{cases} S_i & \text{if } S_j = \varnothing \\ S_j & \text{if } S_i = \varnothing \\ \{Select(s_i, s_j) \mid s_i \in S_i \wedge s_j \in S_j\} & \text{o.w.} \end{cases}$$

As an example, let $S_i = \{\bot\}$ and $S_j = \{seq, para\}$. Then, $Merge(S_i, S_j) = \{seq, para\}$. Likewise, let $S_i = \{ord, unord\}$ and $S_j = \{ord, unord\}$. Then, $Merge(S_i, S_j) = \{unord, ord\}$.

Finally, we define the notation of merged typestate analysis:

6. For presentation purposes, we treat API calls as abstract object creation sites instead of the traditional **`new`** operators as in [30]. However, setting $k > 1$ and using call-string context sensitivity is how this effect is actually achieved.

7. We purposely use API-level allocation sites so as to remain as implementation-neutral as possible.

Definition 7 (Merged Typestate Analysis)**.** Define $MTState_{LTS}(i_s,exp)=S$ where LTS is a labeled transition system, i_s a stream, exp an expression, to be the typestate analysis merging function:

$$MTState_{LTS}(i_s, \quad exp) \quad = \quad \begin{cases} TState_{LTS}(i_s,exp) & \text{if } Pred(\texttt{o.m()})=\varnothing \\ \bigcup\limits_{i_{s_k}\in Pred(i_s)} Merge(TState_{LTS}(i_s,exp), \\ \qquad MTState_{LTS}(i_{s_k},exp)) & \text{o.w.} \end{cases}$$

This final function aggregates typestate over the complete method call chain until the terminal operation after exp. For example, let $i_s=\texttt{filter()}_2\in PointsTo(\texttt{s2})$ and $exp=\texttt{count(..)}$ from listing 4a. Then $MTState_O(i_s,exp)$

$$\begin{aligned} &= \{Merge(TState_O(i_s,exp),MTState_O(\texttt{stream()}_1,exp))\} \\ &= \{Merge(TState_O(i_s,exp),TState_O(\texttt{stream()}_1,exp))\} \\ &= \{Merge(\{\bot\},\{seq,para\})\} \\ &= \{Select(\bot,seq),Select(\bot,para)\} \\ &= \{seq,para\} \end{aligned}$$

3.4.3 Identifying Origin Streams

Once a stream's merged typestate at the terminal operation has been determined, the relationship between this stream and the original (non-intermediate) stream is examined. Because a series of intermediate operations can form a chain of streams starting at a non-intermediate stream, the stream being consumed by a terminal operation may not be the original stream, i.e., it may be one of the derived, intermediate streams. We denote original streams in the computation as *origin* streams. In terms of definition 7, origin streams are those processed in the base case.

An intermediate stream may have multiple origin streams due to branching, polymorphism, etc. Identifying origin streams is important in tracking the complete stream pipeline, as well locating potential areas where refactoring transformations may take place (as in section 3.7). Moreover, identify the stream origin as, e.g., initial stream ordering is dependent on the type from which it was derived or the (static) method that was used to create it. We define the concept of origin objects more generally as follows:

Definition 8 (Origin Objects)**.** Define $Origins(\texttt{o.m()})=\{i_1,i_2,...,i_n\}$ where `o` is an object reference, `m()` a method, `o.m()` results in an object reference, and $i_k\in\{i_1,i_2,...,i_n\}$ for $1\leq k\leq n$ an abstract heap object identifier:

$$Origins(\texttt{o.m()}) \quad = \quad \begin{cases} \varnothing & \text{if } \texttt{o.m() == } \textbf{null}. \\ \{\texttt{o.m()}\} & \text{if } Pred(\texttt{o.m()})=\varnothing \\ \bigcup\limits_{i_j\in Pred(\texttt{o.m()})} Origins(i_j) & \text{o.w.} \end{cases}$$

To illustrate, consider the code in listing 4a. We have that $Origins(\texttt{s2.count()})$

$$\begin{aligned} &= Origins(\texttt{filter()}_2)\cup Origins(\texttt{filter()}_3) \\ &= Origins(\texttt{stream()}_1)\cup Origins(\texttt{stream()}_1) \\ &= \{\texttt{stream()}_1\}\cup\{\texttt{stream()}_1\} \\ &= \{\texttt{stream()}_1\} \end{aligned}$$

3.5 Inferring Behavioral Parameter Side-effects

In this section, we more formally define what it means for behavioral parameters (λ-expressions)[8] that will execute as part of a stream's pipeline to possibly contain side-effects. Side-effect considerations are part of the refactoring precondition checks in table 1 and are an essential part of determining whether a sequential stream can be safely converted to one whose pipeline executes in parallel. The following more formally defines the λ-expressions associated with streams:

Definition 9 (Stream λ-expressions)**.** Define the function $\lambda(i_s)=\lambda exp$ that maps a streams instance i_s to a λ-expression λexp used in creating i_s. If no λ-expression is used creating i_s, then $\lambda exp=\bullet$, an "empty" expression not associated with any meaningful instruction (no-op).

Let i_s be the stream created as a result of the `filter()` operation on line 12 of listing 2. Then, $\lambda(i_s)=$`w -> w > 43.2`. Likewise, let i_s be the stream that results from `skip()` on line 16. Then, $\lambda(i_s)=\bullet$.

Next, we describe the meaning of λ-expressions to contain side-effects:

8. Note that, in Java, behavioral parameters can be represented by any object performing an action, e.g., method references.

TABLE 4
"Reduction ordering matters" (ROM) lookup table.

r. type	ord	t. operation	ROM
non-scalar	unord	N/A	*F*
non-scalar	ord	N/A	*T*
void	N/A	`forEach()`	*F*
void	N/A	`forEachOrdered()`	*T*
scalar	N/A	`sum()`*	*F*
scalar	N/A	`min()`	*F*
scalar	N/A	`max()`	*F*
scalar	N/A	`count()`	*F*
scalar	N/A	`average()`*	*F*
scalar	N/A	`summaryStatistics()`*	*F*
scalar	N/A	`anyMatch()`	*F*
scalar	N/A	`allMatch()`	*F*
scalar	N/A	`noneMatch()`	*F*
scalar	N/A	`findFirst()`	*T*
scalar	N/A	`findAny()`	*F*
scalar	N/A	`collect()`	?
scalar	N/A	`reduce()`	?

* Only applicable to numeric streams.

Definition 10 (λ-expression Side-effects)**.** Define the predicate $LSideEffects(\lambda exp)$ on λ-expressions to be true iff λexp modifies a heap location.

For instance, let λexp represent `w -> w > 43.2` from above. Then, we have $\neg LSideEffects(\lambda exp)$ since `w` does not represent a heap location (i.e., `w` is a **`double`**, a primitive type allocated on the stack). Let λexp represent `s -> results.add(s)` from line 52 of listing 2. Then, we have $LSideEffects(\lambda exp)$ since `s` is a `String`, a reference type allocated on the heap.

Definition 11 (Stream Side-effects)**.** Define the predicate $SSideEffects(i_s)$ on streams to be true iff i_s is associated with a pipeline whose operations contain a λ-expression with possible side-effects:

$$SSideEffects(i_s) \equiv LSideEffects(\lambda(i_s)) \vee \exists \texttt{o.m(p)} \, [i_s \in PointsTo(\texttt{o}) \wedge \texttt{m} \text{ is a term op} \wedge \texttt{p} \text{ is a } \lambda\text{-exp} \wedge LSideEffects(p)] \vee \bigvee_{i_{s_j} \in Pred(i_s)} SSideEffects(i_{s_j})$$

More informally, a stream instance i_s has possible side-effects, i.e., $SSideEffects(i_s)$, iff either a λ-expression used in building i_s, i.e., $\lambda(i_s)$, has side-effects, i.e., $LSideEffects(\lambda(i_s))$, or there exists a call `o.m(p)` such that `o` refers to i_s, i.e., $i_s \in PointsTo(\texttt{o})$, `m` is a terminal operation, and parameter `p` is a λ-expression with possible side-effects, i.e., $LSideEffects(p)$, or if there is a predecessor stream instance i_{s_j} of i_s, i.e., $i_{s_j} \in Pred(i_s)$, that has possible side-effects, i.e., $SSideEffects(i_{s_j})$.

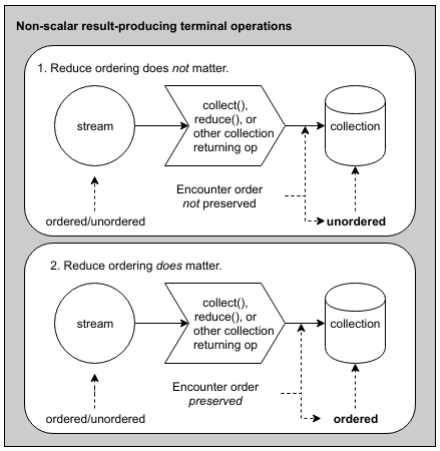
Let i_s be the stream created on line 51 of listing 2, i.e., `filter(s->pattern.matcher(s).matches())`. Assume that the λ-expression does not contain side-effects. Then, we have $\neg LSideEffects(\lambda(i_s)) \equiv \neg LSideEffects($`s -> pattern.matcher(s).mat`
However, consider the terminal operation called on line 52, i.e., `forEach(s->results.add(s))`. We have that $LSideEffects($`s -> res`
since `result` refers to a heap object, namely, the one allocated at line 48. Thus, we have that $SSideEffects(i_s)$.

3.6 Determining Whether Reduction Ordering Matters

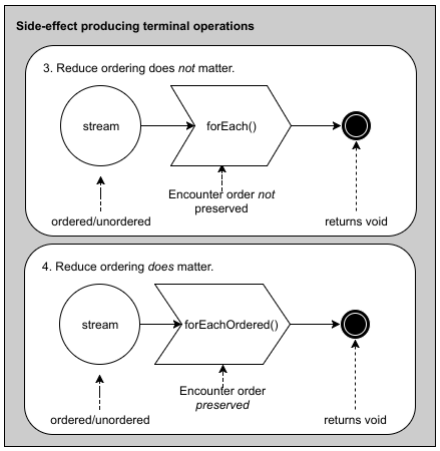
To obtain a result from stream computations, a terminal (reduction) operation must be issued. Determining whether the ordering of the stream immediately before the reduction matters (ROM) equates to discovering whether the reduction result is the same regardless of whether the stream is ordered or not. In other words, the result of the terminal operation does not depend on the ordering of the stream for which the operation is invoked, i.e., the value when the stream is ordered is equal to the value when the stream is unordered. Some reductions (terminal operations) do not return a value, i.e., they are void returning methods. In these cases, the *behavior* rather than the resulting value should be the same. Terminal operations fall into two categories, namely, those that produce a result, e.g., `count()`, and those that produce a side-effect, normally by accepting a λ-expression, e.g., `forEach()` [5]. These situations are separately considered, as shown in fig. 3.

3.6.1 Non-scalar Result Producing Terminal Operations

In the case of non-scalar return values, whether the return type maintains ordering is determined by reusing the reflection technique described in section 3.3.2. Specifically, a stream is reflectively derived from an instance of the non-scalar return (run time) type approximations and its characteristics examined. And, from this, whether reduction order matters is determined as follows. If it is impossible for the returned non-scalar type to maintain an element ordering, e.g., it is a



(a) For non-scalar result-producing terminal operations.



(b) For side-effect producing terminal operations.

Fig. 3. Scenarios for whether reduce ordering matters (ROM).

`HashSet`, then, the result ordering cannot make a difference in the program's behavior. If, on the other hand, the returned type *can* maintain an ordering, we conservatively determine that the reduction ordering *does* matter. As before, if there is any inconsistencies between the ordering characteristics of the approximated types, the default is ordered. This is captured in fig. 3a and table 4 under the *non-scalar* rows (column **r. type** is return type). The N/A in column **t. operation** indicates any terminal operation and, in this case, any such operation returning a non-scalar type. The term "collection" refers to any non-scalar type such as those implementing `java.util.Collection` as well as arrays, which are inherently ordered.

3.6.2 Side-effect Producing Terminal Operations

When there is a void return value, as is the case with side-effect producing terminal operations, then, we need to know the order in which the stream elements are "served" to the λ-expression argument producing the side-effect. Currently, void terminal operations that maintain element ordering are also a parameter to our analysis. As with determining SIOs, a more sophisticated analysis would be needed to possibly approximate this characteristic. In the current Java 8 Stream API, there are only two such methods, namely, `forEach()` and `forEachOrdered()`, as seen in fig. 3b and table 4 under the "void" return type rows.

3.6.3 Scalar Result Producing Terminal Operations

The last case is perhaps the most difficult. While discussing whether non-scalar types (e.g., containers) maintain element ordering seems natural, when the reduction is to a scalar type, it is challenging to determine whether or not the element ordering used to produce the resulting value had any influence over it. Another view of the problem involves determining whether or not the operation(s) "building" the result from the stream are associative. Examples of associative operations include numeric addition, minimum, and maximum, and string concatenation [5]. To address this, we divide the problem into determining the associativity of specialized and general reduction operations.

Specialized Reduction Operations: Luckily, the number and associativity property of specialized reduction operations are fixed. As such, the list of specialized operations along with their associativity property is input to the approach. The ROM values compiled by the authors via API documentation examination for the Java 8 Stream API is listed in table 4 under the "scalar" return type rows.

General Reduction Operations: The remaining general reduction operations are `reduce()` and `collect()`. We have already covered the cases where these operations return non-scalar types in the first two rows of table 4. What remains is the cases when these operations return scalar types. Due to the essence of `collect()`, in practice, the result type will most likely fall into the non-scalar category. In fact, `collect()` is a specialization of `reduce()` meant for mutable reductions. Recall from section 2 that such operations collect results in a container such as a collection [5].

The generality of these reduction operations make determining whether ordering matters difficult. For example, even a simple sum reduction can be difficult for an automated approach to analyze. Consider the following code [5] that adds `Widget` weights together using `reduce()`:

```
widgets.stream().reduce(0, (sum, b) -> sum + b.getWeight(), Integer::sum);
```

The first argument is the *identity* element; the second an *accumulator* function, adding a `Widget`'s weight into the accumulated sum. The last argument combines two integer sums by adding them. The question is how, in general, can we tell that this is performing an operation that is associative like summation? In other words, how can we determine that the reducer computation is independent of the order of its inputs? It turns out that this is precisely the *reducer commutativity* problem [20]. Unfortunately, this problem has been shown to be undecidable by Chen *et al.* [20]. While we will consider approximations and/or heuristics as future work, currently, our approach conservatively fails preconditions in this case as indicated by the question marks in table 4. During our experiments detailed in section 4, these failures only accounted for 5%.

Algorithm 1 Convert stream to parallel

1: **for all** $n \in PT$ such that n is a leaf **do**
2: $curr \leftarrow n$
3: **while** $curr \neq$ **NIL do**
4: **if** $Method(curr) =$ `sequential()` **then**
5: Schedule $curr$ for removal.
6: **else if** $Method(curr) =$ `parallel()` **then**
7: **if** $\forall a \in Ancestors(curr)[|Children(a)| > 1]$ **then** {Nodes up from $curr$ to the root have multiple children}
8: Schedule $curr$ for removal. {To avoid redundancy.}
9: **else** {There is a straight-line "chain" from $curr$ to the root}
10: **break** {`parallel()` remains with no further modification.}
11: **end if**
12: **else if** $Method(curr) =$ `stream()` **then** $\{Parent(curr) =$ **NIL**$\}$
13: Schedule $curr$ to be replaced by `parallelStream()`.
14: **else if** $Method(curr) \neq$ `parallelStream()` **then** $\{curr$ is not already parallel$\}$
15: Schedule `parallel()` to be inserted immediately after $curr$.
16: **end if**
17: $curr \leftarrow Parent(curr)$
18: **end while**
19: **end for**
20: Execute all scheduled transformations.

(a) Before transformation.

(b) After conversion to parallel.

Fig. 4. Predecessor tree for listing 4a.

3.7 Transformation

Once a stream has passed preconditions, there may be multiple possible ways to carry out the corresponding transformation. However, not all transformations may be ones that an expert human developer would have chosen. Here, we strive to select transformations that are (i) semantically equivalent to the original, (ii) exposing the most possible parallelism, and (iii) minimal, i.e., requiring the least amount of code changes. This last point reduces invasiveness.

Stream pipelines, i.e., method call chains of intermediate operations ending in a terminal operation, can be complex with chains possibly spanning multiple branches, methods, and even files. Thus, it can be challenging to pinpoint transformation sites in the general case as the call chains are arbitrary. To assist in the transformation, we leverage the $Pred$ relation from definition 4 by building a *predecessor tree* PT, where each node represents a stream instance (call site), an edge between nodes n_i and n_j exists iff $n_j \in Pred(n_i)$, and the root is a node n_0 such that $\forall n \in PT[n_0 \in Origins(n)]$ (see definition 8). A separate tree exists for each for each origin stream in the program. Origin streams are also those that are identified for transformation, thus, the transformation algorithm begins at the root of each tree if a transformation applies to the stream represented by the root.

3.7.1 Execution Mode

Figure 4a depicts a PT for the code snippet in listing 4a, while algorithm 1 depicts the algorithm for transforming a stream to parallel (transformation to sequential is similar). Steps for already parallel streams are shown for completeness. The action at line 10 is valid because intermediate operations like `parallel()` are processed lazily, i.e., when a terminal operation has been issued. As such, "[t]he most recent sequential or parallel mode setting applies to the execution of the entire stream pipeline" [42]. $Ancestors$ is defined on a node n as follows:

$$Ancestors(n) = \begin{cases} \varnothing & \text{if } n = \textbf{NIL} \vee \\ & Parent(n) = \textbf{NIL} \\ Parent(n) \cup Ancestors(Parent(n)) & \text{o.w.} \end{cases}$$

Figure 4b shows the resulting PT after applying algorithm 1 to the PT in fig. 4a, while listing 4b is the transformed code.

3.7.2 Unordering

Unordering a stream, i.e., actions taken for streams passing P3 (table 1) or P4 (table 2), is somewhat similar to altering its execution mode (above) but with some important differences and special considerations. Firstly, although stream execution mode can be changed at the origin stream by replacing the appropriate API call (e.g., `stream()` to `parallelStream()`),

since stream ordering can be dependent on its source collection type, for semantics preservation and to limit refactoring invasiveness, unordering does not occur in a similar way. Instead, unordering transformations always take place via a call to the `unordered()` intermediate operation (e.g., line 33 in listing 2b).

While the unordering transformation can be accomplished similar to algorithm 1 by substituting `parallel()` with `unordered()` and `sequential()` with `sorted()`, there are some special considerations regarding the insertion of `unordered()`. For instance, to maximize efficient parallel computation, such calls are inserted before all stateful intermediate operations (SIOs). This can be seen on line 33 in listing 2b, where `unordered()` is placed before `distinct()`, an SIO.

4 EVALUATION

4.1 Implementation

Our approach was implemented as a publicly available, open source Eclipse IDE [31] plug-in [28] and built upon WALA [32] and SAFE [33]. Eclipse is leveraged for its extensive refactoring support [43] and that it is completely open-source for all Java development. WALA is used for static analyses such as side-effect analysis (ModRef), and SAFE, which depends on WALA, for its typestate analysis. SAFE was altered for programmatic use and "intermediate" typestates (cf. section 3.4.2). For the refactoring portion, Eclipse ASTs with source symbol bindings are used as an intermediate representation (IR), while the static analysis consumes a Static Single Assignment (SSA) [44] form IR.

The OPTIMIZE COMPLEX MUTABLE REDUCTION refactoring is not currently implemented. There are some complications with determining ordering with `Map`s as these are multi-tier data structures. In other words, the `Map` itself is a container (`EntrySet`) having constituent containers (e.g., `List<Widget>` as values; line 53, listing 3). Thus, "digging" into these containers requires a bit more engineering work. Once that is complete, however, the implementation is straight-forward using the approach set-forth in section 3.1.3 and is included in future work.

Per the discussion in section 3.3.2, since stream ordering may depend on the stream's source run time type, to determine stream ordering, our implementation interprocedurally approximates (using a points-to analysis) the run time type of stream sources via type propagation using the iterative fixed-point solver available in WALA. If the type cannot be determined accurately in this way, the type's ordering is defaulted to ordered. Although this may cause missed optimization opportunities, an ordered attribute will not cause our approach to take action, guaranteeing semantics preservation.

Once the possible stream source type(s) has been obtained, reflection is used to determine ordering attributes. First, built-in reflection mechanisms are utilized (i.e., `Class.newInstance()`). However, this can be problematic when either a default (no-arg) constructor does not exist or is not accessible. In such cases, Objenesis [45], a tool normally used for Mock Objects, is used to bypass constructor calls. Ordering is retrieved by obtaining a stream from an instance of type (again, via reflection) and subsequently calling the `characteristics()` method on the newly created stream instance's `Spliterator` [38].

As discussed in section 3.4, our approach utilizes a k-CFA call graph construction algorithm. To make our experiments tractable and to treat client-side API invocations as stream creations (since the focus of this work is on manipulation of client code), we made k an input parameter to our analysis (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. Recall that k amounts to the call string length in which to approximate object instances, thus, $k=1$ would consider constructor calls as object creation locations, while $k=2$ would consider calls to methods calling constructors as ("client") object creation sites. The tool currently uses a heuristic to inform developers when k is too small via a precondition failure. It does so by checking that call strings include at least one client method starting from the constructor call site. Future work involves automatically determining an optimal k, perhaps via stochastic optimization. The call graph used in the typestate analysis is pruned by removing nodes that do not have reaching stream definitions.

4.2 Experimental Evaluation

Our evaluation involved studying 11 open source Java applications and libraries of varying size and domain (table 5). Subjects were also chosen such that they are using Java $>= 8$ and have at least one stream declaration (i.e., a call to a stream API) that is reachable from an entry point (i.e., a candidate stream). Column **KLOC** denotes the thousands of source lines of code, which ranges from $\sim$1K for monads to $\sim$354K for jetty. Column **eps** is the number of entry points. For non-library subjects, all main methods were chosen, otherwise, all unit test methods were chosen as entry points. Column **k** is the maximum k value used (see section 4.2.1). Subjects compiled correctly and had identical unit test (27,955; mostly from jetty) results and compiler warnings before and after the refactoring.

The analysis was executed on an Intel Xeon E5 machine with 16 cores and 30GB RAM and a 25GB maximum heap size. Column **tm (m)** is the running time in minutes, averaging $\sim$6.602 secs/KLOC. An examination of three of the subjects revealed that over 80% of the run time was for the typestate analysis, which is performed by SAFE. This analysis incorporates aliasing information and can be lengthy for larger applications. However, since our approach is automated, it can be executed on a nightly basis or before major releases.

TABLE 5
Experimental results.

subject	KLOC	eps	k	str	rft	P1	P2	P3	t (m)
htm.java	41.14	21	4	34	10	0	10	0	1.85
JacpFX	23.79	195	4	4	3	3	0	0	2.31
jdp*	19.96	25	4	28	15	1	13	1	31.88
jdk8-exp*	3.43	134	4	26	4	0	4	0	0.78
jetty	354.48	106	4	21	7	3	4	0	17.85
jOOQ	154.01	43	4	5	1	0	1	0	12.94
koral	7.13	51	3	6	6	0	6	0	1.06
monads	1.01	47	2	1	1	0	1	0	0.05
retroλ	5.14	1	4	8	6	3	3	0	0.66
streamql	4.01	92	2	22	2	0	2	0	0.72
threeten	27.53	36	2	2	2	0	2	0	0.51
Total	641.65	751	4	157	57	10	46	1	70.60

* jdp is java-design-patterns and jdk8-exp is jdk8-experiments.

TABLE 6
Refactoring failures.

failure	pc	cnt
F1. InconsistentPossibleExecutionModes		1
F2. NoStatefulIntermediateOperations	P5	1
F3. NonDeterminableReductionOrdering		5
F4. NoTerminalOperations		13
F5. CurrentlyNotHandled		16
F6. ReduceOrderingMatters	P3	19
F7. HasSideEffects	P1	4
	P2	41
Total		100

4.2.1 Setting k for the k-CFA

As discussed in section 3.4, our approach takes as input a maximum call string length parameter k, which is used to construct the call graph using nCFA. Each call graph node is associated with a *context*, which, in our case, is the call string. This allows our analysis to approximate stream object creation in the *client* code rather than in the framework, where the stream objects are instantiated. Otherwise, multiple calls to the same API methods that create streams would be considered as creating one new stream.

During our experiments, a default k value of 2 was used. This is the minimum k value that can be used to distinguish client code from framework stream creation. However, depending on which stream framework methods are utilized in a particular project, this value may be insufficient. We detect this situation via a heuristic of examining the call string and determining whether any client code exists. If not, k may be too small.

Setting k constitutes a trade-off. A k that is too small will produce correct results but may miss streams. A larger k may enable the tool to detect and subsequently analyze more streams but may increase run time. Thus, an optimal k value can be project-specific. In our experiments, however, we determined k empirically based on a balance between run time and the ratio between total (syntactically available) streams and candidate streams (i.e., those detected by the typestate analysis). Notwithstanding, in keeping k between 2 and 4 (cf. table 5), good results and reasonable runtime were observed. Thus, it was not difficult to find an "effective" k.

4.2.2 Intelligent Parallelization

Streams are still relatively new, and, as they grow in popularity, we expect to see them used more widely. Nevertheless, we analyzed 157 (origin) streams reachable from entry points (column **str**) across 11 subjects. Of those, we automatically refactored $\sim$36.31% (column **rft** for *refactorable*) despite being highly conservative. These streams are the ones that have passed all preconditions; those not passing preconditions were not transformed (cf. table 6).

Columns **P1–3** are the streams passing the corresponding preconditions (cf. tables 1 and 2). Columns **P4–5** have been omitted as all of their values are 0.[9] The number of transformations can be derived from these columns as preconditions are associated with transformations, amounting to $10+46+(1*2)=58$.

4.2.3 Refactoring Failures

Table 6 categorizes reasons why streams could not be refactored (column **failure**), some of which correspond directly to preconditions (column **pc**). Column **cnt** depicts the count of failures in the respective category and further categorized by

9. Implementation of preconditions 6–13 is part of future work.

TABLE 7
Average run times of JMH benchmarks.

# benchmark	orig (s/op)	refact (s/op)	su
1 shouldRetrieveChildren	0.011 (0.001)	0.002 (0.000)	6.57
2 shouldConstructCar	0.011 (0.001)	0.001 (0.000)	8.22
3 addingShouldResultInFailure	0.014 (0.000)	0.004 (0.000)	3.78
4 deletionShouldBeSuccess	0.013 (0.000)	0.003 (0.000)	3.82
5 addingShouldResultInSuccess	0.027 (0.000)	0.005 (0.000)	5.08
6 deletionShouldBeFailure	0.014 (0.000)	0.004 (0.000)	3.90
7 specification.AppTest.test	12.666 (5.961)	12.258 (1.880)	1.03
8 CoffeeMakingTaskTest.testId	0.681 (0.065)	0.469 (0.009)	1.45
9 PotatoPeelingTaskTest.testId	0.676 (0.062)	0.465 (0.008)	1.45
10 SpatialPoolerLocalInhibition	1.580 (0.168)	1.396 (0.029)	1.13
11 TemporalMemory	0.013 (0.001)	0.006 (0.000)	1.97

precondition, if applicable. Significant reasons streams were not refactorable include λ-expression side-effects (F7, 45%) and that the reduction ordering is preserved by the target collection (19%, c.f. section 2).

Some of the refactoring failures were due to cases currently not handled by our tool (F5), which are rooted in implementation details related to model differences between representations [28]. For example, streams declared inside inner (embedded) classes are problematic as such classes are part of the outer AST but the instruction-based IR is located elsewhere. Though we plan to develop more sophisticated mappings in the future, such failures only accounted for 16%. Other refactoring failures include F4, where stream processing does not end with a terminal operation in all possible executions. This amounts to "dead" code as any queued intermediate operations will never execute. F3 corresponds to the situation described in section 3.6.3, F1 to the situation where execution modes are ambiguous on varying execution paths, and F2 means that the stream is already optimized.

4.2.4 Performance Evaluation

Many factors can influence performance, including dataset size, number of available cores, JVM and/or hardware optimizations, and other environmental activities. Nevertheless, we assess the performance impact of our refactoring. Although this assessment is focused on our specific refactoring and subject projects, in the general case, it has been shown that a similar refactoring done manually has improved performance by 50% on large datasets [46, Ch. 6].

Existing Benchmarks: We assessed the performance impact of our refactoring on the subjects listed in table 5. One of the subjects, htm.java [47], has formal performance tests utilizing a standard performance test harness, namely, the Java Microbenchmark Harness (JMH) [48]. Using such a test harness is important in isolating causes for performance changes to the code changes themselves [46, Ch. 6.1]. As such, subjects with JMH tests will produce the best indicators of performance improvements. Two such tests were included in this subject.

Converted Benchmarks: Although the remainder of the subjects did not include formal performance tests, they did include a rich set of unit tests. For one subject, namely, java-design-patterns [49], we methodically transformed existing JUnit tests that covered the refactored code to proper JMH performance tests. This was accomplished by annotating existing `@Test` methods with `@Benchmark`, i.e., the annotation that specifies that a method is a JMH performance test. We also moved setup code to `@Before` methods, i.e., those that execute before each test, and annotated those with `@Setup`. This ensures that the test setup is not included in the performance assessment. Furthermore, we chose unit tests that did not overly involve I/O (e.g., database access) to minimize variability. In all, nine unit tests were converted to performance tests and made our changes available to the subject developers.

Augmenting Dataset Size: As all tests we designed for continuous integration (CI), they executed on a minimal amount of data. To exploit parallelism, however, we augmented test dataset sizes. For existing benchmarks, this was done under the guidance of the developers [50]. For the converted tests, we chose an N (dataset size) value that is consistent with that of the largest value used by Naftalin [46, Ch. 6]. In this instance, we preserved the original unit test assertions, which all passed. This ensures that, although N has increased, the spirit of the test, which may reflect a real-life scenario, remains intact.

Results: Table 7 reports the average run times of five runs in seconds per operation. Rows 1–9 are for java-design-patterns, while rows 10–11 are for htm.java; benchmark names have been shortened for brevity. Column **orig** is the original program, **refact** is the refactored program, and **su** is the speedup ($runtime_{old}/runtime_{new}$). Values associated with parentheses are averages, while the value in parenthesis is the corresponding standard deviation. The average speedup resulting from our refactoring is 3.49.

4.2.5 Discussion

The findings of Naftalin [46, Ch. 6] using a similar manual refactoring, that our tool was able to refactor 36.31% of candidate streams (table 5), and the results of the JMH tests on the refactored code (table 7) combine to form a reasonable motivation for using our approach in real-world situations. Moreover, this study gives us insight into how streams, and in a broader sense, concurrency, are used, which can be helpful to language designers, tool developers, and researchers.

As mentioned in section 4.2.2, columns **P4–5** in table 5 all have 0 values. Interestingly, this means that no (already) parallel streams were refactored by our tool. Only two candidate streams, stemming from only a single subject, htm.java, were originally parallel. This may indicate that developers are either timid to use parallel streams because of side-effects, for example, or are (manually) unaware of when using parallel streams would improve performance [46]. This further motivates our approach for automated refactoring in this area.

From table 6, F6 and F7 accounted for the largest percentage of failures (64%). For the latter, this may indicate that despite that "many computations where one might be tempted to use side-effects can be more safely and efficiently expressed without side-effects" [5], in practice, this is either not the case or more developer education is necessary to avoid side-effects when using streams. This motivates future work in refactoring stream code to avoid side-effects if possible.

Imprecision is also a possibility as we are bound by the conservativeness of the underlying ModRef analysis provided by WALA. To investigate, we manually examined 45 side-effect failures and found 11 false positives. Several subject developers, on the other hand, confirmed correct refactorings, as discussed in section 4.2.6. As for the former, a manual inspection of these sites may be necessary to confirm that ordering indeed must be preserved. If not, developers can rewrite the code (e.g., changing `forEachOrdered()` to `forEach()`) to exploit more parallelism opportunities.

The average speedup of 1.55 obtained from htm.java (benchmarks 10–11) most likely reflects the parallelism opportunities available in computationally intensive programs [51]. Benchmarks 1–6, which had good speedups as well, also mainly deal with data. Benchmark 7 had the smallest speedup at 1.03. The problem is that the refactored code appears in areas that "will not benefit from parallelism" [52], demonstrating a limitation of our approach that is rooted in its problem scope. Specifically, our tool locates sites where stream client code is safe to refactor and is possibly optimizable based on language semantics but does not assess optimizability based on input size/overhead trade-offs.

4.2.6 Pull Request Study

To assess our approach's usability, we also submitted several pull requests (patches) containing the results of our tool to the subject projects. As of this writing, eight requests were made, with three pending (e.g., [50]) and five rejected. One rejected request [52] is discussed in section 4.2.5. Others (e.g., [49]) confirmed a correct refactoring but only wanted parallel streams when performance is an observed problem.

4.3 Threats to Validity

The subjects may not represent the stream client code usage. To mitigate this, subjects were chosen from diverse domains as well as sizes, as well as those used in previous studies (e.g., [53], [54]). Although java-design-patterns is artificial, it is a reference implementation similar to that of JHotDraw, which has been studied extensively (e.g., [55]).

Entry points may not be correct, which would affect which streams are deemed as candidates, as well as the performance assessment as there is a trade-off between scalability and number of entry points. Since standard entry points were chosen (see section 4.2), representing a super set of practically true entry points. For the performance test (see table 7), the loads may not be representative of real-world usage. However, we conferred with developers regarding this when possible [50]. For the performance tests we manually generated from unit tests, a systematic approach to the generation was taken using the same parameters (N) on both the original and refactored versions.

5 Related Work

Automatic parallelization can occur on several levels, including the compiler [56], [57], run time [58], and source [17]. The general problem of full automatic parallelization by compilers is extremely complex and remains a grand challenge [59]. Many attempt to solve it in only certain contexts, e.g., for divide and conquer [60], recursive functions [61], distributed architectures [62], graphics processing [63], matrix manipulation [64], asking the developer for assistance [65], and speculative strategies [66]. Our approach focuses on MapReduce-style code over native data containers in a shared memory space using a mainstream programming languages, which may be more amenable to parallelization due to more explicit data dependencies [16]. Moreover, our approach can help detect when it is *not* advantageous to run code in parallel, and when unordering streams can possibly improve performance.

Techniques other than ours enhance the performance of streams as well. Hayashi *et al.* [67] develop a supervised machine-learning approach for building performance heuristics for mapping Java applications onto CPU/GPU accelerators via analyzing parallel streams. Ishizaki *et al.* [68] translate λ-expressions in parallel streams into GPU code and automatically generates run time calls that handle low-level operations. While all these approaches aim to improve performance, their input is streams that are *already* parallel. As such, developers must still *manually* identify and transform sequential streams. Nonetheless, these approaches may be used in conjunction with ours.

Harrison [69] develops an interprocedural analysis and automatic parallelization of Scheme programs. While Scheme is a multi-paradigm language, and shared memory is modeled, their transformations are more invasive and imperative-focused, involving such transformations as eliminating recursion and loop fusion. Nicolay *et al.* [70] have a similar aim but are focused on analyzing side-effects, whereas we analyze ordering constraints.

Many approaches use streams for other tasks or enhance streams in some way. Cheon *et al.* [71] use streams for JML specifications. Biboudis *et al.* [1] develop "extensible" pipelines that allow stream APIs to be extended without changing

library code. Other languages, e.g., Scala [2], JavaScript [3], C# [4], also offer streaming APIs. While we focus on Java 8 streams, the concepts set forth here may be applicable to other situations, especially those involving statically-typed languages, and is a topic for future work.

Other approaches refactor programs to either utilize or enhance modern construct usage. Gyori *et al.* [16] refactor Java code to use λ-expressions instead of imperative-style loops. Tsantalis *et al.* [72] transform clones to λ-expressions. Khatchadourian and Masuhara [73] refactor skeletal implementations to default methods. Tip *et al.* [74] use type constraints to refactor class hierarchies, and Gravley and Lakhotia [75] and Khatchadourian [76] refactor programs to use enumerated types.

Typestate has been used to solve many problems. Mishne *et al.* [77] use typestate for code search over partial programs. Garcia *et al.* [78] integrate typestate as a first-class citizen in a programming language. Padovani [79] extends typestate oriented programming (TSOP) for concurrent programming. Other approaches have also used hybrid typestate analyses. Bodden [80], for instance, combines typestate with residual monitors to signal property violations at run time, while Garcia *et al.* [78] also make use of run time checks via gradual typing [79].

6 Conclusion & Future Work

Our automated refactoring approach "intelligently" optimizes Java 8 stream code It automatically deems when it is safe and possibly advantageous to run stream code either sequentially or in parallel and unorder streams. The approach was implemented as an Eclipse plug-in and evaluated on 11 open source programs, where 57 of 157 candidate streams (36.31%) were refactored. A performance analysis indicated an average speedup of 3.49.

In the future, we plan to handle several issues between Eclipse and WALA models and incorporate more kinds of (complex) reductions like those involving maps, as well as look into approximations to combat the problems set forth by Chen *et al.* [20]. Also, we will implement the multi-tier type analysis required for the Optimize Complex Mutable Reduction refactoring. Approximating SIOs may also involve heuristics, e.g., analysis of API documentation. Lastly, we will explore applicability to other streaming frameworks and languages.

Acknowledgments

We would like to thank Atanas Rountev, Eric Bodden, Eran Yahav, Ameya Ketkar, and anonymous reviewers for their insightful feedback and for referring us to related work. Support for this project was provided by PSC-CUNY Award #61793-00 49, jointly funded by The Professional Staff Congress and The City University of New York. Bagherzadeh was supported by Oakland University. Ahmed was supported by Oakland University's GReAT award.

References

[1] A. Biboudis, N. Palladinos, G. Fourtounis, and Y. Smaragdakis, "Streams a la carte: Extensible Pipelines with Object Algebras," in *European Conference on Object-Oriented Programming*, 2015, pp. 591–613. DOI: 10.4230/LIPIcs.ECOOP.2015.591.

[2] EPFL. (2017). Collections–mutable and immutable collections–scala documentation, [Online]. Available: http://scala-lang.org/api/2.12.3/scala/collection/index.html (visited on 08/24/2018).

[3] Refsnes Data. (2015). JavaScript array map() method, [Online]. Available: http://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 (visited on 08/24/2018).

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

[6] J. Lau, "Future of Java 8 language feature support on Android," *Android Developers Blog*, Mar. 14, 2017. [Online]. Available: https://android-developers.googleblog.com/2017/03/future-of-java-8-language-feature.html (visited on 08/24/2018).

[7] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 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 *International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2008, pp. 329–339. DOI: 10.1145/1346281.1346323.

[9] S. Ahmed and M. Bagherzadeh, "What do concurrency developers ask about?: A large-scale study using stack overflow," in *International Symposium on Empirical Software Engineering and Measurement*, 2018, 30:1–30:10. DOI: 10.1145/3239235.3239524.

[10] M. Bagherzadeh and H. Rajan, "Order types: Static reasoning about message races in asynchronous message passing concurrency," in *International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, 2017, pp. 21–30. DOI: 10.1145/3141834.3141837.

[11] 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.

[12] Oracle. (2017). Thread interference, [Online]. Available: http://docs.oracle.com/javase/tutorial/essential/concurrency/interfere.html (visited on 04/16/2018).

[13] R. Warburton, *Java 8 Lambdas: Pragmatic Functional Programming*, 1st ed. Apr. 7, 2014, p. 182, ISBN: 1449370772.

[14] Stack Overflow. (Feb. 2018). Newest 'java-stream' questions, [Online]. Available: http://stackoverflow.com/questions/tagged/java-stream (visited on 03/06/2018).

[15] D. Mazinanian, A. Ketkar, N. Tsantalis, and D. Dig, "Understanding the use of lambda expressions in Java," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, 85:1–85:31, Oct. 2017, ISSN: 2475-1421. DOI: 10.1145/3133909.

[16] A. Gyori, L. Franklin, D. Dig, and J. Lahoda, "Crossing the gap from imperative to functional programming through refactoring," in *Foundations of Software Engineering*, ACM SIGSOFT, 2013, pp. 543–553. DOI: 10.1145/2491411.2491461.

[17] D. Dig, J. Marrero, and M. D. Ernst, "Refactoring sequential java code for concurrency via concurrent libraries," in *International Conference on Software Engineering*, IEEE, 2009, pp. 397–407. DOI: 10.1109/ICSE.2009.5070539.

[18] E. Brodu, S. Frénot, and F. Oblé, "Transforming JavaScript event-loop into a pipeline," in *Symposium on Applied Computing*, 2016, pp. 1906–1911. DOI: 10.1145/2851613.2851745.

[19] C. Radoi, S. J. Fink, R. Rabbah, and M. Sridharan, "Translating imperative code to MapReduce," in *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM, 2014.

[20] Y.-F. Chen, C.-D. Hong, N. Sinha, and B.-Y. Wang, "Commutativity of reducers," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2015, pp. 131–146. DOI: 10.1007/978-3-662-46681-0_9.

[21] T. Xiao, J. Zhang, H. Zhou, Z. Guo, S. McDirmid, W. Lin, W. Chen, and L. Zhou, "Nondeterminism in MapReduce considered harmful? an empirical study on non-commutative aggregators in MapReduce programs," in *ICSE Companion*, 2014, pp. 44–53. DOI: 10.1145/2591062.2591177.

[22] C. Csallner, L. Fegaras, and C. Li, "Testing MapReduce-style programs," in *Foundations of Software Engineering*, 2011, pp. 504–507. DOI: 10.1145/2025113.2025204.

[23] F. Yang, W. Su, H. Zhu, and Q. Li, "Formalizing MapReduce with CSP," in *International Conference and Workshops on the Engineering of Computer-Based Systems*, IEEE, Oxford, UK, Mar. 2010, pp. 358–367. DOI: 10.1109/ECBS.2010.50.

[24] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *Operating Systems Design and Impl.*, 2008, pp. 29–42.

[25] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: Efficient iterative data processing on large clusters," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 285–296, Sep. 2010. DOI: 10.14778/1920841.1920881.

[26] E. Jahani, M. J. Cafarella, and C. Ré, "Automatic optimization for MapReduce programs," *Proc. VLDB Endow.*, vol. 4, no. 6, pp. 385–396, Mar. 2011. DOI: 10.14778/1978665.1978670.

[27] R. Gu, X. Yang, J. Yan, Y. Sun, B. Wang, C. Yuan, and Y. Huang, "SHadoop: Improving MapReduce performance by optimizing job execution mechanism in Hadoop clusters," *Journal of Parallel and Distributed Computing*, vol. 74, no. 3, pp. 2166–2179, 2014. DOI: 10.1016/j.jpdc.2013.10.003.

[28] R. Khatchadourian, Y. Tang, M. Bagherzadeh, and S. Ahmed, "A tool for optimizing Java 8 stream software via automated refactoring," in *International Working Conference on Source Code Analysis and Manipulation*, IEEE, Sep. 2018, pp. 34–39. DOI: 10.1109/SCAM.2018.00011.

[29] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 1, pp. 157–171, Jan. 1986. DOI: 10.1109/tse.1986.6312929.

[30] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay, "Effective typestate verification in the presence of aliasing," *ACM Transactions on Software Engineering and Methodology*, vol. 17, no. 2, pp. 91–934, May 2008. DOI: 10.1145/1348250.1348255.

[31] Eclipse Foundation. (Aug. 2018). Eclipse IDEs, [Online]. Available: http://eclipse.org (visited on 08/24/2018).

[32] WALA Team. (Jun. 2015). T.J. Watson Libraries for Analysis, [Online]. Available: http://wala.sf.net (visited on 01/18/2017).

[33] E. Yahav. (Aug. 2018). SAFE typestate analysis engine, [Online]. Available: http://git.io/vxwBs (visited on 08/24/2018).

[34] Oracle. (2017). HashSet (Java SE 9) & JDK 9, [Online]. Available: http://docs.oracle.com/javase/9/docs/api/java/util/HashSet.html (visited on 04/07/2018).

[35] ——, (2017). Stream (Java Platform SE 9 & JDK 9)–Concatenation, [Online]. Available: http://docs.oracle.com/javase/9/docs/api/java/util/stream/Stream.html#concat-java.util.stream.Stream-java.util.stream.Stream-.

[36] ——, (2018). ConcurrentHashMap (Java SE 10 & JDK 10), [Online]. Available: http://docs.oracle.com/javase/10/docs/api/java/util/concurrent/ConcurrentHashMap.html.

[37] B. Steensgaard, "Points-to analysis in almost linear time," in *Principles of Programming Languages*, 1996, pp. 32–41.

[38] Oracle. (2017). Spliterator (Java SE 9 & JDK 9), [Online]. Available: https://docs.oracle.com/javase/9/docs/api/java/util/Spliterator.html (visited on 08/24/2018).

[39] O. G. Shivers, "Control-flow analysis of higher-order languages of taming lambda," PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1991.

[40] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*, 2nd ed. Secaucus, NJ, USA: Springer-Verlag, 2004, ISBN: 3540654100.

[41] P. Wadler, "Linear types can change the world," in *IFIP TC*, vol. 2, 1990, pp. 347–359.

[42] Oracle. (2017). Java.util.stream (java se 9 & jdk 9)–parallelism, [Online]. Available: https://docs.oracle.com/javase/9/docs/api/java/util/stream/package-summary.html#Parallelism.

[43] 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.

[44] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Symposium on Principles of Programming Languages*, ACM SIGPLAN-SIGACT, 1988, pp. 12–27. DOI: 10.1145/73560.73562.

[45] EasyMock. (2017). Objenesis, [Online]. Available: http://objenesis.org (visited on 08/24/2018).

[46] M. Naftalin, *Mastering Lambdas: Java Programming in a Multicore World*. McGraw-Hill, 2014, ISBN: 0071829628.

[47] Numenta. (2018). Hierarchical Temporal Memory implementation in Java, [Online]. Available: http://git.io/fNbnK.

[48] Oracle. (2018). JMH, [Online]. Available: http://openjdk.java.net/projects/code-tools/jmh (visited on 03/30/2018).

[49] I. Seppälä. (2018). Design patterns implemented in Java, [Online]. Available: http://git.io/v5lko (visited on 08/24/2018).

[50] D. Ray. (Mar. 2018). Pull request #539–numenta/htm.java, [Online]. Available: http://git.io/fAqDq (visited on 03/21/2018).

[51] M. Kumar, "Measuring parallelism in computation-intensive scientific/engineering applications," *IEEE Transactions on Computers*, vol. 37, no. 9, pp. 1088–1098, Sep. 1988, ISSN: 0018-9340. DOI: 10.1109/12.2259.

[52] E. Luontola. (Mar. 2018). Pull request #140–orfjackal/retrolambda, [Online]. Available: http://git.io/fAqHz.

[53] R. Khatchadourian and H. Masuhara, "Proactive empirical assessment of new language feature adoption via automated refactoring: The case of Java 8 default methods," in *International Conference on the Art, Science, and Engineering of Programming*, 2018, 6:1–6:30. DOI: 10.22152/programming-journal.org/2018/2/6.

[54] A. Ketkar, A. Mesbah, D. Mazinanian, D. Dig, and E. Aftandilian, "Type migration in ultra-large-scale codebases," in *International Conference on Software Engineering*, ser. ICSE '19, Montreal, Quebec, Canada: IEEE Press, 2019, pp. 1142–1153. DOI: 10.1109/ICSE.2019.00117.

[55] M. Marin, L. Moonen, and A. van Deursen, "An integrated crosscutting concern migration strategy and its application to JHotDraw," in *International Working Conference on Source Code Analysis and Manipulation*, 2007.

[56] M. Wolfe, "Parallelizing compilers," *ACM Comput. Surv.*, vol. 28, no. 1, pp. 261–262, Mar. 1996. DOI: 10.1145/234313.234417.

[57] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua, "Automatic program parallelization," *Proceedings of the IEEE*, vol. 81, no. 2, pp. 211–243, 1993.

[58] B. Chan and T. S. Abdelrahman, "Run-time support for the automatic parallelization of java programs," *The Journal of Supercomputing*, vol. 28, no. 1, pp. 91–117, 2004.

[59] G. C. Fox, R. D. Williams, and G. C. Messina, *Parallel computing works!* Morgan Kaufmann, 2014.

[60] R. Rugina and M. Rinard, "Automatic parallelization of divide and conquer algorithms," in *ACM SIGPLAN Notices*, ACM, vol. 34, 1999, pp. 72–83.

[61] M. Gupta, S. Mukhopadhyay, and N. Sinha, "Automatic parallelization of recursive procedures," *International Journal of Parallel Programming*, vol. 28, no. 6, pp. 537–562, 2000.

[62] M. Ravishankar, J. Eisenlohr, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic parallelization of a class of irregular loops for distributed memory systems," *ACM Trans. Parallel Comput.*, vol. 1, no. 1, 7:1–7:37, Oct. 2014, ISSN: 2329-4949. DOI: 10.1145/2660251.

[63] A. Leung, O. Lhoták, and G. Lashari, "Automatic parallelization for graphics processing units," in *Principles and Practice of Programming in Java*, ACM, 2009, pp. 91–100.

[64] S. Sato and H. Iwasaki, "Automatic parallelization via matrix multiplication," in *ACM SIGPLAN Notices*, ACM, vol. 46, 2011, pp. 470–479.

[65] H. Vandierendonck, S. Rul, and K. De Bosschere, "The paralax infrastructure: Automatic parallelization with a helping hand," in *International Conference on Parallel Architectures and Compilation Techniques*, IEEE, 2010, pp. 389–399.

[66] J. G. Steffan and T. C. Mowry, "The potential for using thread-level data speculation to facilitate automatic parallelization," in *International Symposium on High-Performance Computer Architecture*, IEEE, 1998, pp. 2–13.

[67] A. Hayashi, K. Ishizaki, G. Koblents, and V. Sarkar, "Machine-learning-based performance heuristics for runtime cpu/gpu selection," in *Principles and Practices of Programming on The Java Platform*, 2015, pp. 27–36. DOI: 10.1145/2807426.2807429.

[68] K. Ishizaki, A. Hayashi, G. Koblents, and V. Sarkar, "Compiling and optimizing Java 8 programs for GPU execution," in *International Conference on Parallel Architecture and Compilation*, 2015, pp. 419–431. DOI: 10.1109/PACT.2015.46.

[69] W. L. Harrison, "The interprocedural analysis and automatic parallelization of scheme programs," *LISP and Symbolic Computation*, vol. 2, no. 3, pp. 179–396, Oct. 1989, ISSN: 1573-0557. DOI: 10.1007/BF01808954.

[70] J. Nicolay, C. de Roover, W. de Meuter, and V. Jonckers, "Automatic parallelization of side-effecting higher-order scheme programs," in *International Working Conference on Source Code Analysis and Manipulation*, 2011, pp. 185–194. DOI: 10.1109/SCAM.2011.13.

[71] Y. Cheon, Z. Cao, and K. Rahad, "Writing JML specifications using Java 8 streams," University of Texas at El Paso, 500 West University Avenue, El Paso, Texas 79968-0518, USA, Tech. Rep. UTEP-CS-16-83, Nov. 2016.

[72] N. Tsantalis, D. Mazinanian, and S. Rostami, "Clone refactoring with lambda expressions," in *International Conference on Software Engineering*, 2017, pp. 60–70. DOI: 10.1109/ICSE.2017.14.

[73] R. Khatchadourian and H. Masuhara, "Automated refactoring of legacy Java software to default methods," in *International Conference on Software Engineering*, May 2017, pp. 82–93. DOI: 10.1109/ICSE.2017.16.

[74] F. Tip, R. M. Fuhrer, A. Kieżun, M. D. Ernst, I. Balaban, and B. De Sutter, "Refactoring using type constraints," *ACM Transactions on Programming Languages and Systems*, vol. 33, no. 3, pp. 91–947, May 2011, ISSN: 0164-0925. DOI: 10.1145/1961204.1961205.

[75] J. M. Gravley and A. Lakhotia, "Identifying enumeration types modeled with symbolic constants," in *Working Conference on Reverse Engineering*, 1996, p. 227, ISBN: 0-8186-7674-4.

[76] R. Khatchadourian, "Automated refactoring of legacy Java software to enumerated types," *Automated Software Engineering*, pp. 1–31, Dec. 2016. DOI: 10.1007/s10515-016-0208-8.

[77] A. Mishne, S. Shoham, and E. Yahav, "Typestate-based semantic code search over partial programs," in *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM, 2012, pp. 997–1016. DOI: 10.1145/2384616.2384689.

[78] R. Garcia, É. Tanter, R. Wolff, and J. Aldrich, "Foundations of typestate-oriented programming," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 4, 12:1–12:44, Oct. 2014, ISSN: 0164-0925. DOI: 10.1145/2629609.

[79] L. Padovani, "Deadlock-free typestate-oriented programming," in *International Conference on the Art, Science, and Engineering of Programming*, AOSA, Apr. 2018. DOI: 10.22152/programming-journal.org/2018/2/15.

[80] E. Bodden, "Efficient hybrid typestate analysis by determining continuation-equivalent states," in *International Conference on Software Engineering*, ACM, 2010, pp. 5–14. DOI: 10.1145/1806799.1806805.