Paper Critique
Eliminating abstraction overhead of Java stream pipelines using ahead-of-time program optimization

Anders Møller and Oskar Haarklou Veileborg

Kewal Shah (662808782)
kshah205@uic.edu
March 31, 2021

Introduction

Java Streams enables the developers to work on streams of elements–like collections–using a pipeline of functional-style operations [1]. A few advantages of using streams are lazy evaluation, easy switch between sequential and parallel execution, ease of programming, and improved code readability. These advantages come at the cost of execution overheads (CPU and memory) affecting the performance of the code.

A code written using java streams would make a greater number of virtual calls as compared to its corresponding traditional imperative style code. During run-time, a stream pipeline with N elements and K depth would make a maximum of $N \times K$ virtual calls just to push the elements through the stream pipeline [2]. Depending on the operations in the pipeline, it would need more virtual calls to execute those operations. Making virtual calls is an expensive operation as it requires the JVM to create a new *Frame* for the method to be called and push it to the *JVM Stack* along with many other house-keeping operations [3].

Møller and Veileborg have developed a technique to address the performance issues related to the execution of Java streams. Their technique enables the developers to write concise code using stream pipelines along with the efficiency of low-level imperative code at run-time. The authors achieve this by transforming and optimizing the bytecode of the stream pipelines into the bytecode of corresponding low-level imperative code using Ahead of Time (AOT) compilation.

Related Work

A lot of work exists in compiler optimizations and specifically in bytecode-to-bytecode optimizations. Two very popular compiler optimization techniques are inlining and stack allocations. Budimlic and Kennedy have developed a bytecode-to-bytecode transformation technique to address the issue of virtual method calls by inlining all the data and code [4] [5]. Choi and others have developed a technique for stack allocation using escape analysis [6].

The authors of this paper have developed their technique based on such similar works. They are the first to implement a technique that improves the Java streams' performance by bytecode-to-bytecode transformations. They are also the first to investigate how AOT compilation can overcome Java stream's inherent execution overhead.

Khatchadourian and others have developed a technique to improve the execution performance by identifying whether it should convert a given sequential stream into a parallel stream and vice versa. Their solution just addresses the problem of whether it would be helpful to execute a stream sequentially or parallelly, it does not improve the performance of the stream's execution itself.

Various tools exist for other languages like C# and Scala to convert their declarative code into optimized imperative code. These existing approaches are hard to implement in Java because Java is not inherently a functional programming language, and it does not provide advanced meta-programming capabilities.

Claims

The authors claim that their technique

- "does not require adding new language features or modifications of the existing code".
- "does not depend on API-specific knowledge".
- "is easy to integrate into existing program development processes".
- can generate bytecode for stream pipelines which are as efficient as hand-written imperative code.

Proposed Solution

The authors have developed a four steps process to convert and optimize the stream pipeline's bytecode to its equivalent imperative code. This entire process takes place as a part of the AOT compilation. This four steps process is as follows.

1. Pre-Analysis

   The goal of this step is to identify the stream pipelines by simply looking for all variables of type *Stream* in the bytecode. The authors have not elaborated on how to look for *Stream* type variables in the bytecode. Where *Stream* objects are not created in the class but passed as parameters, return values, or a member variable of an object – the tool does not perform optimization of such stream pipelines.

   Next, this step also involves identifying the concrete type of the stream source. The authors have proposed using an off-the-shelf pointer analysis tool to identify the possible concrete type for the stream source. The authors used a type of pointer analysis called demand-driven analysis as they need to evaluate only relevant parts of the code. They have not mentioned which demand-driven analysis tool was finally used in the experiments.

   The results include the code segments corresponding to stream pipelines along with basic analysis of the stream source, which is then sent to the next step.

2. Interprocedural analysis

   This step performs a detailed analysis of the stream pipeline identified in the previous step. The goal of this step is to analyze each function call in the stream pipeline and understand how this instruction affects the values of objects the stream pipeline is operating on. Understanding the flow of control and data in the stream pipeline helps in the next steps of generating the bytecode for its equivalent imperative code.

   To achieve this, the authors developed a technique that is inspired by related previously existing techniques. The authors' technique comprises maintaining two graphs, one for modeling the relationships between all the concrete and abstract types, and the other for modeling the relationships between the objects. Using the elements from these two graphs, an abstract state for each object with its type and abstract value is maintained. This analysis of abstract states helps in resolving each function call unambiguously.

The authors have also clearly stated that in situations where the “object created in the analyzed part of the code may escape that part of the code”, “call target cannot be resolved with sufficient precision” and “recursion causing the analysis to diverge” would cause the analysis to fail and abort the optimization process.

3. Inlining and Stack allocation

   Making virtual method calls is expensive as discussed above, thus method inlining should help in improving the performance of a program. Using the abstract state information generated by the previous step, each callee is uniquely resolved and an attempt is made to inline the method along with its input parameters. The variables to be used in the stream pipeline are also allocated on the stack instead of the heap memory so that they can be locally accessed.
   Inlining and stack allocation are two very popular compiler optimization techniques that the authors have used in their approach to optimize the stream pipelines.
   The output of this step is an imperative code equivalent to the stream pipeline.

4. Cleanup

   The imperative code generated by the previous step contains lots of redundant bytecode instructions. These instructions are identified by techniques like strongly live variable analysis, nullness analysis, reachability analysis, etc., and then removed to make the code more concise.

Results

The authors evaluate their technique by answering two research questions. The first research question addresses the performance of the code generated through their technique compared to hand-optimized code. They performed this analysis on a set of 11 micro-benchmarks across a set of 4 different JVMs. Through this analysis, the authors concluded that the code generated through their technique is comparable to the baseline hand-optimized code.

The second research question put forth by the authors addresses the extent to which their technique can optimize real-world stream pipeline examples. The authors selected 28 different java projects that use streams from the RepoReapers dataset. They found out that their technique can optimize 77% of the streams found in these 28 java projects.

Critique

The concept of a technique to optimize the bytecode of a Java stream pipeline using AOT compilation is quite remarkable. Considering that it is the first attempt for a technique to transform and optimize the declarative Java stream’s bytecode to its equivalent imperative bytecode, it is quite sophisticated. The paper explains each aspect of the technique in detail with simple examples which makes the paper easy to comprehend.

Being a bytecode-to-bytecode optimization technique, the authors’ claim that their technique does not require additional language features or modifications of the existing code stands true. The authors also claim that it is easy to integrate their tool into the existing program development process, but the authors have specified nothing about how to use their tool. Thus, without this knowledge, it is difficult to say whether is it easy to integrate into the existing program development process or not.

The authors have provided some good information regarding the performance of their technique. They tested the impact of their technique's optimization as compared to the baseline on a set of 11 micro-benchmarks; I think that this set of 11 micro-benchmarks represents only a few types of operations that can be performed using streams. A suggestion from my side to the authors would be to explain how applicable the results of these micro-benchmarks are to the real-world. The authors also tested their technique on a set of real-world java projects which use streams, but in this case, they just tested how many streams got optimized and not the impact of optimization as compared to baseline. I think analyzing the impact of optimization on streams from real-world java projects would have given a concrete understanding of the technique's efficacy.

I think that the authors should have also tested their technique using varying amounts of data because sometimes streams perform differently for a different amount of data. Comparing the impact of optimization against varying amounts of data would have given a better understanding of the situations where this technique would provide the best results. I think the paper should also mention the specification of the machine/compute-resources used to test the technique so that others can compare their results with the ones provided in the paper.

The technique mentioned in the paper gets used during the AOT compilation process. As of JDK 16, the AOT feature is being removed from both Oracle JDK and Open JDK [7]. Currently, it is unclear whether these features would be added to the future version of JDK or not. Thus, anyone who plans to upgrade to JDK 16 could not use this technique in their projects.

Overall, the authors have provided detailed research which is comprehensive and well-articulated. They have also mentioned cases where their technique would not work and have taken up these cases as a part of future work. Ultimately, the idea presented in this paper is practical and novel; it provides the foundation for future researchers in this domain to work on.

References

[1] Oracle, "Package java.util.stream Description," [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html.

[2] O. Kiselyov, A. Biboudis, N. Palladinos and Y. Smaragdakis, "Stream fusion, to completeness," *In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*, pp. 285-289, 2017.

[3] Oracle, "The Structure of the Java Virtual Machine," [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html.

[4] Z. Budimlic and K. Kennedy, "Optimizing Java: theory and practice," in *Concurrency and Computation: Practice and Experience*, 1998.

[5] Z. Budimlic and K. Kennedy, "Static Interprocedural Optimizations in Java," 1998.

[6] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar and S. Midkiff, "Escape analysis for Java.," in *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '99)*, 1999.

[7] OpenJDK, "JDK Bug System," 2020. [Online]. Available: https://bugs.openjdk.java.net/browse/JDK-8255616.

[8] A. Møller and O. . H. Veileborg, "Eliminating abstraction overhead of Java stream pipelines using ahead-of-time program optimization," *Proceedings of the ACM on Programming Languages*, p. 29, 2020.

[9] R. Khatchadourian, Y. Tang, M. Bagherzadeh and S. Ahmed, "Safe automated refactoring for intelligent parallelization of Java 8 streams," in *In Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*, 2019.