MAJOR TECHNICAL PROJECT ON

Improving Stack Allocation in Eclipse OpenJ9 Java Runtime

MTP REPORT

to be submitted by

Dheeraj
B17041

Swapnil Rustagi
B17104

for the award of the degree
of

BACHELOR OF TECHNOLOGY IN
COMPUTER SCIENCE AND ENGINEERING

SCHOOL OF COMPUTING AND ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MANDI

June 2021

Abstract

In Java and other modern object-oriented languages, an object can be allocated either on the heap or on the stack. In Java, by default, all objects are allocated on the heap. Allocating objects on the stack improves performance by reducing the garbage collector overhead.

Escape analysis allows us to determine which objects can be stack-allocated. It is not possible to perform computationally expensive precise inter-procedural escape analysis during JIT compilation. Thakur and Nandivada recently proposed a framework called PYE [1] as a way to use the results of precise static analyses during JIT compilation.

In our project, we used PYE's idea to improve the number of stack allocations. We performed precise inter-procedural escape analysis statically and used its output to perform stack allocation in Eclipse OpenJ9 Java runtime. We used the Soot framework to perform the escape analysis and the results were encoded using the Java Bytecode indices. In OpenJ9, we read these results and stack allocate the objects. Results of static analysis can never be perfect because of the unsoundness in call graph and dynamic language features like hot code replacement, reflection, etc. In such cases, we optimistically stack allocate the objects and implement a mechanism in OpenJ9 to detect escapes at runtime and also handle them by copying the objects to the heap and updating the references to the new heap location.

We also computed stack allocation and heapification statistics for Dacapo [2] and SPECjbb2005 benchmarks.

Keywords: *Java, Stack Allocation, Escape Analysis, OpenJ9, Soot, PYE*

Contents

Chapter 1

Introduction

In Java, by default, all objects (and arrays) are allocated on the heap [3]. Other than the built-in primitive data types (*int*, *float*, *double*, etc.), use of all other data types requires object allocation. Java has automatic memory management and heap memory no longer in use is periodically freed by the garbage collector, but it has a significant overhead.

In some cases, it is possible to allocate objects on the stack instead. Many Java compilers perform an analysis known as *escape analysis* [4] to determine which objects can be allocated on the stack.

Allocating objects on the stack whenever possible is beneficial for performance. Accessing objects allocated on the stack is faster than accessing objects allocated on the heap due to better cache usage (spatial locality of reference). Also, stack frames get popped as soon as a method finishes execution and returns, so stack allocated objects do not require garbage collection.

OpenJ9 JVM can perform precise intra-procedural escape analysis during runtime in the JIT compiler, however it performs imprecise inter-procedural analysis because it is computationally expensive and will lead to very high runtime overheads. It is possible to perform precise inter-procedural static analysis during compile time without incurring the runtime overheads.

Thakur and Nandivada recently proposed a framework called PYE[1]. This framework allows us to perform precise analysis statically and use its results during JIT compilation. This opens up a new way to perform stack allocations during runtime. We can perform precise inter-procedural escape analysis statically and use its results during runtime.

The static analysis and JIT works in very different environment and very little common information is passed on to the JIT. The JIT only sees the byte codes, so we must link the escape analysis information with the byte codes.

The results of static analysis cannot be trusted by the JIT. The static analysis might contain some unsoundness which can lead to wrong object being stack allocated. Java also supports various dynamic features like hot code replacement and reflection. In presence of dynamic features the results of static analysis can be unsound because static analysis cannot look into what is happening during runtime. Hence, we require a fall-back mechanism to make sure that the program execution does not crash because of unsoundness in the static analysis results.

Chapter 2

Background

2.1 Java program execution

Java source code is first statically compiled to Java bytecode. The static compiler does not perform any significant optimizations. Java bytecode is a platform-independent instruction set, where opcodes and operands are represented by one byte each. A Java runtime is required to execute this bytecode on a system. A Java runtime comprises mainly of the Java virtual machine (JVM), along with the Java class library (JCL) and just-in-time (JIT) compiler(s).

When a Java program is to be executed, the JVM interprets the bytecode. Interpreting the bytecode offers reasonable performance if the program does not do heavy computation. However, for computationally expensive long running programs, interpreting the bytecode is usually much slower than compiling the bytecode to machine code and executing the machine code. As a result, when executing programs, most JVMs interpret the methods and profile their execution – if a method is executed more times than a certain threshold, the JVM then compiles the bytecode of that method to machine code and executes the machine code.

This compilation, which happens when the Java program is already being executed (by the bytecode interpreter) is referred to as *just-in-time (JIT) compilation*. Like other compilers, a Java JIT compiler also performs many optimizations.

2.2 OpenJ9 Java runtime

Eclipse OpenJ9 [5] is a Java runtime, initially developed by IBM, and later open sourced, which focuses on performance for cloud applications. It is used in production by IBM and many others.

The Testarossa JIT compiler in OpenJ9 takes bytecode and converts it into an intermediate tree representation with nodes representing operations and operands. It then applies several optimizations (one of which is escape analysis) on this tree representation, and after optimizations, processes each node to generate corresponding machine code. However, since optimizations are happening at runtime, there are some limitations in the existing escape analysis in OpenJ9 (explained in Section 2.4).

In OpenJ9, methods are JIT compiled at various optimization levels, corresponding to their invocation and execution profile. Methods that do not take up significant execution time are compiled with fewer optimizations.

The overall program execution, including garbage collection, even for JIT compiled code is still managed by the JVM, and the generated machine code makes calls to JVM methods for various reasons like memory allocation, throwing exceptions, etc. These JVM methods which JIT compiled code can call into are known as runtime *helper* methods within OpenJ9. This project requires implementing a new helper method for *dynamic heapification* (described in Section 3.1).

2.3 Escape Analysis

An object is said to escape its method of allocation if it can outlive the activation record of that method. Escape analysis finds all the objects which are escaping. Let us take the following example:

```
1 class EscapeAnalysis{
2    static Obj g;
3    void foo() {
4       Obj A = new Obj(); // O4
5       Obj B = new Obj(); // O5
6       B.f = new Obj(); // O6
7       Obj D = bar(B);
8    }
9    Obj bar(Obj x) {
10      g = x.f;
11      Obj E = new Obj(); // O11
12      return E;
13   }
14 }
15 class Obj {Integer i;Obj f};
```

Fig. 2.1: Code snippet for demonstrating escape analysis.

To find the objects escaping a method, we need to find the objects which are outliving the activation records of their allocation.
Code the Java code snippet in Figure 2.1. Say we represent the object(s) allocated at line l as O_l.

- O_6 is escaping. Object O_6 is being assigned to a static variable and can be used anywhere.
- O_{11} is also escaping. Object O_{11} is being returned from `bar` and thus function `foo` can use O_{11}.
- Object O_4 and O_5 are not escaping.

2.4 Limitations of OpenJ9's Escape Analysis

Consider the code snippet in Figure 2.1. As O_4 is not being passed to any method, an intra-procedural analysis (not taking any method calls into account) is enough to conclude that O_4 does not escape. However, this is not the case for O_5 and O_6. Consider the call to the method `bar` at line 7. This call can make the object pointed to by B (O_5) or one of its fields (O_6), escape. In this snippet, the statement at line 10 actually does make the object O_6 escape. To conclude whether a call makes some object in the caller method escape or not, it requires an analysis of the callee method as well.

OpenJ9 performs precise intra-procedural escape analysis, but method calls are handled imprecisely. The callee method itself may also have method calls, and determining whether an object escapes or not may require the analysis of multiple methods.

While OpenJ9 analyzes calls, this analysis is happening at runtime and so must be very efficient. In cases where a chain of multiple calls need to be analyzed, OpenJ9 stops the analysis after a certain (small) depth in the call chain and marks the corresponding object as escaping (even in cases where it is not).

2.5 PYE Framework for Escape Analysis

PYE (Precise-Yet-Efficient) framework [1] describes a way to perform precise and time consuming analysis and yet achieve efficient run times.

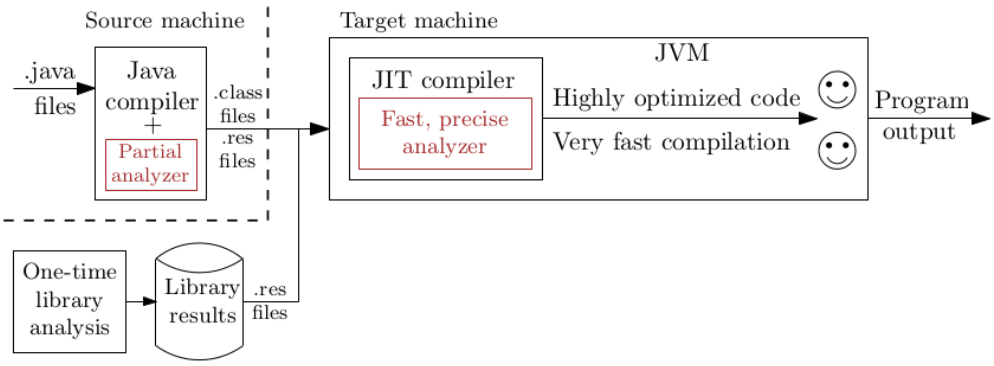

Fig. 2.2: PYE Framework.

Major highlights from the paper:

- Rather than doing all the analysis during run time, analysis can be done during compilation to Java bytecode.
- The application code and Java library code should be segregated and analysed separately. The results should be merged during JIT compilation.
- This approach allows us to perform precise analysis without much cost at the time of JIT compilation. Also, segregation of library and application makes sure that we do not analyse the ”static” code again and again.

2.6 Soot

Soot [6] is a state-of-the-art Java optimization framework. Soot allows us to perform analysis on Java source code and has several in-built analyses. Specifically for this project, we are using Soot's source to source transformation from Java to Jimple intermediate representation (IR). We also use Soot's in-built Spark [7] call graph for inter-procedural analysis.

2.7 Bytecode Indices and Object Allocation Bytecodes

In Java bytecode, the opcodes *new*, *newarray*, and *anewarray* correspond to the allocation of an object, an array of primitive types, and an array of objects, respectively. For this project, we analyze whether the allocations corresponding to any of these bytecodes can be stack allocated.

Each byte (corresponding to opcodes and operands) in the bytecode for a method has an index; these indices are known as bytecode indices (BCI). For this project, bytecode indices are used to pass information from static analysis to the JIT compiler. The result of the static analysis contains the bytecode indices corresponding to allocations of the objects which are not escaping. The JIT compiler can read this list of indices and make the corresponding allocation on the stack.

2.8 Stava

Stava project started in 2019 as Nikhil T.R.'s MTP project. The Stava project uses PYE's idea for escape analysis, but the work could not be made production ready during his tenure of MTP.

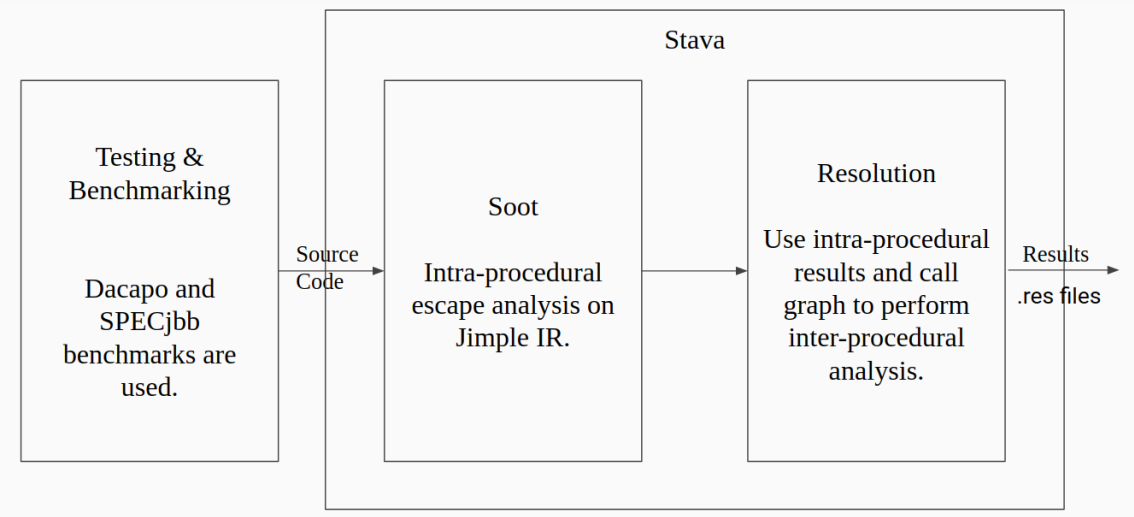

Fig. 2.3: Architecture Design of Stava.

The source code of Stava [8] consists of two phases: intra-procedural static analyser and inter-procedural resolver. Static analyser performs intra-procedural escape analysis on source method separately and stores the dependency of each object's escape status on other object's escape status. These dependencies are named *summaries* in Stava. Static analyser also creates a points-to graph of the source code.
A points-to relationships captures the relationship between the objects. We can take code snippet in Figure 2.1 as an example. The Points-to relationships for this snippet are shown in Figure 2.4:

```
1 {
2     A -> O4;
3     B -> O5;
4     O5.f -> O6;
5     D -> <return of bar()>, O11;
6     E -> O11;
7     q -> O5.f, O6
8 }
```

Fig. 2.4: Points-to relationships for Code Snippet in Figure 2.1.

In the second phase of Stava, the *summaries*, *points-to graph* and *call graph* generated in first phase are used to perform inter-procedural escape analysis on the source code.

Chapter 3

Objectives and Scope of the Project

This project aims at using static analysis to improve precision of stack allocation in the just-in-time (JIT) compiler of Eclipse OpenJ9. The project is split into two parts, static analysis and modifying OpenJ9's Testarossa JIT compiler to use results of static analysis to do stack allocation.

The static (escape) analysis finds which all objects do not outlive their allocating activation record. These objects can be allocated on the stack, other objects cannot be allocated on stack and such objects are ignored.

We can spend more time on escape analysis during static compilation, so our objective is to perform escape analysis statically and pass the results to OpenJ9. During JIT compilation, optimizations like inlining can increase opportunities for stack allocation. For instance, if a method simply allocates an object and returns it, then the static analysis would be precise to mark the object as escaping but if the JIT compiler inlines the method into the caller(s), the object may be non-escaping.

In this project, we modify the escape analysis in OpenJ9 such that we can gain the benefit of both approaches. At any point if existing analysis concludes that an object is escaping, we check the static analysis results and only mark the object escaping if it is escaping according to the static analysis as well. Consequently, the precision of the escape analysis in OpenJ9 can be improved, resulting into an increase in the number of stack allocated objects.

The scope of the project includes fixing/improving static analysis for Java 8 code. We also aim to make sure that result of the static analysis is usable during JIT compilation, especially in presence of Java features described in Section 3.2.

3.1 Escape Analysis

The precision and unsoundness in the results of the Escape Analysis is directly reflected into the number of optimistic stack allocation and dynamic heapifications performed. Heapifications are very costly. The objective of static escape analysis is to reduce the number of heapifications as much as possible. If the static analysis is perfectly sound and no dynamic feature is invoked during runtime, then the number of heapifications should be zero.

The scope of this project is to implement and improve the static escape analysis. The implemented scheme should have near non-existent unsoundness as well as good precision. The scheme should be able to graciously handle all language features of Java 8 programs.

3.2 Optimistic Stack Allocation and Dynamic Heapification

Java supports features like dynamic classloading [9] and hot-code replacement which can cause changes in the Java program's code at runtime. This can lead to static analysis results no longer being valid for some method(s). In these cases, we can still use our static analysis results, however, we need a runtime mechanism to check if we had wrongly allocated an escaping object on the stack. Furthermore, for such objects we also need a mechanism to copy the object from the stack to the heap and update all previous references to the object to the new heap location (the *heapification* of an object). Although this heapification for a wrongly stack allocated object has significant overhead, we expect the situations where heapification is required to arise rarely in practice, and that using the static analysis results will yield an overall performance improvement. We call this novel approach as *optimistic stack allocation*.

3.2.1 Conditions for escape at runtime

A stack allocated object O allocated in a method M, escapes when the address of O is being stored into a reference accessible outside the stack frame of M. This leads to the following cases where escapes could happen:

1. Store into a heap location
2. Store into a field of a parameter passed to M, even if the parameter is also a reference to a stack allocated object

3. Return from M

4. Throwing an exception (in case O was an exception object) - can be viewed as storing the exception object into the JVM environment

3.2.2 Heapification check

To detect and handle escapes at runtime, we added a check before reference stores and returns to check if the reference being stored/returned is on the stack. If it is, then we heapify the object. The implementation details are in Section 4.2.2 while Figure 3.1 shows the overall working of the optimistic stack allocation scheme.

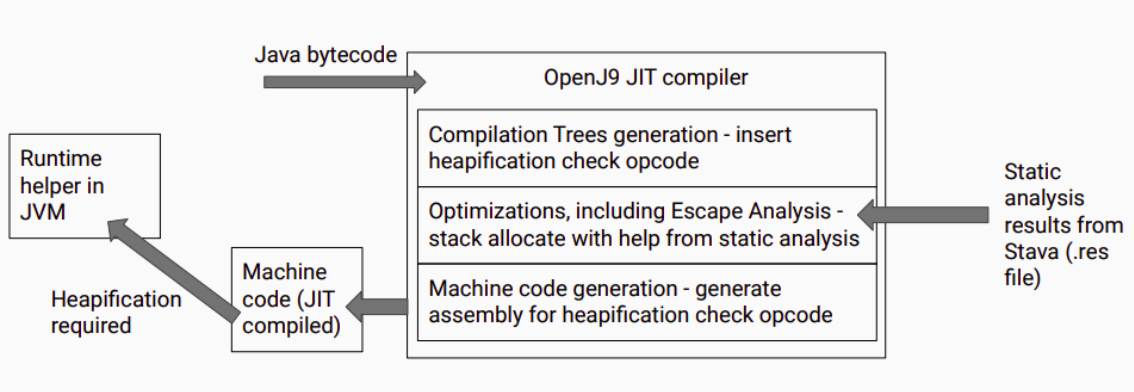

Fig. 3.1: Optimistic stack allocation scheme

This heapification check is conservative with respect to stores. If both the destination and value in a store are stack allocated objects on the same stack frame, then it is not an escape. However, the implementation will still heapify the object pointed to by the value reference, because it simply checks that the value being stored is on the stack.

It is possible to have a more precise implementation which correctly recognize non-escaping stores by doing a walk of the stack frames to determine if objects are in same stack frame. However it would have a very high overhead, comparable to that of heapification itself, but unlike heapification, the check maybe required frequently in practice. So we opt for a low-overhead but imprecise check. As a consequence of this implementation, the static analysis also marks all stores to fields of non-escaping objects as escapes.

Chapter 4

Work done

4.1 Odd semester

4.1.1 Static Analysis [Dheeraj]

- **Bytecode Indices**
  Soot provides us option to use any of the two backends: ASM [10] and Coffi. Coffi has been deprecated and does not support Java 8 source code, whereas ASM does not support bytecode indices. To add support for Java 8 code, ASM and Soot were extended to support bytecode indices. A new *bytecodeOffset* field was added to the *AbstractInsnNode* class. This value was then filled by the *methodVisitor* of *ClassReader*. In Soot, the value of this *bytecodeOffset* variable is read and *ByteCodeOffset* tag is created for each instruction.

- **Fixed handling of *newarray* statements.**
  Soot merges multiple instructions into a single Jimple Statement. When allocating arrays, the bytecode index associated with the entire Jimple statement is not the bytecode index of *newarray* instruction but of the following instruction. For example, consider the statement: *Integer[] a = new Integer[5];*

```
0: java.lang.Integer[] r0;
4: r0 = newarray (java.lang.
   Integer)[5];
```

Fig. 4.1: Jimple transformation and the BCI of each statement.

```
0: iconst_5
1: anewarray #6 // class java
   /lang/Integer
4: astore_3
```

Fig. 4.2: Bytecode of the Jimple code and BCI of each Instruction.

For the results to be usable in JIT Compiler, BCI of *r0 = newarray (java.lang.Integer)[5];*

Jimple Statement should be **1** instead of **4**. This issue was fixed by splitting the Jimple statements and finding the bytecode index of required new statement.

- **Implemented Deterministic Resolver:**
  Stava's current `Resolution` phase is not deterministic and the output depends on the order in which methods were analysed during static analysis phase. This renders the results unusable. Thus, the old resolution was scraped all-together and a new resolver was written. This new Resolver is named as `ReworkedResolver`. The `ReworkedResolver` first constructs all objects as a pair of *SootMethod* and *ObjectNode* into a single object of class `StandardObject`. It then resolves all method calls and replaces dummy parameters with actual arguments passed and updates the *summaries* data structure accordingly.

Algorithm 1: mergeSummaries: Merge summaries of caller with callee.

```
input : summaries: Map<StandardObject, EscapeStatus>, callGraph:
        CallGraph
for object,status in summaries do
    for cv in status.ConditionalValues() do
        if cv of type Argument then
            argumentNumber ← cv.parameterNumber;
            callList ← callGraph.getCallers(object.method());
            objectList ← getParameters(callList, argumentNumber);
            summaries.get(object).add(objectList);
        else
            continue;
        end
    end
end
```

Algorithm 2: constructGraph: Construct Graph using Summaries and Points-to Graph

```
input  : summaries: Map<StandardObject, EscapeStatus>, pointsToGraph:
         PointsToGraph
output: graph: Map<StandardObject, List<StandardObject>>
graph ← Map < StandardObject, List < StandardObject >>
for object,status in Summaries do
    for cv in status.ConditionalValues() do
        graph.get(object).add(cv.getObject());
    end
end
for object, fields in pointsToGraph do
    for field in fields do
        for fieldObject in field.objects() do
            graph.get(fieldObject).add(object);
        end
    end
end
return graph
```

It then builds an object dependency graph based on the summaries data structure, points-to graph and call graph. Object dependency graph is a graph consisting of directed edges, where each edge $A \rightarrow B$ represents that if B escapes then A also escapes. Next, we find the strongly connected components (SCC) and build a condensed directed acyclic graph of strongly connected components. In the next step, we traverse this condensed graph in reverse topological order and analyse each of the SCCs. If any object in the SCC depends on an escaping object, then all objects of that SCC are marked as escaping.

- **Fix issues in Intra-procedural phase:**
  The existing implementation of Intra-procedural analysis used normal hash-maps to maintain the summaries. Soot analyzes multiple method concurrently, which resulted in data-races. All thread-unsafe hash-maps were replaced with thread-safe hash-maps to fix this issue.

4.1.2 OpenJ9 Runtime [Swapnil]

- **Implemented reading of .res files in OpenJ9**

  The .res files contain the results generated by static analysis and consist of key-value pairs, with method signatures as key, and a list of bytecode indices (BCIs) as

value. This list contains BCIs of all allocations within that method that are stack-allocatable.

- **Using static analysis results for stack allocation**

  The Testarossa JIT compiler runs optimizations on the method being compiled, so escape analysis finds out all nodes corresponding to object/array allocations and then analyzes whether each of these is escaping or not. It does so in two parts, checking escapes via calls, and checking escapes via non-call statements (intra-procedural). Checking escapes via calls triggers the analysis of the callee procedure (again, both non-call and call statements) to see if it lets the argument passed to it escape.

  The escape analysis in OpenJ9 was modified to not perform analysis of method calls but to directly use the static analysis results. This means that for the code snippet in Figure 2.1, OpenJ9 no longer needs to analyze the method `bar` to check if any of O_4, O_5 or O_6 are escaping (*this is no longer the case, see Section 4.2.2 instead*). The .res file will only contain the BCIs for O_4 and O_5 in list for the method foo, which will cause OpenJ9 to correctly mark O_6 as escaping.

  Simply checking the list of BCIs for the method undergoing escape analysis is not feasible as the JIT compiler might inline a method into another method. For instance, the method `bar` could be inlined into `foo`. In this case, the method `bar` is never compiled and optimized independently, instead the call to `bar` is replaced with the body of `bar` in the method `foo`. This can lead to issues like two nodes in `foo` now having the same bytecode index (bytecode indices are assigned by the static compiler starting from 0 for each method) – so it would be unknown which one is being referred to by the static analysis result. However, OpenJ9 provides a way to get the signature of the method from which a node originated, and so this issue was fixed by checking if an allocation node originated from an inlined method, and if it did, then getting the signature of the inlined method and checking the entry in the .res file corresponding to the inlined method instead of the method being analyzed.

- **Optimistic stack allocation and dynamic heapification**

  - Added runtime check in Testarossa's code generator to check if a stack address is being stored into a heap location: added code in the address store evaluator to fetch the starting/base address and ending address of the heap memory region of the JVM, and check if the destination address is within these bounds (which indicates it is heap allocated), and the source address is outside these bounds (which indicates it is stack allocated). *This has now been refactored into a separate opcode, with changes to the conditions (see Section 4.2.2).*

4.2 Even Semester

4.2.1 Static Analysis [Dheeraj]

- **Implemented handling of special instructions:**

  Java allow users to call pre-compiled *native methods*. The implementation of these *native methods* is written in C/C++. The Static Analysis cannot analyse such methods. As a result, any object which interacts with *native method* is marked as *escaping* and should be allocated on the heap.

  Some of the DaCapo benchmarks uses JSR [11] instructions in their byte codes. JSR instructions ideally should not be present in the class files because they are deprecated but JVMs can execute them without any issue. ASM backend is unable to compute BCIs for methods with JSR instructions. Such methods are classified as *noBCImethods*. Any object allocated in these functions or any object interacting with these methods is marked as *escaping* and should be allocated on the heap.

- **Fix implementation of Worklist Algorithm:**

  The Data Flow Values in worklist algorithm should monotonically move up the lattice. If we cannot guarantee that values are moving monotonically then we cannot guarantee the termination of analysis. This bug existed in *stava* which caused few functions in *jdk.internal.** classes to not terminate. The bug was fixed by taking a meet of the current data flow value and upcoming data flow value to impart monotonicity.

  The next issue in the worklist algorithm caused *stava* to crash whenever *INs* and *OUTs* of some Unit was not available. This should not be the case, because the values can be computed in future iterations of the worklist algorithm.

- **Fix issues in Intra-procedural phase:**

  Each method has declaring class and name as identifiers. It is possible to have two methods with same name in two different classes. The existing implementation of *hashCode()* of *ConditionalValue* class ignored the class name and only took method name into account for computing the hashcode. Therefore, we can cannot add multiple methods of same name into HashSets of ConditionalValue. This resulted in dependence of HashSets on the order of processing of dependencies. The order of dependencies is not constant across different executions, thus the results were not constant. This issue was fixed by taking declaring class into consideration for computing the hash code.

- **CallGraph issues:**

  Stava initially used *Spark* [7] call graph to perform inter-procedural analysis. We noticed that *Spark* call graph is unsound, which resulted in unsound static analysis results. As a fall-back mechanism, *CHA* [12] call-graph is now being used if there are no out-going edges from a call-site.

  We noticed that in case of **invokespecial** [13] instructions, even the CHA call-graph is unsound. **invokespecial** instructions do not need dynamic binding and are used to call methods of same class or of the super class. In such cases, the target method can be extracted using the *InstanceInvokeExpr.getMethod()* method.

4.2.2 OpenJ9 Runtime [Swapnil]

- **Use of static analysis results during escape analysis**
  - Made changes in OpenJ9's escape analysis in places where the objects are marked as escaping, to check static analysis results first. However, we can't simply do this at all places, because of runtime constraints. For instance, there are checks on object sizes, and an object might be too big to accommodate on the stack. In such cases we must let OpenJ9 discard the object from stack allocation even if an object is non-escaping according to static analysis.
  - Marked *optimistically* allocated objects separately and disabled certain transformations. If we used static analysis to mark an object as non-escaping in OpenJ9, then we call it *optimistically* allocated and there is a possibility that this object might need to be heapified. This means for such an object we should not do some transformations which are valid for objects stack allocated based on the existing analysis. For instance, synchronization elision (eliminating monitor enters and exits for stack allocated objects) must not be done for an optimistic allocation.

- **Optimistic stack allocation and dynamic heapification**
  - Implemented a helper method in the JVM which takes the stack object address as an argument and heapifies the object. We find the class of the object, allocate a new object of the class on the heap and copy the field values. Then, similar to compaction phase in some garbage collection policies in OpenJ9, we use the stackwalker in OpenJ9 to go through all live references to the stack object, in all frames and update them to the new heap location.
  - Added a new opcode in the JIT compiler (named *TR::possibleHeapification* - all opcodes are in the *TR* namespace in the code), which takes a reference as its only child. In the evaluator for this opcode, we generate assembly instructions

to fetch the address bounds of the Java stack, and then do a comparison to see if the reference is within these bounds. If yes, then we call out to the VM helper for heapification, else we jump over it. This opcode is inserted during the trees generation phase, while generating trees for bytecodes corresponding to:

  * Stores to instance fields, static fields and object arrays (bytecodes *putfield*, *putstatic* and *aastore* respectively)
  * Returns of references (*areturn* bytecode)
    A separate opcode is used for return checks with an additional condition to heapify a return value only if it is allocated on the returning method's stack. This avoids heapification where a passed paramter is returned.
  * Throwing of exceptions (*athrow* bytecode)
  * Calls to JNI methods in *sun.misc.Unsafe*: *putObject*, *putObjectVolatile* and *compareAndSwapObject* which may lead to reference stores.

- Similar checks in the bytecode interpreter for the bytecodes and *sun.misc.Unsafe* mentioned above, except for returns, because an interpreted method would not have any stack allocated objects in its stack frame. The interpreter is part of the JVM itself, so the functionality of the VM helper for the JIT compiler, is similarly implemented in the interpreter itself.

- **Fixes for TamiFlex**

  Fixed two issues in the Play-out agent in TamiFlex [14] when running with OpenJ9. TamiFlex transforms classes at runtime so that reflective calls during execution can be logged. The logs from TamiFlex are used to generate the call graph in Stava at reflective call sites. TamiFlex was not correctly generating the *stackMapTable* [15] in transformed classes, leading to bytecode verification failure in OpenJ9. Also TamiFlex logged lambda classes according to HotSpot's naming convention, OpenJ9 had a slightly different naming convention for lambda classes.

Chapter 5

Evaluation and Results

5.1 Implementation Details

We implemented our project in two parts: (i) *Stava*(see Section 2.8)- approximately 2500 lines of changes to existing scheme; (ii) Stack Allocation changes and heapification in Eclipse OpenJ9 Runtime - approximately 500 lines of code. The *Stava* itself is divided into two parts: (i) Intra-procedural analyser - 2000 lines of code; (ii) Inter-procedural resolver - 650 lines of code. The changes in OpenJ9 are spread across the JVM (run-time helper for heapification and corresponding changes in bytecode interpreter), escape analysis optimization (read and use static analysis results, and counters for statistics), IL generator (inserting heapification check opcodes when generating trees from bytecode) and code generator (generating the check to see if object is on stack).

We have modified the existing source code of *ASM* backend and *Soot* Framework to facilitate bytecode generation - approximately 120 lines of changes. In addition to this, we have also bug-fixed *Tamiflex* [14] to support analyses with OpenJ9 JDK - approximately 25 lines of changes. The *Tamiflex* tool instruments the code to generate *reflection log*. This *reflection log* is used by *soot* to generate Call-Graph for reflective calls.

5.2 Setup Details

We evaluated Stava on 10 benchmarks from 2 benchmark suites: (i) *avrora, sunflow, batik, lusearch, luindex, eclipse, pmd, h2 and fop* from the *DaCapo* 2009 benchmark suite [2] - using settings for small size; (ii) SPECjbb 2005 [16] - using default settings. The rest of the Dacapo benchmarks could not be analysed - either by Soot or by the Tamiflex. The static analysis has been performed on the IIT Mandi High Performance Cluster (HPC) with 50GBs of Heap Size in the *day* queue. We also evaluated the improvement in number

of stack allocations in OpenJ9 by using the static analysis, over the same benchmarks with default settings (except for *batik* where we used small size due to a runtime exception with default size, which occurs even without our changes). This was also done by executing on the IIT Mandi HPC.

5.3 Results

5.3.1 Static Escape Analysis

S. No.	Benchmark	Inter-procedural phase Time	Resolution Time	Total Time	Total Objects	Non-escaping objects after static analysis	Non-Escaping objects after Resolution	Percentage after static analysis	Percentage after resolution
1	avrora	105.639	19.502	125.141	32306	587	2925	1.82%	9.05%
2	sunflow	173.199	22.058	195.257	44283	1102	5126	2.49%	11.58%
3	batik	155.36	35.146	190.506	50682	1193	4740	2.35%	9.35%
4	lusearch	56.921	9.186	66.107	31758	610	2951	1.92%	9.29%
5	luindex	67.128	9.871	76.999	34219	616	2957	1.80%	8.64%
6	eclipse	226.033	20.752	246.785	44283	1102	5126	2.49%	11.58%
7	pmd	83.291	12.622	95.913	35401	918	3520	2.59%	9.94%
8	h2	61.466	7.717	69.183	36047	654	3458	1.81%	9.59%
9	fop	257.529	20.449	277.978	58601	1182	5518	2.02%	9.42%
10	specjbb	116.755	14.124	130.879	39338	941	4350	2.39%	11.06%
11	GeoMean	114.219	15.499	130.308	39914.7	856.8	3951.7	2.15%	9.90%

Table 5.1: No. of non-escaping objects reported by Stava for DaCapo and SPECjbb benchmarks.

5.3.2 Stack allocation results

Allocation sites denote the number of objects in methods that could be stack allocated while runtime allocations counts the actual number allocated on the stack, taking into account multiple invocations of the method.

S. No.	Benchmark	Allocation sites (Existing)	Allocation sites (Optimistic scheme)	Improvement factor	Runtime allocations (Existing)	Runtime allocations (Optimistic scheme)	Improvement factor
1	avrora	24	72	3	12	73	6.08
2	batik	36	106	2.94	45	126	2.8
3	eclipse	103	231	2.24	211	401	1.9
4	fop	45	117	2.6	1	7	7
5	h2	74	134	1.81	106	182	1.72
6	luindex	26	80	3.07	9	70	7.78
7	lusearch	28	76	2.71	43	80	1.86
8	pmd	40	151	3.78	1258	1860	1.48
9	sunflow	243	336	1.38	268428	268458	1.0001
10	specjbb	114	241	2.11	12	98	8.17

Table 5.2: No. of stack allocations with forced compilation (every method is JIT compiled from the start instead of being interpreted, using -Xjit:count=0).

S. No.	Benchmark	Allocation sites (Existing)	Allocation sites (Optimistic scheme)	Improvement factor	Runtime allocations (Existing)	Runtime allocations (Optimistic scheme)	Improvement factor
1	avrora	10	12	1.2	374616	374814	1.0005
2	batik	1	10	10	395	2316	5.86
3	eclipse	35	39	1.11	1274119	1274195	1.00005
4	fop	3	8	2.67	397	5095	12.83
5	h2	56	81	1.45	553393	1104289	1.99
6	luindex	15	26	1.73	3445	5008	1.45
7	lusearch	3	7	2.33	226960	325159	1.43
8	pmd	14	37	2.64	0	0	N/A
9	sunflow	112	114	1.02	7667073	7667073	1
10	specjbb	87	93	1.07	136493665	136493667	1

Table 5.3: No. of stack allocations with normal execution.

Chapter 6

Future Work

- We plan to submit this work either to Principles of Programming Languages (POPL) 2022 conference or to ACM Transactions on Programming Languages and Systems (TOPLAS) journal.
- Inlining opens up a lot of opportunities in stack allocations. We plan to improve static analysis to give inlining hints and determine which objects can be stack-allocated if some method has been inlined.
- Currently, we mark every returned object as escaping in the static analysis. This is not precise. Suppose we have this method:

```
A foo(A param) {
   return param;
}
void bar() {
   A var1 = new A(); // O1
   A var2 = foo(var1);
}
```

Fig. 6.1: Parameter returned from the method.

In this example, objects pointed by *param* i.e. *O1* can be allocated on the stack of method *bar*. Current scheme cannot stack allocate such objects because they are being returned from a method.

- Exploring ways to overcome the conservativeness of the heapification check, (described in Section 3.2.2) to enable reference stores among objects on the same stack frame to be considered non-escaping.

Chapter 7

Project Outcomes

- Nikhil T R, Dheeraj, Swapnil Rustagi, Arjun Bharat, Vijay Sundaresan, Andrew Craik, Daryl Maier, Manas Thakur, and V. Krishna Nandivada, “Improving Stack Allocation on Eclipse OpenJ9 using Precise Static Analysis”, 4th Workshop on Advances in Open Runtime and Cloud Performance Technologies (AORCPT), co-located with CASCON-EVOKE, November 2020.

Bibliography

[1] Manas Thakur and V. Krishna Nandivada. "PYE: A Framework for Precise-Yet-Efficient Just-In-Time Analyses for Java Programs". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* (July 2019).

[2] S. M. Blackburn et al. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". In: *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. Portland, OR, USA: ACM Press, Oct. 2006, pp. 169–190. DOI: http://doi.acm.org/10.1145/1167473.1167488.

[3] Tim Lindholm et al. *The Java Virtual Machine Specification - Java SE 11 Edition*. section 2.5.3. Redwood City, California, USA: Oracle America, Inc., 2018. also available at https://docs.oracle.com/javase/specs/jvms/se11/html/jvms-2.html#jvms-2.5.3.

[4] Bruno Blanchet. "Escape Analysis: Correctness Proof, Implementation and Experimental Results". In: *25th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'98)*. San Diego, California: ACM Press, Jan. 1998, pp. 25–37.

[5] Eclipse Foundation. *Eclipse OpenJ9*. 2021. URL: https://www.eclipse.org/openj9/.

[6] Raja Vallée-Rai et al. "Soot - a Java Bytecode Optimization Framework". In: *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON '99. Mississauga, Ontario, Canada: IBM Press, 1999, p. 13.

[7] Ondrej Lhoták and Laurie Hendren. "Scaling Java points-to analysis using SPARK". In: vol. 2622. Apr. 2003, pp. 153–169. ISBN: 978-3-540-00904-7. DOI: 10.1007/3-540-36579-6_12.

[8] Compl-IITMandi. *Stava*. 2021. URL: https://github.com/CompL-IITMandi/stava.

[9] Thomas Kotzmann and Hanspeter Mössenböck. "Escape analysis in the context of dynamic compilation and deoptimization". In: *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments* (June 2005). DOI: 10.1145/1064979.1064996.

[10] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. "ASM: A code manipulation tool to implement adaptable systems". In: *In Adaptable and extensible component systems*. 2002.

[11] Oracle. *jsr*. 2021. URL: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html#jvms-6.5.jsr.

[12] Jeffrey Dean, David Grove, and Craig Chambers. “Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis”. In: *Proceedings of the 9th European Conference on Object-Oriented Programming*. ECOOP ’95. Berlin, Heidelberg: Springer-Verlag, 1995, pp. 77–101. ISBN: 3540601600.

[13] Oracle. *Java Virtual Machine Specification: invokespecial*. 2021. URL: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html#jvms-6.5.invokespecial.

[14] Eric Bodden et al. “Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders”. In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE ’11. Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, pp. 241–250. ISBN: 9781450304450. DOI: 10.1145/1985793.1985827. URL: https://doi.org/10.1145/1985793.1985827.

[15] Oracle. *Java Virtual Machine Specification: The StackMapTable Attribute*. 2021. URL: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.7.4.

[16] SPEC. *SPECjbb*. 2005. URL: https://www.spec.org/jbb2005/.