# CS-6004 : Code Optimization for Object-Oriented Languages Assignment-4

A Asish (23M0759), Rohit Singh Yadav(23M0773)

---

## 1 Objective

To show performance improvement after replacing virtualinvoke call at a monomorphic callsite with staticinvoke call.

## 2 Generating analysis results

We have reused our PA3's interprocedural data flow analysis code to do the type analysis for each variable. We are keeping track of object types that are flowing to each callsite, and then finding monomorphic callsites. We are then replacing the virtualinvoke call at that callsite with a staticinvoke call according to the type of object that is flowing to that callsite. To enable calling using staticinvoke, we are adding a static method that is clone of the method that was being called through virtualinvoke during runtime in the unmodified code.

## 3 Generating Jimple files

While doing the above analysis, we were getting an exception as shown below in figure 1 that we were unable to resolve even after the best of our efforts. We also tried asking on piazza but didn't find answer on how to resolve it.

Our analysis was running fine till the end, it also added new static method to the class and changed the virtualinvoke unit to staticinvoke unit, but was giving the exception at the end of the program. So we had to take a roundabout way of moving forward. We printed all the methods of all classes in our code, and then manually edited the jimple files to reflect the new structure of classes.

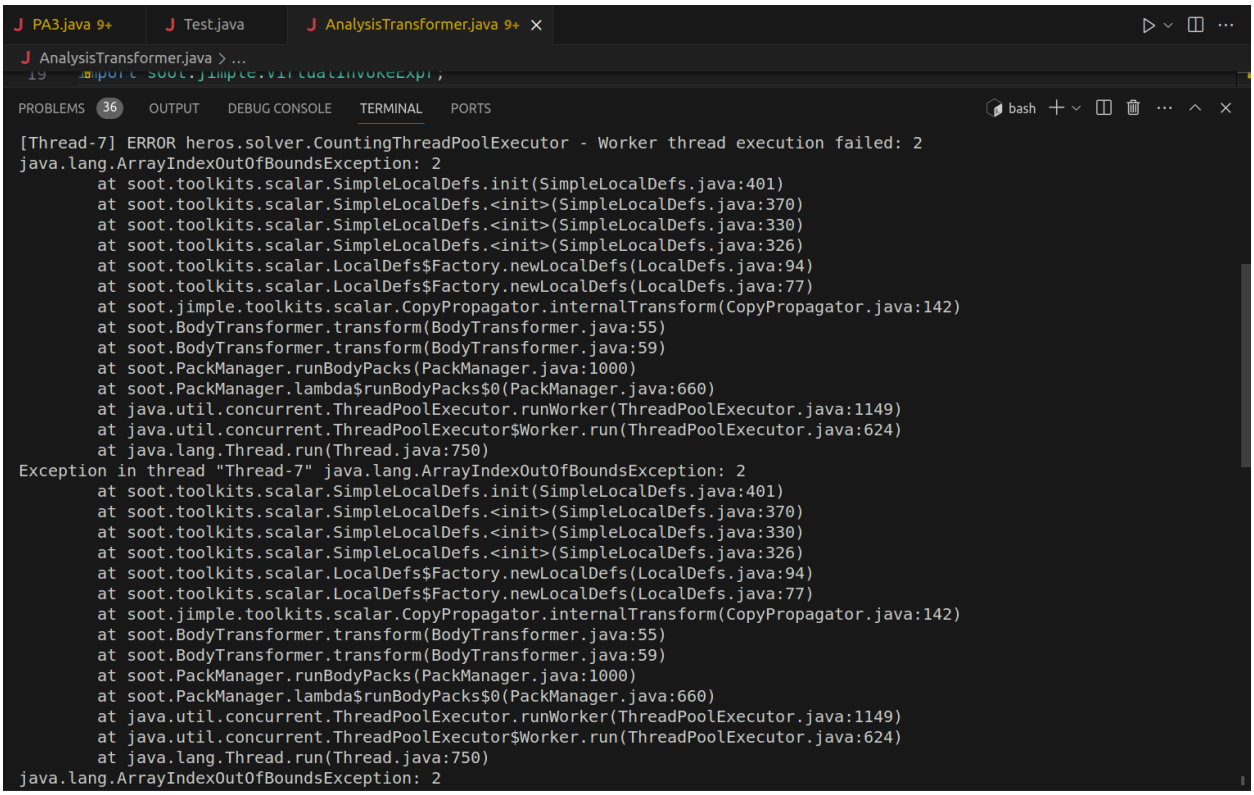


Figure 1: ArrayIndexOutOfBoundException

# 4 Generating class files

We used the following command on command-line and modified the jimple files to generate the modified class files :

Command - `java -cp sootclasses-trunk-jar-with-dependencies.jar soot.Main -src-prec jimple -f class -process-dir <PATH-TO-JIMPLE-DIR> -d <PATH-TO-OUTPUT-DIR> -verbose`

# 5 Test Cases

We have made two test cases to illustrate the performance improvement through our modification of class files. Both test cases make the use of loops to keep the program running for long time enough. All method calls are happening inside loop. Both modified and unmodified class files have virtualinvoke calls inside loop, with one virtualinvoke call in the modified class file replaced with staticinvoke call. One test case has the overridden method in two classes and another test case has overridden method in three classes. We made such test cases to see if increasing the number of potential classes to search in case of virtualinvoke call also increases the difference between cost of a virtualinvoke and a staticinvoke call.

Test cases have been included in the github repository whose link is provided at the end of this report.

# 6 Measuring the improvement

We used the 'time' command to measure the time difference between execution times modified class files and unmodified class files for different number of loop iterations and both test cases.

We used this command to run the class files on OpenJ9: time /Desktop/a4_coool/openj9-openjdk-jdk8/build/linux-x86_64-normal-server-release/images/j2sdk-image/bin/java -Xint Test1 Please adjust the path to OpenJ9 according to the local path on your system.

CSV files with the runtimes of both testcases for all iterations has been provided in the github repository. The graphs illustrating performance are shown below :

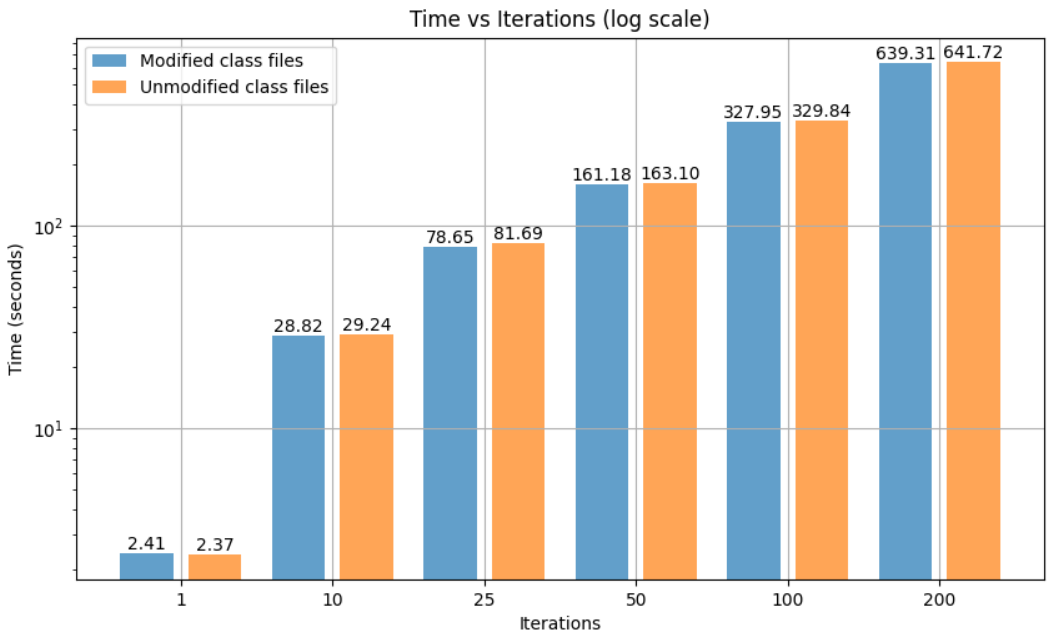


Figure 2: Results for test case 1

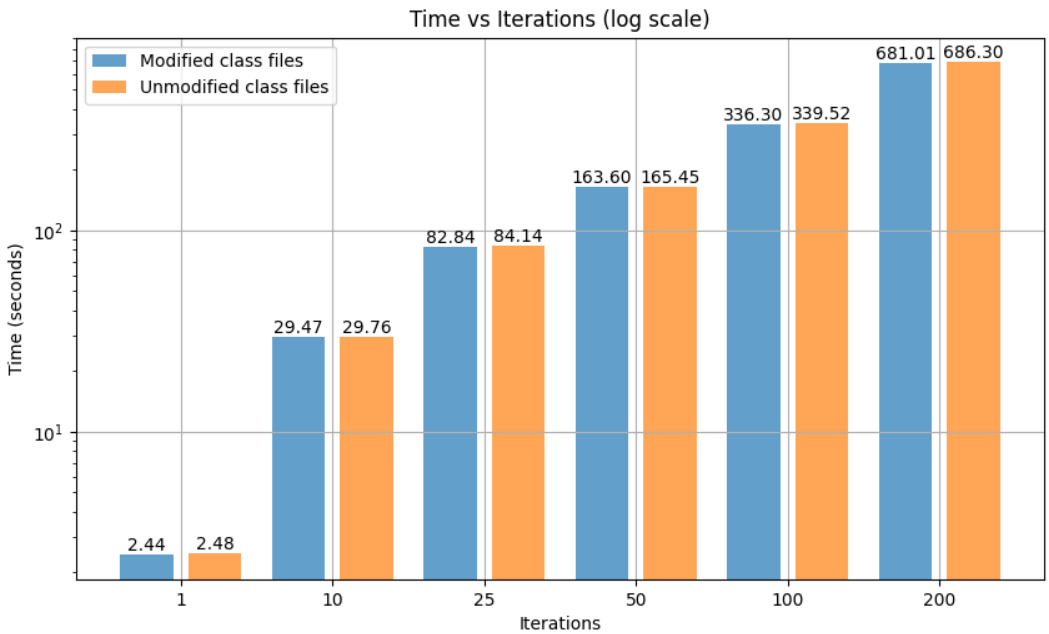


Figure 3: Results for test case 2

We were able to see upto 5 seconds of improvement in runtime in the bestcase (testcase 2 with 2x10$\hat{7}$ iterations), as illustrated in the screenshots from time command given below:

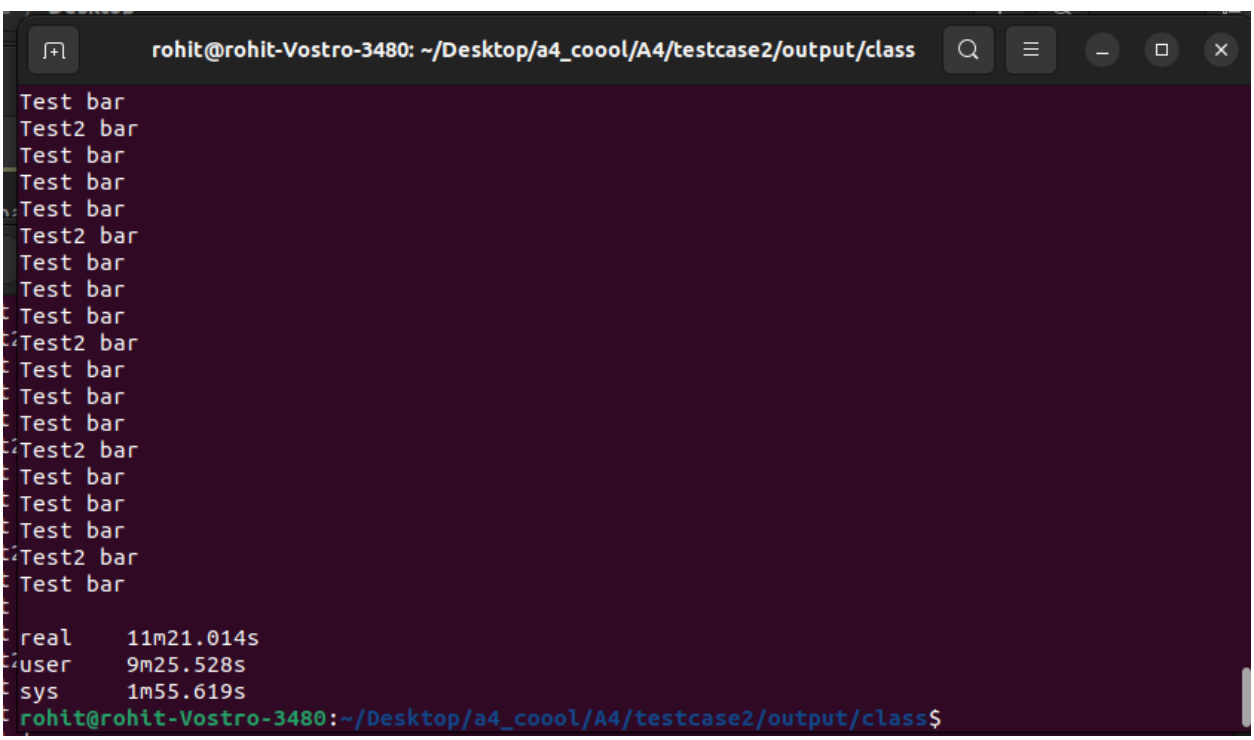


Figure 4: Run time for modified class file

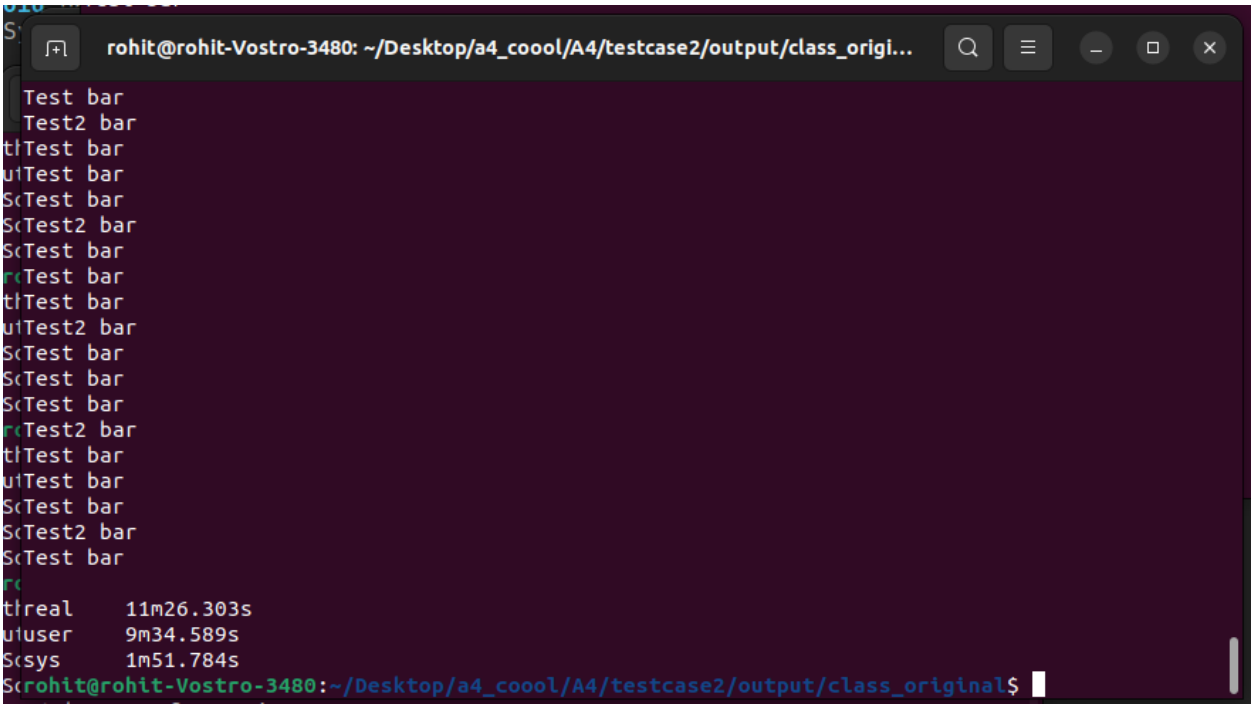


Figure 5: Run time for unmodified class file

# 7 Github repository

Here is the link of github repository of our code : Github link

Instructions to reproduce our results :

1. Download our github repository.

2. Choose a java test case from the folder 'test cases', copy it into the 'testcase' folder and compile it.

3. In the folder, run the following commands to see the output of our analysis.
   Command 1: javac -cp .:sootclasses-trunk-jar-with-dependencies.jar PA3.java
   Command 2: java -cp .:sootclasses-trunk-jar-with-dependencies.jar PA3

4. Our analysis would be printing out information about monomorphic call sites, and the structure of all classes after we tried to modify the code. But it would be giving the exception as stated earlier in the report, thus would not be able to generate transformed jimple files.

5. In the next step, we modified jimple files manually according to the output of above analysis.

6. After modifying jimple files manually, we converted them to class files using the command provided in section 4.

7. Run these class files as well as the unmodified class files using the commands described in section 6.