2018

Poster: Towards safe refactoring for intelligent parallelization of Java 8 streams

Yiming Tang
CUNY Graduate Center

Raffi T. Khatchadourian
CUNY Hunter College

Mehdi Bagherzadeh
Oakland University

Syed Ahmed
Oakland University

Poster: Towards Safe Refactoring for Intelligent Parallelization of Java 8 Streams

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

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

Mehdi Bagherzadeh
Oakland University
mbagherzadeh@oakland.edu

Syed Ahmed
Oakland University
sfahmed@oakland.edu

ABSTRACT

The Java 8 Stream API sets forth a promising new programming model that incorporates functional-like, MapReduce-style features into a mainstream programming language. However, using streams correctly and efficiently may involve subtle considerations. In this poster, we present our ongoing work and preliminary results towards an automated refactoring approach that assists developers in writing optimal stream code. The approach, based on ordering and typestate analysis, determines when it is safe and advantageous to convert streams to parallel and optimize parallel streams.

CCS CONCEPTS

• **Software and its engineering → Software evolution**; *Automatic programming*; *Maintaining software*;

KEYWORDS

refactoring, parallelization, typestate analysis, Java 8, streams

ACM Reference Format:
Yiming Tang, Raffi Khatchadourian, Mehdi Bagherzadeh, and Syed Ahmed. 2018. Poster: Towards Safe Refactoring for Intelligent Parallelization of Java 8 Streams. In *ICSE '18 Companion: 40th International Conference on Software Engineering Companion, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3183440.3195098

1 INTRODUCTION

Writing parallel programs can be difficult. MapReduce [1], a popular programming paradigm for writing a certain class of parallel programs, abstracts away much of this complexity by facilitating multi-node processing using succinct functional-like constructs.

Recently, mainstream languages such as Java 8 have adopted functional-style, MapReduce-inspired constructs for parallel *and* sequential data structure processing. In the case of Java, this functionality is embodied by the Stream API, introduced in Java 8 [4].

MapReduce, however, 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. Since streams enable developers to pass behavioral parameters (λ-expressions) to collections for deferred execution, they can be easily executed either sequential or in parallel, making them especially attractive to those not normally familiar with functional programming. But, a burden is now placed on developers to *manually* determine whether running stream code in parallel results in an efficient yet interference-free program. Using streams correctly and efficiently requires many subtle considerations. In fact, ~4K questions on streams have been posted on Stack Overflow (http://stackoverflow.com/questions/tagged/java-stream), of which ~5% remain unanswered.

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 inter-procedural 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 as necessary changes can often be widespread. In this poster, we present our ongoing work and preliminary results in developing a novel automated approach, based on ordering and typestate analysis, that automatically identifies and executes refactoring opportunities where improvements can be made to Java 8 Stream code with the hope of this "intelligent" parallelization resulting more efficient, semantically-equivalent code. Although our preliminary data suggests that the approach is promising, speedup analysis is for future work.

2 MOTIVATION AND INSIGHT

Fig. 1 uses the Java 8 Stream API to process `Widget` collections. Fig. 1a is the original version, while Fig. 1b is the improved (but semantically equivalent) refactored version. A `Collection` of `Widgets` is declared (line 1) that does not maintain element ordering as `HashSet` does not support it. Note that ordering is dependent on the run time type rather than the compile-time type.

A `stream`, a data source view representing an element sequence supporting MapReduce-style operations, of `unorderedWidgets` is created on line 2. This stream's operations execute sequentially. Streams may also be associated with an *encounter order* (element visitation), which can be dependent on the stream's source. In this case, it will be unordered since `HashSets` are unordered.

On line 3, the stream is `sorted()` by the corresponding *intermediate* operation, the result of which is a (possibly) new stream with

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

```
1 Collection<Widget> unorderedWidgets = new HashSet<>();
2 List<Widget> sortedWidgets = unorderedWidgets.stream()
3   .sorted(Comparator.comparing(Widget::getWeight))
4   .collect(Collectors.toList());
5 Collection<Widget> orderedWidgets = new ArrayList<>();
6 Set<Double> distinctWeightSet = orderedWidgets.stream().parallel()
7   .map(Widget::getWeight).distinct()
8   .collect(Collectors.toCollection(TreeSet::new));
```

(a) Stream code snippet prior to refactoring.

```
1 Collection<Widget> unorderedWidgets = new HashSet<>();
2 List<Widget> sortedWidgets = unorderedWidgets.~~stream()~~parallelStream()
3   .sorted(Comparator.comparing(Widget::getWeight))
4   .collect(Collectors.toList());
5 Collection<Widget> orderedWidgets = new ArrayList<>();
6 Set<Double> distinctWeightSet = orderedWidgets.stream().~~parallel()~~
7   .map(Widget::getWeight).distinct()
8   .collect(Collectors.toCollection(TreeSet::new));
```

(b) Improved stream client code via refactoring.

Figure 1: Code snippet of `Widget` collection processing using the Java 8 Steam API.

the encounter order rearranged accordingly. Intermediate operations are deferred until a *terminal* operation is executed (line 4). The execution results in a `List` of `Widgets` sorted by `weight`.

It may be possible to increase performance by running this stream's "pipeline" (operation sequence) in parallel. Fig. 1b, line 2 displays the corresponding refactoring. Note, however, that had the stream been *ordered*, running the pipeline in parallel may actually result in worse performance due to the multiple passes and/or data buffering required by an operation like `sorted()`. Because the stream is *unordered*, the reduction can be done much more efficiently by breaking the problem into sub-problems [4].

In contrast, line 5 instantiates an `ArrayList`, which maintains element ordering. A `Set` of distinct widget weights is created beginning on line 6. Unlike previously, this collection takes place in *parallel* due to the corresponding call. Note though that there is a possible performance degradation here as `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 it is required to know whether it is safe to do so, which can be error-prone if done manually.

Our insight is that it may be possible to determine if it is safe to unorder a stream by analyzing the type of the resulting reduction. In this case, it is a `collection` 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 collecting into such a `Set` 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)`. Unfortunately, since the `TreeSet` will preserve the encounter order, we must keep the stream ordered. To improve performance here, it is advantageous to run this pipeline, perhaps surprisingly, *sequentially*. This transformation takes place in Fig. 1b, line 6.

3 OPTIMIZATION APPROACH

We propose several new refactorings, which include Convert Sequential Stream to Parallel and Optimize Parallel Stream. The first deals with determining if it is advantageous (performance-wise) and safe (e.g., no race conditions, semantics alterations) to transform a sequential stream to one whose pipeline runs in parallel. The second deals with a stream that is *already* set to execute in parallel. The question here is what steps (i.e., transformations) can be taken to improve its performance, whether it is *unordering* the stream or *converting* the stream to execute sequentially.

Our in-progress automated refactoring approach involves using typestate analysis [2,5] to determine stream attributes when a terminal operation is issued. A typestate analysis variant is being developed since operations like `sorted()` return (possibly) new streams derived from the receiver with their attributes altered. To determine collection attributes, e.g., element ordering, a combination of points-to analysis and reflection is used, with the former to interprocedurally approximate return value run time types, and the latter to instatiated the class to obtain ordering data. This is viable as collections do not normally alter ordering during object lifetime.

Our generalized typestate analysis works by tracking the state of stream instances using the two labeled transition systems (LTSs), one of which tracks execution mode and the other ordering. Stream typestate is then merged with that of "intermediate" streams to obtain the final typestate at the terminal operation since that is when all of the (queued) intermediate operations will execute.

4 PRELIMINARY RESULTS

Our refactoring approach has been implemented as an Eclipse (http://eclipse.org) plug-in and built upon WALA (http://wala.sf.net). A preliminary experiment on 11 open source Java projects demonstrates that our tool promisingly deems ~33% of 128 total streams refactorable, i.e., those passing our preconditions. Determining whether the refactoring results in more optimal code is part of our future work. Major reasons that streams are not refactorable include λ-expressions side-effects (~45%) and that the reduction ordering is preserved by the target collection (~22%, c.f. §2). Although speedup analysis is for future work, it has been shown that a similar manual refactoring can improve performance [3, Ch. 6].

5 CONCLUSION & FUTURE WORK

We have outlined our work-in-progress towards an automated refactoring approach that "intelligently" optimizes Java 8 stream code. The approach, based on ordering and typestate analysis, automatically deems when it is safe and advantageous to run stream code either sequentially or in parallel. In the future, we will expand our corpus and formulate a transformation algorithm.

REFERENCES

[1] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113.

[2] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2008. Effective Typestate Verification in the Presence of Aliasing. *ACM Transactions on Software Engineering and Methodology* 17, 2, Article 9 (2008), 34 pages.

[3] Maurice Naftalin. 2014. *Mastering Lambdas: Java Programming in a Multicore World* (1st ed.). McGraw-Hill Education Group.

[4] Oracle Corporation. 2016. java.util.stream (Java Platform SE 8)–Classes to support functional-style operations on streams of elements. (2016). http://bit.ly/1jqPUXi

[5] Robert E Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE* SE-12, 1 (Jan. 1986), 157–171.