# ValFinder: Finding Hidden Value-Type Classes

Arjun H Kumar
s21008@students.iitmandi.ac.in
Indian Institute of Technology Mandi, India

Manas Thakur
manas@iitmandi.ac.in
Indian Institute of Technology Mandi, India

## ABSTRACT

In modern object-oriented programming languages, object identity enables fundamental features such as field mutation and synchronization. However, it also significantly affects the performance. In particular, each distinct field access requires a memory load of the corresponding object followed by an indirection. Several compiler analyses and optimizations such as escape analysis and field scalarization can eliminate these costs in specific scenarios; however, such optimizations are usually limited in their scope and applicability. Languages like Java allow for optimizing the access cost for objects of certain "primitive" types; however, OO programs often contain additional user-defined types whose objects do not depend on an identity that is separate from their "value". An important development in this space has been the Project Valhalla [2], which aims to improve the performance profile of conventional objects in Java, and make it comparable to the performance of primitive types. Valhalla introduces the notion of value types [4], which essentially empowers objects to be identity-less. In order to facilitate an improved performance for such objects, an important optimization that can be performed by a value-types supporting Java Virtual Machine (JVM) is *object inlining* [1] or *flattening*.

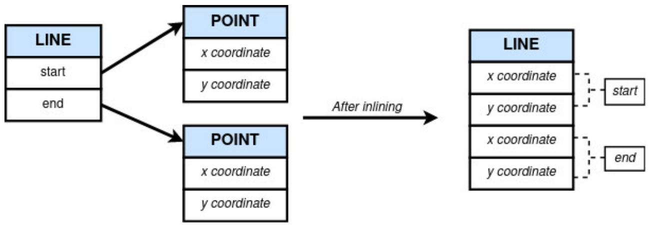

**Figure 1: Example of object inlining.**

Inlining an object modifies the reference to a field inside a class object such that the object pointed to by the field is encoded directly inside the object of the class containing the same (the "container" object). An inlined object avoids overheads from object headers, memory indirections, as well as heap allocation, and exhibits improved cache locality. Figure 1 illustrates this idea. Here, on the right-hand side, two *Point* objects, pointed to by the fields *start* and *end*, are inlined into a container object of type *Line*. As a result, accesses to the objects pointed to by *start* and *end* can be done directly from the container object of type *Line*.

Figure 2 further illustrates a realistic use-case of value types. The method `isLimitExceeded` in class `A` is a representative transaction limit-checker executed by transaction processing engines. It contains a for loop iterating through all transactions on a current day, and a condition check that refers to a field of an object that stores

```java
class TransnInfo { //Container Class
   long transactionID;
   AccountDetails debAccnt, credAccnt;
   double amount; }
class AccountDetails { //Primitive Value Class
   long accID; }
class A {
   final double LIMIT = 5000000;
   public boolean isLimitExceeded(long accID) {
      double amountSpent = 0;
      List<TransnInfo> tInfoLst = getCompletedTs();
      for (TransnInfo trnsInfo : tInfoLst) {
         if (trnsInfo.debAccnt.accID == accID)
            amountSpent += trnsInfo.amount; }
      return amountSpent >= this.LIMIT; } }
```

**Figure 2: Use case scenario of object inlining.**

account details of the source account of the corresponding transaction. Each access to the field `accID` of an `AccountDetails` object would effectively require two memory indirections, one for the `TransnInfo` object and another for the field `debAccnt`. However, if we inline the fields of `AccountDetails` objects into the container `TransnInfo` object, the field access becomes much cheaper and two fewer objects get allocated on the heap for every `TransnInfo` object. Additionally, the likelihood of cache misses reduces for consequent accesses. In a real world scenario, millions of such transactions might happen on a daily basis, which indicates the magnitude of the impacts such an elegant optimization can bring about.

Reckon that given a program, it may not be straightforward to identify the classes that could be marked as value types. Similarly, it may neither be possible to inline all the objects of value types (e.g. atomicity checks for nullness) nor may it be beneficial to do so (e.g. increase in object size). Hence, the current implementation in Eclipse OpenJ9 [3] first filters value types by putting additional restrictions to identify *primitive types*, and then uses a fixed flattening size parameter *(ValueTypeFlatteningThreshold)* to determine if the objects of primitive types can be inlined into suitable container objects by the JVM instance. In this talk, we describe our tool ValFinder, which identifies and marks value and primitive types in real-world Java programs.

ValFinder uses the Soot framework [7] to perform static analysis of Java Bytecode, and identifies classes that can be marked as `value` and `primitive`. It then uses JavaParser [5] to transform Java source code accordingly. Project Valhalla is expected to be launched as part of Java 20. In the meanwhile, we intend to extend ValFinder to identify (objects of) value and primitive types that could offer significant benefits due to object inlining, and to implement the same in OpenJ9 as an alternate, more flexible object flattening strategy. We hope to finally propose a static+dynamic [6] approach that allows JVMs to best utilize the potential benefits offered by the notion of immutability brought in by value types.

*AORCPT '22, November 15-17, 2022, Toronto, Canada*
2022. ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

## REFERENCES

[1] Julian Dolby and Andrew Chien. 2000. An Automatic Object Inlining Optimization and Its Evaluation. *SIGPLAN Not.* 35, 5 (may 2000), 345–357. https://doi.org/10.1145/358438.349344

[2] Brian Goetz. 2021. State of Valhalla. https://openjdk.org/projects/valhalla/design-notes/state-of-valhalla/01-background

[3] Dan Heidinga and Sue Chaplain. 2018. Eclipse OpenJ9; not just any Java Virtual Machine. https://www.eclipse.org/community/eclipse_newsletter/2018/april/openj9.php

[4] Dan Smith. 2021. JEP draft: Value Objects (Preview). https://openjdk.org/jeps/8277163

[5] JavaParser Development Team. 2021. Javaparser. https://javaparser.org/

[6] Manas Thakur and V. Krishna Nandivada. 2019. PYE: A Framework for Precise-Yet-Efficient Just-In-Time Analyses for Java Programs. *ACM Transactions on Programming Languages and Systems* 41, 3, Article 16 (July 2019), 37 pages. https://doi.org/10.1145/3337794

[7] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java Bytecode Optimization Framework. In *CASCON First Decade High Impact Papers* (Toronto, Ontario, Canada) *(CASCON '10)*. IBM Corp., USA, 214–224. https://doi.org/10.1145/1925805.1925818