# Programming Assignment - 4 Report (CS6004)

Abhijat Bharadwaj (210020002), Prekshu Pranav Singh (210020098)

## 1 Introduction

In this assignment, we have focused on removing Redundant Null Checks to optimise the code. The performance of the optimization has been measured in terms of difference in the run time of the program. The analysis has been performed on a code containing redundant null checks. Firstly, we have performed null check analysis without removing the redundancy and measured the time taken by the program to execute. Then, we performed the analysis based on redundant null checks. Using the transformer, we removed those redundant null checks and measured the time taken by the program to execute. The difference in these two times gives our measure of improvement.

Please find the code in the below mentioned GitHub Repository.
**GitHub Repository:** https://github.com/Keymii/cs60004-pa4

## 2 About the Code

### 2.1 File Structure

The file structure is as follows

```
\
| - RedundantNullChecks.java
| - NullCheckAnalysis.java
| - MergeNullChecks.java
| - *.class
| - sootclasses-trunk-jar-with-dependencies.jar
| - sootOutput\
|   | - RedundantNullChecks.class
|   | - RedundantNullChecks.jimple
|   | - transformed\
|   |   | - RedundantNullChecks.class
|   |   | - RedundantNullChecks.jimple
```

- **RedundantNullChecks.java**
  This Java program serves as our test case file. A number between 0 and 19 is picked at random. If the number is less than 10, the variable *obj* is initialized as *null*. In the first conditional statement, the program only proceeds furthes if *obj* is not null. Still, further code has redundant null checks, these are analyzed and transformed for optimization by NullCheckAnalysis and MergeNullChecks files. The program is exectuted for 9,99,99,999 iterations to measure execution time.

- **NullCheckAnalysis.java**
  This Java program analyzes Java bytecode and identify redundant null checks in the class, RedundantNullChecks. The transformation inspects each conditional statement (JIfStmt) within the method bodies of the class. If the condition of the if statement involves a comparison between a null constant and a non-null variable (using equality or inequality operators), it flags these as redundant null checks by printing the line number. The analysis is executed by running Soot with the required arguments, where the main-class is RedundantNullChecks.

- **MergeNullChecks.java**
  This Java program perform static analysis and bytecode manipulation, targeting the RedundantNullChecks class. The goal is to analyze and remove redundant null checks within the bytecode. During the analysis phase, a transformation is added to the jtp which checks each conditional statement (JIfStmt) in the bytecode. If a

statement performs a null check (using EqExpr for equality or NeExpr for inequality against a NullConstant), the program determines if this check is redundant by checking the flow of checks. It keeps track of the last non-redundant null check encountered and marks any subsequent redundant checks for removal. If a non-redundant check is encountered, it resets the tracking. After the analysis, all marked redundant checks are removed from the bytecode. Finally, the modified bytecode is saved in the sootOutput directory.

### 2.2 Exeution Steps

- **Clone the Repository**
  `git clone https://github.com/Keymii/cs60004-pa4`
- **Compile the Java and Soot files**
  (Note that the repository is already compiled and you can directly move the sootOutput folder to OpenJ9 VM)
  `javac -cp .:sootclasses-trunk-jar-with-dependencies.jar *.java`
- **Execute the Null-Check Analysis**
  `java -cp .:sootclasses-trunk-jar-with-dependencies.jar NullCheckAnalysis`
- **Execute the Redundant Null Check Merger**
  `java -cp .:sootclasses-trunk-jar-with-dependencies.jar MergeNullChecks`
- **Move sootOutput folder into OpenJ9 VM**
- **Execute the Original RedundantNullChecks class**
  `cd sootOutput/`
  `java -Xint RedundantNullChecks`
- **Execute the Transformed RedundantNullChecks class**
  `cd sootOutput/transformed/`
  `java -Xint RedundantNullChecks`

## 3 Results

We found that on *99999999 iterations* of RedundantNullChecks class,

### 3.1 Attempt 1

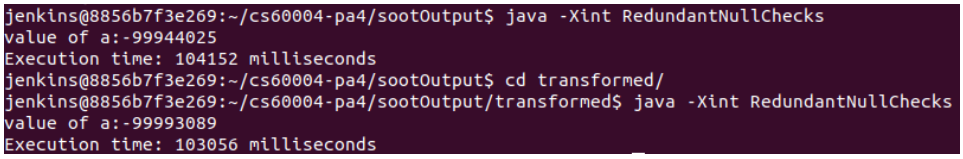


Figure 1: Improvement in performance measured by total execution time (attempt 1)

The original RedundantNullChecks class took 104152 milliseconds for execution, while the transformed class took only 103056 milliseconds. Thus, the null-check optimization transformation improved the performance by 1096 milliseconds.

### 3.2 Attempt 2

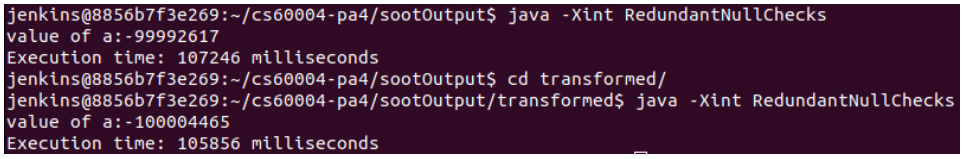


Figure 2: Improvement in performance measured by total execution time (attempt 2)

The original RedundantNullChecks class took 107246 milliseconds for execution, while the transformed class took only 105856 milliseconds. Thus, the null-check optimization transformation improved the performance by 1390 milliseconds.