Compiler Optimization in Java

CS 6004: Compiler Optimization in Object-Oriented Languages

Master of Technology
Computer Science and Engineering

by

Dhruv Kathrotiya (23M0741)

Utsav Manani (23M0788)

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai 400 076

Spring 2024

Contents

List of Figures

Chapter 1

Optimiztaion Implemented

In the experiment, we have tried to implement a small part of loop optimization. There are many loop optimizations that can be performed on an uncompiled code. The main loop optimization techniques include the following:

- **Code Motion:** Code motion is used to decrease the amount of code in loop. This transformation takes a statement or expression which can be moved outside the loop body without affecting the semantics of the program.
- **Induction Variable Elimination:** It can reduce the number of additions in a loop. It improves both code space and run time performance.
- **Reduction in Strength:** Strength reduction is used to replace the expensive operation by the cheaper once on the target machine. The addition of a constant is cheaper than a multiplication. So we can replace multiplication with an addition within the loop.

We have tried to implement the **Code Motion** part in our experiment.

1.1 Code Structure

```
for (Loop loop : loopFinder.getLoops(body))

        head = loop.getHead();
        stmtsToRemove = new ArrayList<>();
        int i=0;
        for (Stmt unit : loop.getLoopStatements()) {
```

```
            loopInvariantCheck(stmtsToRemove, unit, loop, false);
            i++;
        }

        for(Stmt s: stmtsToRemove){
            units.remove(s);
            units.insertAfter(s, units.getPredOf(head));
        }

    }
```

The above code is the main part of processing every method. We get a call graph for the program from soot and parse every method in that call graph. Now we iterate over all the loops present in a method using the above code.

Now the first inner loop iterates over all the statements present in a loop and passes them to a function called **loopInvariantCheck**. The function takes 4 arguments, a list, a statement, a loop, and a boolean flag. This function contains the main logic about what statements can be removed from the loop and placed before it. This function populates the list "stmtsToRemove" with all the statements that can be removed from the loop.

The second inner loop iterates over all the statements from the list and removes them from the loop body and attaches them again before the head of the loop. So in essence the code identifies which statements can be moved outside the loop and places them in order before the start of the loop so that the semantics of the program doesn't get affected

The below function does the job of identifying what statements of the loop can be moved outside the loop.

```
protected static void loopInvariantCheck(List<Stmt>
stmtsToRemove, Stmt u, Loop l, boolean f){

   Value lhs,rhs;
   if(u instanceof JAssignStmt){
       lhs = ((AssignStmt)u).getLeftOp();
       rhs = ((AssignStmt)u).getRightOp();
       if(rhs instanceof JNewExpr || f){
```

```
                stmtsToRemove.add(u);
                int i=0;
                for(Stmt s: l.getLoopStatements()){
                    if(!f){
                        if(s == u) stmtsToRemove.add(
                        l.getLoopStatements().get(i+1));
                    }
                    if(s instanceof JAssignStmt){

                        if(((AssignStmt)s).getRightOp()
                        .toString().equals(lhs.toString())){
                            loopInvariantCheck(stmtsToRemove, s, l, true);
                        }
                    }
                    i++;
                }
            }
        }

    }
```

The logic of the function is simple. It checks for the assignment statements and particularly for the statements having RHS instance of NewExpr. So what we assume is if a new keyword is used and an object is made inside a loop and is stored to some variable and if it is not assigned to some variable used outside of the loop, then we can take that statement outside of the loop. So in general if rhs of any assignment statement has some static value or constant value, which is not being updated after the loop iteration and the lhs of the same assignment statement does not escape the loop then we can take such statements outside the loop.

The first if condition checks whether u is instance of JAssignStmt and if rhs is instance of JNewExpr then we add the statement into the list stmtsToRemove. Now we check if the lhs of the statement id used as rhs inside some other statements in the loop body, if so we also add that statement to the list of removable statements and recursively call the same function on the new statement that we added to the list.

Chapter 2

Experimental Results

we have tried manipulating the following code as an example code. The code contains creation of new node inside the loop and assigning its field some value. Here the New Node creation can be taken outside the loop.

```
    public class Test {

public static void main(String[] args) {

Node g = null;
for(int i=0;i<100000000;i++){
g = new Node();
g.i = 5;
}
System.out.println("g.i is "+g.i);
}
}

class Node{
Node n;
int i;
}
```

The following is the jimple file of the code without any changes to the loop.

```
    public static void main(java.lang.String[])
    {
```

```
        java.lang.StringBuilder $r0, $r2, $r3;
        java.io.PrintStream $r1;
        int $i0, i1;
        java.lang.String $r4;
        Node $r5, r7;
        java.lang.String[] r6;

        r6 := @parameter0: java.lang.String[];

        r7 = null;

        i1 = 0;

     label1:
        if i1 >= 100000000 goto label2;

        $r5 = new Node;

        specialinvoke $r5.<Node: void <init>()>();

        r7 = $r5;

        $r5.<Node: int i> = 5;

        i1 = i1 + 1;

        goto label1;

     label2:
```

In the above jimple code the statements

```
$r5 = new Node;
```

and

```
specialinvoke $r5.<Node: void <init>()>();
```

can be taken out of the loop to optimize the loop for efficient execution. After running AnalysisTransform on the following code the modified jimple file is as follows.

```
public static void main(java.lang.String[])
{
    java.lang.StringBuilder $r0, $r2, $r3;
    java.io.PrintStream $r1;
    int $i0, i1;
    java.lang.String $r4;
    Node $r5;
    java.lang.String[] r6;

    r6 := @parameter0: java.lang.String[];

    i1 = 0;

    $r5 = new Node;

    specialinvoke $r5.<Node: void <init>()>();

 label1:
    if i1 >= 100000000 goto label2;

    $r5.<Node: int i> = 5;

    i1 = i1 + 1;

    goto label1;

 label2:
```

Now we have converted the jimple files into class files using the following command.

```
java -cp sootclasses-trunk-jar-with-dependencies.jar soot.Main
-src-prec jimple -f class -process-dir <Soot-Output-Directory>
-d <Target-directory>  -verbose
```

The following shows the experimental outputs of the running time of both the programs using the time command in linux.

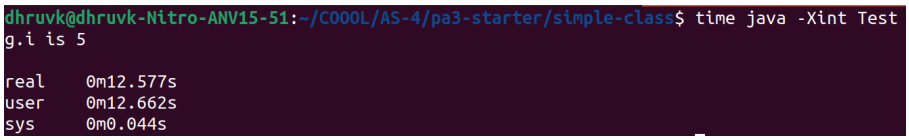

Figure 2.1: execution time of unmodified class file

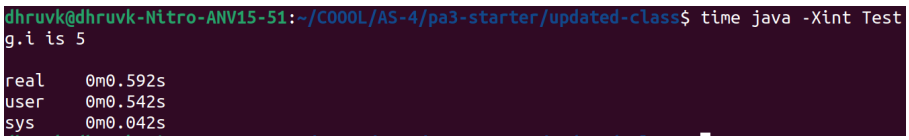

Figure 2.2: execution time of modified class file

This code is still at an experimental state and can be extended for including further testcases where rhs can be library call or in general a static value which doesn't change with the iterations of the loop.

The link for the Github repo - pa-4-code.