INDIAN INSTITUTE OF TECHNOLOGY MANDI
HIMACHAL PRADESH, INDIA - 175075
www.iitmandi.ac.in

PROGRESS REPORT FOR THE ACADEMIC YEAR 2023

Scholar's Name: ARJUN H KUMAR	**Roll No:** S21008
School: SCEE	**Date of Registration:** 9th August 2021
Semester: IV	**CGPA:** 8.57
Date of Last Presentation: 29th July 2022	**Date of Current Presentation:** x July 2023

1 Research Objectives

Using program analysis-

1. Identify various access patterns under which value-type objects should be flattened in their respective containers.
2. Build an appropriate flattening strategy for such objects in Eclipse OpenJ9 VM.
3. Improve Java applications by using a static + JIT analysis technique based on the developed flattening strategy.
4. Explore prospective optimizations that can be enabled in JVM due to the introduction of identity-less objects.

2 Introduction

In modern object-oriented programming languages, object identity enables fundamental features such as field mutation and synchronization. However it also affects the performance and memory footprint of an application significantly. Each distinct field access requires a memory load of the corresponding object followed by an indirection to access the field object which is an additional overhead. Moreover every object is heap-allocated, and each heap allocated object has headers of one or more words. An application may involve several thousand objects which contain such a predefined header. On the contrary, if an object is identity-less then the predefined header is no longer required and can contribute to reducing the memory footprint of the application. Several compiler analyses and optimizations such as escape analysis and field scalarization can eliminate these costs for the objects that maintain identity; 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 object-oriented programs often consist of 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 Project Valhalla [Goetz, 2021], 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 [Smith, 2021], which essentially empowers objects to be identity-less. 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 or flattening [Lhoták and Hendren, 2002].

2.1 Object Flattening

Flattening 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. This creates a compact memory representation for the object containing the field. A flattened field inside an object avoids overheads from object headers, memory indirections, heap allocation, and exhibits improved cache locality. The object which contains the encoded field object is termed as the "container" object.

Figure 1 illustrates the idea of flattening. 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*. Moreover, the allocation costs and memory occupied for two *Point* objects are saved along with a more coherent memory representation for *Line* object.

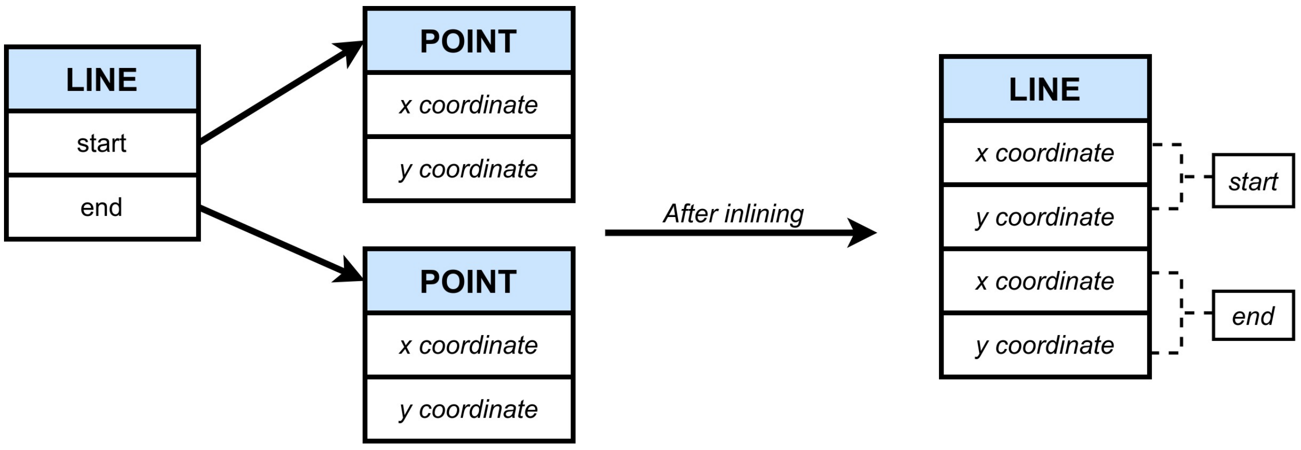

Figure 1: Flattening of objects

The flattening optimization needs to be perfomed carefully considering all the effects it can cause. For instance, flattening a field instance which is of large size or flattening too many objects into a container may lead to bloating. Bloating can impact the cache memory adversely and may lead to potential loss of benefits gained by flattening. Another scenario is the case of object comparison in which a container object may suffer degardation in performance resulting from a high number of comparisons similar to the comparison operation performed by the native *String* class.

Such impactful factors may be determined by analyzing the source code statically in some cases, and in others may require information used during the execution of the program. Experimenting with programs that involve value-type objects and flattening optimization can help to discover more such impactful factors and access patterns to be considered for this optimization.

Currently, in JVMs, a fixed parameter-based criterion is used to decide the flattening of objects. Our research targets a combination of static + dynamic analyses, an idea from the PYE framework [Thakur and Nandivada, 2019] in order to selectively decide potential candidates for object flattening in the Eclipse OpenJ9 VM. We also intend to expand the optimizations possible in this VM in the context of value types.

3 Work Done and Target Set for Last Year

3.1 Study on implementation of object inlining inside the OpenJ9 VM

It may neither be possible to inline all the objects of value-type classes (due to atomicity checks for nullness) nor may it be beneficial to do so. Similar to the function of inlining optimization, inlining all potential candidates may lead to incoherently sized objects. Imprudent increase in the size of objects may cause negative impacts on the memory efficiency of the corresponding application and may result in loss of all potential benefits obtained from object flattening.

To ensure sanity and efficiency, the current implementation in Eclipse OpenJ9 [Heidinga and Chaplain, 2018] first filters value-type classes by putting additional restrictions to identify primitive value-type classes, and then using a parameter-based strategy to limit the object size and decide if a field can be inlined into its corresponding container object. For this, a JVM parameter (-XX:ValueTypeFlatteningThreshold) has been introduced as a limiting threshold. The parameter specifies the maximum size (in bytes) of a flattenable object. Primitive value-type objects [Smith, 2020] that exceed this threshold size do not get optimized by flattening itself into its container instances. The instance size is computed dynamically considering all fields of an object and its predefined header size. The JVM parameter (-XX:ValueTypeFlatteningThreshold) is provided as input during runtime. By default the OpenJ9 VM considers this value to be 0 unless defined explicitly. The workflow for the parameter-based strategy is described in Figure 2.

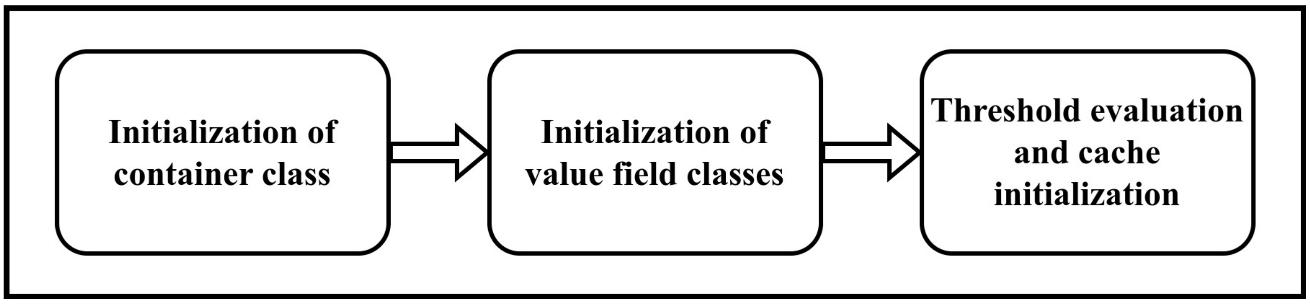
Initialization of container class
Initialization of value field classes
Threshold evaluation and cache initialization

Figure 2: Flattening of objects

The bytecode '*new*' resulting from the initialization of an object will trigger the loading of a class unless already done. During the initialization of a class all the flattenable field value classes of the corresponding class are also loaded. This is done by analyzing the type-class descriptor. Flattenable value classes have a unique type-class descriptor which is specifically prefixed with the character '*Q*'. Once the flattenable field class has been loaded a threshold check is done. The threshold condition is described in equation 1.

$$InstanceSize \leq VM -> FlatteningThreshold \quad (1)$$

After the class loading of the value-type class is completed, the flattened field cache for the corresponding container class is made an entry which describes the shape and modifiers of the flattened field. This field cache is used while instantiating each instance of the class to track which fields are flattened.

3.2 Identification of value-type classes in programs

Value-types and flattening though already present in languages like *C#* is a novel concept introduced for Java and is expected to be released as part of JDK-21. Inorder to experiment with such a cutting-edge feature codebases/ benchmarks needed to be explicitely created. As per the JEP specification [Smith, 2020] detailing the Flattened Heap Layouts for Value Objects, there exists a clear set of restrictions abiding which a class can be flattened into their respective container class object. A tool that can analyze existing codebases to find such classes which abide by the neccessary restrictions would help to create an developemental space for experimenting value-type objects and flattening. Motivated by this problem; a tool "ValFinder" was developed.

ValFinder uses the Soot framework [Vallée-Rai et al., 2010] to perform static analysis of Java Bytecode, and identifies classes that can be marked as value and primitive. It then uses JavaParser [Team, 2021] to transform Java source code accordingly. ValFinder consists of two modules as described in the Figure 3:

1. Analysis module
2. Transformation module

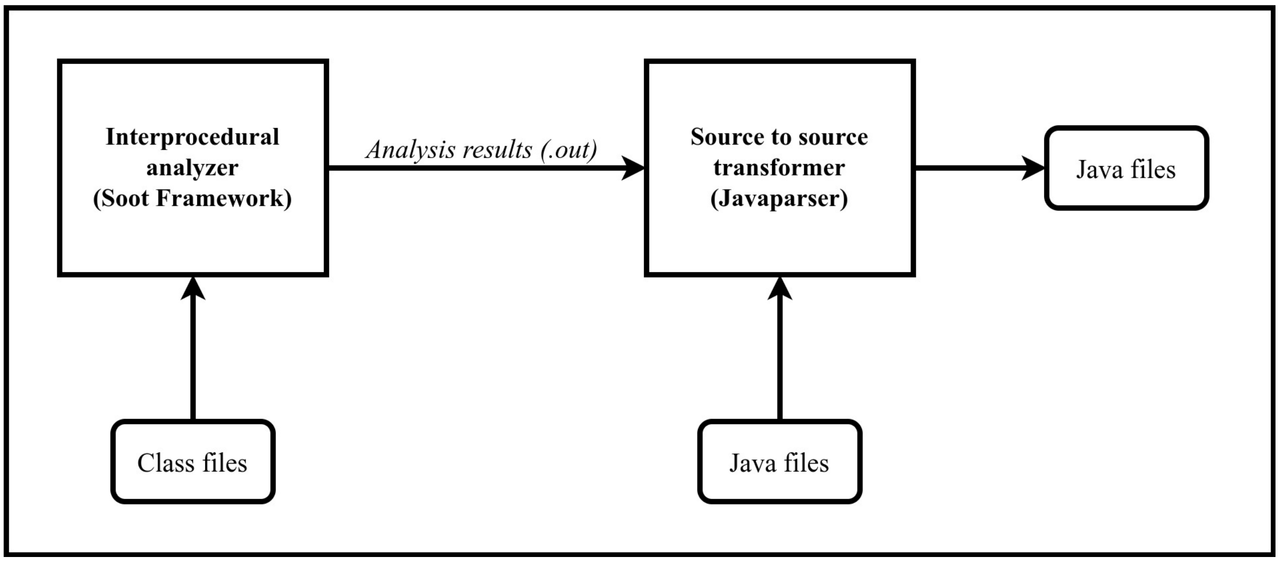
Interprocedural analyzer (Soot Framework)
Analysis results (.out)
Source to source transformer (Javaparser)
Java files
Class files
Java files

Figure 3: Overview of ValFinder

ValFinder starts by taking in the class files of a program that is to be analyzed as input and initiates performing an interprocedural analysis from the analysis module. In this module the dependencies between various classes and the value-type satisfiability restrictions are evaluated. The output of this module is a set of value and primitive classes identified from the input set. This set of identified classes are fed to the transformation module along with source code of the corresponding class files which were fed as input during the analysis phase. In the transformation module we use a modified JavaParser which supports the semantics of value types classes.

Using the modified JavaParser we repair the source code of the classes identified during the analysis phase. This is achieved by iterating over the AST generated for the source code and subsequently modifying the node information. These changes are neccessary so that the identified classes adhere to the syntactic specifications of value/primitive classes. The output of this phase is hence the refactored source code which can potentially be benefitted through object flattening. This completes a pipeline for automatic source to source conversion of eligible non-value type classes to value-type classes. ValFinder was further enhanced with various other auxilliary analysis to highlight the impact of restrictions that cause non-compliance among classes.

3.3 Granularity shift

As per the JEP [Smith, 2020] specification and current implementation in JVMs the instance size of a value-type object does not change; hence when a value-class is loaded for the first time a flag is used to mark the class as flattenable or not. Every instance field of a flattenable marked value-class type will get flattened irrespective of other factors. This level of granularity provides us with less control for flattening. There are instances of a value-type field which we would prefer to inline and vice versa. To enable the same we modified the current implementation of Eclipse OpenJ9 VM to support field level granularity over class level granularity.

For enabling field level granularity; we need to pass information to the VM about which fields to be considered for flattening. This information is provided as input to the Eclipse OpenJ9 VM in the form of *.out* file. This file contains a list field metadata (field signature and corresponding container class signature); this information is sufficient to uniquely identify fields inside the VM.

For controlling the granularity level in VM two new VM arguments has been introduced (-XX:+UseStaticResults and -XX:-UseStaticResults). -XX:+UseStaticResults flag instructs the VM to use field level granularity and only consider the fields mentioned in the input *.out* file for flattening. -XX:-UseStaticResults flag reverts the granularity to class level where all eligible fields are flattened.

During the VM initialization phase the VM thread checks the granularity level based on the input parameters provided. If field level granularity is specified then we read the '.out' file and store the corresponding field information in a global VM datastructure. During the class initialization phase as described in section 3.1; we match the class signature of the acting container class and iterate it's fields to match the field signature from the global datastructure. Once the field has been identified as a potential flattening candidate we proceed with the threshold condition check and class cache initialization similar to native implementation described in section 3.1.

Along with a finer granularity level; the modifications to the Eclipse OpenJ9 VM aligns with our approach of using a static + dynamic scheme to selectively flatten objects. By tuning the input *.out* file provided to the VM we can decide which fields are to be considered for flattening and vice versa. This mechanism helps us to statically analyze code and pass insightful results to the JVM ecosystem where flattening decision is ultimately made.

4 Planned Work for the Next Year

1. Implementing a static + dynamic analysis to compute the number of distinct cache loads possible between two inlined field loads.
2. Completing an end to end pipeline retrofitting all the analyses together.
3. Evaluating the strategy over a set of standard benchmarks for JVM.
4. Preparing a manuscript describing the complete work.

5 Workshops/Conferences Attended

1. International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion), Virtual, December 5th-10th, 2022.
2. 16th Innovations in Software Engineering Conference (ISEC), IIIT Allahabad, India, February 23rd-25th, 2023.
3. Software Engineering Research in India (SERI) Update Meeting, Goa University, India, June 2nd-3rd, 2023

6 Papers Published/Communicated and Other Achievements:

1. Arjun Harikumar and Manas Thakur. "ValFinder: Finding Hidden Value-Type Classes". 6th Workshop on Advances in Open Runtimes and Cloud Performance Technologies (AOR-CPT), part of IBM WeaveSphere, Toronto, Canada, November 16th, 2022.

References

Brian Goetz. State of Valhalla, December 2021. URL https://openjdk.org/projects/valhalla/design-notes/state-of-valhalla/01-background.

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

Ondrej Lhoták and Laurie Hendren. Run-time evaluation of opportunities for object inlining in java. In *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande*, JGI '02, page 175–184, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581135998. doi: 10.1145/583810.583830. URL https://doi.org/10.1145/583810.583830.

Dan Smith. JEP 401: Primitive Classes (Preview), August 2020. URL https://openjdk.org/jeps/401.

Dan Smith. JEP draft: Value Objects (Preview), November 2021. URL https://openjdk.org/jeps/8277163.

JavaParser Development Team. Javaparser, 2021. URL https://javaparser.org/.

Manas Thakur and V. Krishna Nandivada. PYE: A Framework for Precise-Yet-Efficient Just-In-Time Analyses for Java Programs. *ACM Transactions on Programming Languages and Systems*, 41(3), July 2019. ISSN 0164-0925. doi: 10.1145/3337794. URL https://doi.org/10.1145/3337794.

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

REPORT BY APC/DC COMMITTEE

1. **Has the student met the targets set for last year?**

   (a) Mention the Achieved Targets:

   -
   -

   (b) If not what are the major reasons?

   N/A

2. **Is there a reasonable target set for next year? Give detailed plan.**

3. **What is the perception of the student and guide(s) about the fraction of thesis work completed?**

   N/A

4. **What is the approximate time scale for thesis submission (only for students in their 5th year or above for Ph.D. and 3rd year and above for M.S. students).**

   N/A

5. **Any other observations of the committee.**

INDIAN INSTITUTE OF TECHNOLOGY MANDI
HIMACHAL PRADESH, INDIA - 175075
www.iitmandi.ac.in

Recommendation of APC/DC *(Tick Appropriately)*

1. (a) Continuation of Registration is **Recommended/ Not Recommended.**
   (b) Continuation of Scholarship/Research Assistantship **Recommended/ Not Recommended.**
   (c) Enhancement of Scholarship from JRF to SRF is **Recommended/ Not Recommended** (only after Two Year of Registration).
2. **Source of Funding/Scholarship:**
3. **OVERALL PERFORMANCE: Very Good/Good/Satisfactory/Unsatisfactory**
4. **Any Other Recommendation/Comments** *(Attach separate sheet).*

COMMITTEE MEMBERS

S. No.	Faculty Name	School/Department	Signature	Remarks
1.	Dr. A.D. Dileep	SCEE		
2.	Dr. Aditya Nigam	SCEE		
3.	Dr. Gaurav Bhutani	SCEE		
4.	Dr. Manas Thakur	SCEE		
5.	Dr. Varunkumar Jayapaul	SCEE		

Signature of the Supervisor
Date:

School Chairperson
Date:

Associate Dean (Research)
Date:

Note:

(i) Ph.D. Scholar shall, after Registration, submit a written report to Doctoral Committee in the required format, annually for the first three years, and every six months thereafter.

(ii) M.S. Scholar shall, after Registration, submit annually a written report to Academic Progress Committee.

(iii) Attach additional sheets if required.

INDIAN INSTITUTE OF TECHNOLOGY MANDI
HIMACHAL PRADESH, INDIA - 175075
www.iitmandi.ac.in

APC/DC RECOMMENDATION (Part B)

Scholar's Name: ARJUN H KUMAR **Roll No:** S21008
School: SCEE **Date of APC/DC meeting:** X July 2022

	Performance (Poor, Average, Good, Very good, Exceptional)	**Suggestions**
Oral Communication and Presentation		
Subject Knowledge		
Research Output		
OVERALL PERFORMANCE *(as per Part-A): Very Good/Good/Satisfactory/Unsatisfactory:*		
Overall feedback/Remarks:		

APC/Doctoral Committee

	Faculty Name	**Signature**
Chairperson APC/DC	Dr. Aditya Nigam	
Guide	Dr. Manas Thakur	
Member	Dr. A.D. Dileep	
Member	Dr. Gaurav Bhutani	
Member	Dr. Varunkumar Jayapaul	

I have read and noted the above for compliance.

Signature of the scholar with Date: