Phoenix Sun, Wenxuan Mei, Yangcheng Chi

COMP0012

Coursework 2

Compiler Group 26

13th April 2021

Compiler Coursework Part II: Code Optimization Report

After three weeks of learning, experimenting and exploring the structure and the principle of the bytecode optimization process of Java, our group has finally completed our own version of constant folder optimization. The code is primarily our own works and was partially inspired and guided by the Moodle examples and BCEL Manuel and APIdoc, the full website citation is included in the Page 10 Works Cited page. Below we will walk through all challenges and specific algorithms we have implemented into our codes.

Overall structure:

Our optimization's basic structure is that first, it reads all methods in the class and iterates and does the optimization on each method. In the method optimizeMethod, first, it reads in the bytecode instructions of the method, and then it initializes a method generator with the original method as the baseline. After initialization, it applies simple folding, constant variable folding, and dynamic variable folding separately on the current optimizing method. After the method is optimized, it would replace the original method in the class, and the optimization process is finished.

Simple Folding:

Algorithm:

To store constants, we created a constant stack of Number objects. An ArrayList of InstructionHandle is also created to store the instructions that need to be deleted as a remove list because optimized instructions replace them. First, we iterate the list of bytecode instructions. We only check the simplest case where two continuous LDCs instruction followed with an arithmetic instruction, LCMP or ifInstruction, or there is a conversion between them.

If the condition is met, we check if the instruction is the following cases:

1.If it is LDC or LDC2_W or constant push instruction, we will get the constant value from the instruction and push the constant into the constant stack. The instruction is added to the remove list as the following part of the folding method will perform calculation later. The result of the calculation will replace the unoptimized value.

2.If it is an arithmetic operation, we will check the size of the constant stack. If the stack is less than 2, there is no need to perform any arithmetic operations, so the instruction is skipped and the remove list is cleared as no optimization is performed. Otherwise, the arithmetic operation is performed between the two constants based on the types of the constant(int, long, float, double) and the operation(Addition, Subtraction, Multiplication, Division). The two constants will be popped from the stack, and the result of the calculation is then added to the constant stack. Since the optimization, the previous LDC or constant push instructions in the remove list are deleted from the instruction list. The calculated value is then popped from the stack and added to the instruction list. Therefore, the old instructions are replaced by the new instructions, then optimization of an arithmetic operation is finished.

3.If it is a conversion instruction, it will then be deleted because the result is already calculated, and the instruction is no longer needed.

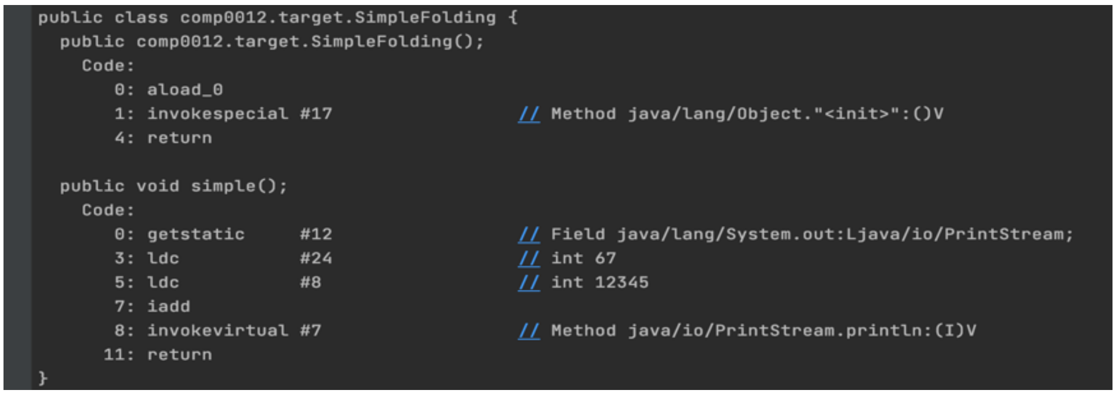
4.If it is an IfInstruction, we will perform the logic operation. First, we check the constant stack size to make sure operation is performed when the size of the stack is 2. However, there is an exception which is IFLE, as this operation only takes one constant. After this, we check whether there is a goto instruction. No goto instruction means that the "else" branch of the "if" does not exist. In that case, we will perform the logic operation on the constants in the constant stack. If the comparison succeeds, IfInstruction is removed. Otherwise, the if part is removed. Otherwise, if there is an "else" branch, we will perform the logic operation, and if it is true, the if part is reserved and else part is removed. If it is false, the else part is reserved and if part is removed. After the result is calculated, instructions in the remove list are removed.

5.If it is an LCMP where the comparison between long type constant occurs, the stack size is again checked. The result of the comparison is then pushed into the constant stack and previous instructions are removed from the remove list.

The result of arithmetic or logic calculation will replace the formula at the end of the method and simple folding optimization is finished.

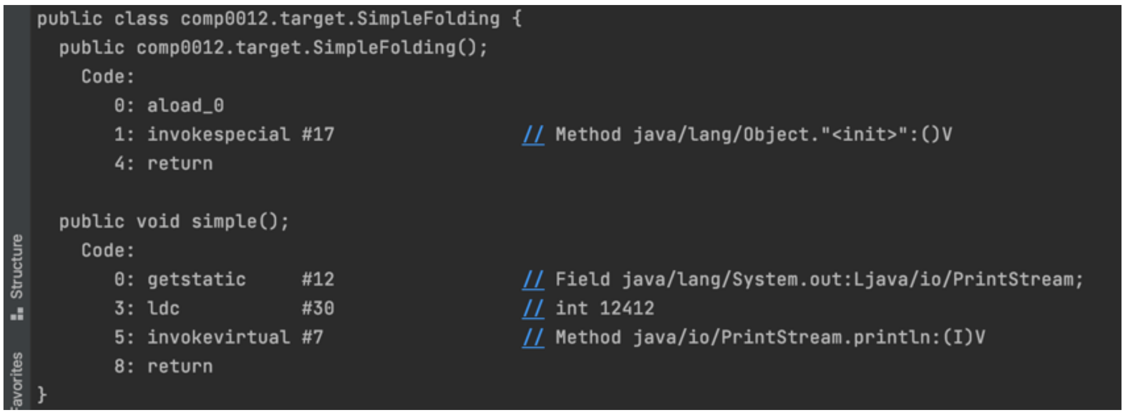
Example outcome:

Before optimized:



```
public class comp0012.target.SimpleFolding {
  public comp0012.target.SimpleFolding();
    Code:
       0: aload_0
       1: invokespecial #17                 // Method java/lang/Object."<init>":()V
       4: return

  public void simple();
    Code:
       0: getstatic     #12                 // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #24                 // int 67
       5: ldc           #8                  // int 12345
       7: iadd
       8: invokevirtual #7                  // Method java/io/PrintStream.println:(I)V
      11: return
}
```

After optimized:



```
public class comp0012.target.SimpleFolding {
  public comp0012.target.SimpleFolding();
    Code:
       0: aload_0
       1: invokespecial #17                 // Method java/lang/Object."<init>":()V
       4: return

  public void simple();
    Code:
       0: getstatic     #12                 // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #30                 // int 12412
       5: invokevirtual #7                  // Method java/io/PrintStream.println:(I)V
       8: return
}
```

As the bytecode shows, 2 LDC with value 67 and 12345 and iadd instructions are replaced by the LDC with value 12412. Simple folding optimization is applied.

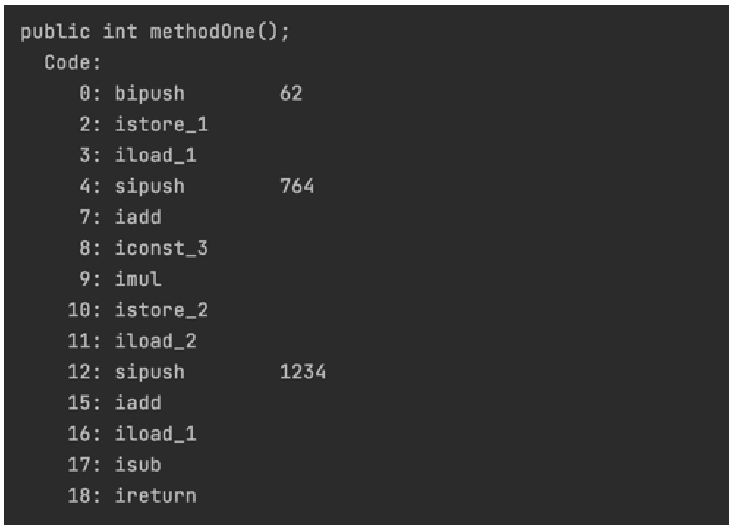
Constant Variable Folding:

Algorithm:

To do constant variable folding, we first create two hash maps to store the literal value of the variable and track whether the variable is constant. First, we check whether the variable is constant. We use an Iterator and InstructionFinder to search StoreInstruction and IINC instruction. If we find one, the getIndex method is used to get the variable's index from the instruction. If the index did not appear in the hash map, then put this index in the hash map to be true to mark it as a constant variable. Otherwise, this index is marked as false to show it appeared before and it will be marked as a dynamic variable. After locating all constant variables, a loop is implemented to reduce and simplify loadInstruction step by step until there are no load instructions and all constant variables are not invoked. Inside the loop, simple folding is first applied to get as many literal values as possible. Then instructionFinder and Iterator is used to search specific instruction patterns of pushInstruction followed by storeInstruction. If such a pattern is found, we first check whether the variable index obtained by the storeInstruction is mapped firmly in the hash map to check if the variable is constant. The literal value from the pushInstruction is obtained and mapped with the variable index in the literal value hash map. After assigning variables with their literal value, the next step is to search where the variable is called by searching loadIstructions and replace the variable with its literal value. As before, InstructionFinder and Iterator are used to search loadInstruction. We will check if the variable index obtained from the instruction is mapped with a literal value if one is found. Suppose the literal value is used to add new instructions and previous load instructions are deleted. In that case, the variables are replaced with their literal values and constant variable folding optimization is finished.

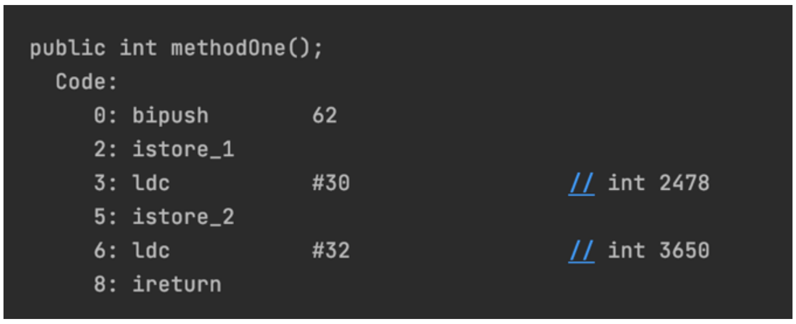
Example Outcome:

Before optimized:



```
public int methodOne();
  Code:
     0: bipush        62
     2: istore_1
     3: iload_1
     4: sipush        764
     7: iadd
     8: iconst_3
     9: imul
    10: istore_2
    11: iload_2
    12: sipush        1234
    15: iadd
    16: iload_1
    17: isub
    18: ireturn
```

After optimized:



```
public int methodOne();
  Code:
     0: bipush        62
     2: istore_1
     3: ldc           #30                 // int 2478
     5: istore_2
     6: ldc           #32                 // int 3650
     8: ireturn
```

As the bytecode shows, the arithmetic operation like iadd is replaced by the result of calculation by simply folding, and load instructions are replaced by the literal value of the loaded variable. In the end, constant folding is applied.

Dynamic and constant variable folding:

Algorithm:

Our dynamic variable folding also includes a constant variable folding and the bytecode instructions are more simplified than that of single constant variable folding. Therefore, we decided to drop the previous single constant variable folding method as dynamic variable folding has already done constant variable folding.

First, we created two hashmaps to record each variable's literal value and check if a variable is called in a loop. To check whether a variable is called in a loop, we first check the goto instruction target. If this target's position is before the goto instruction, this goto is the goto in a loop. If the target is a load instruction, put its index as a key in the Hashmap, so that any load and store instruction with this index will not be considered. Furthermore, the LDC or Constantpush that are used for these store instructions will also not be considered. After that, two stacks are created to store constants and instructions that need to be removed. Also, a loop is implemented to remove store and load instructions repeatedly. When all load and store instructions are removed and replaced with new instructions, optimization is finished. In the loop, we first check the situation where LDCs are followed with a store instruction and whether it is in a loop.

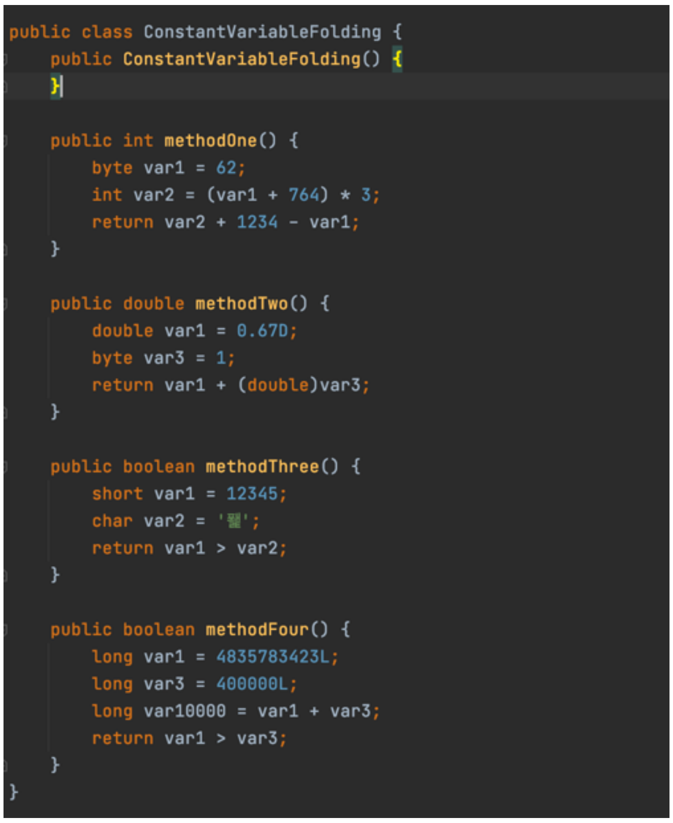
If the current instruction is LDC and the next instruction is stored and not in a loop, The constant stack will store the value of LDC, and the instruction is then pushed into the removal stack. If the current instruction is storeInstruction, we first check the constant stack's size to check if simple folding is applied. If the size is 0, we will do simple folding first and break the loop to start a new one. Otherwise, we will pop the value from the constant stack and use a hashmap to map the value pointed by the storeInstruction to the variable. If the variable exists in the Hashmap, its value is then replaced with the new value. If it is not, then the variable with the value is added to the Hashmap. And then previous LDC instruction and store instruction are all removed. If the current instruction is LoadInstruction, we will search the value of variable LoadInstruction refers to and then the loadInstruction is replaced with the LDC value of the variable.

After all store and load instructions are removed, a simple folding is applied. Then targets in the loop are redirected to make sure the ifInstruction does not lose its target when some instructions are removed. To do that, we will record the goto instructions and find the ifInstruction they are referring to. Then we will replace the old ifInstruction with a new one targeting the instruction next to the goto. After this, the dynamic variable optimization is finished.

Example outcome:

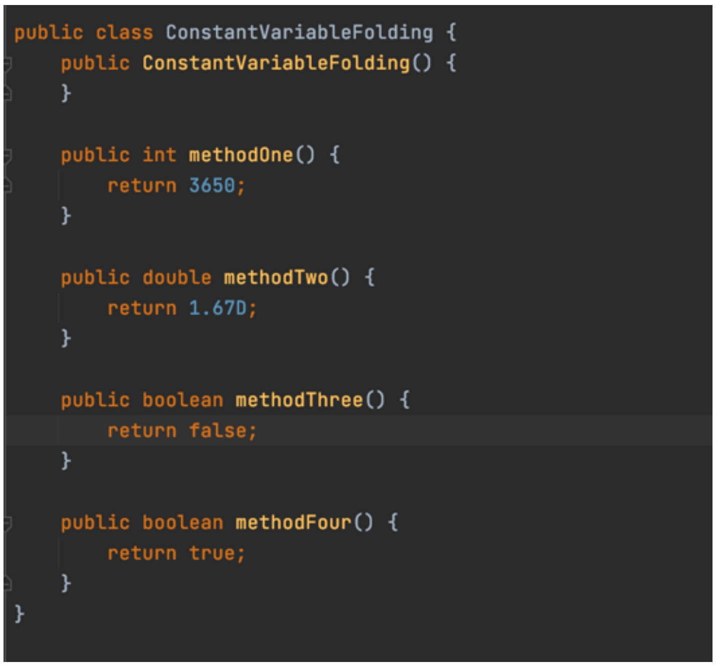
The difference in constant folding:

The constant variable folding method in constant variable folding test:



```
public class ConstantVariableFolding {
    public ConstantVariableFolding() {
    }

    public int methodOne() {
        byte var1 = 62;
        int var2 = (var1 + 764) * 3;
        return var2 + 1234 - var1;
    }

    public double methodTwo() {
        double var1 = 0.67D;
        byte var3 = 1;
        return var1 + (double)var3;
    }

    public boolean methodThree() {
        short var1 = 12345;
        char var2 = '[illegible]';
        return var1 > var2;
    }

    public boolean methodFour() {
        long var1 = 4835783423L;
        long var3 = 400000L;
        long var10000 = var1 + var3;
        return var1 > var3;
    }
}
```

The dynamic variable folding method in constant variable folding test:

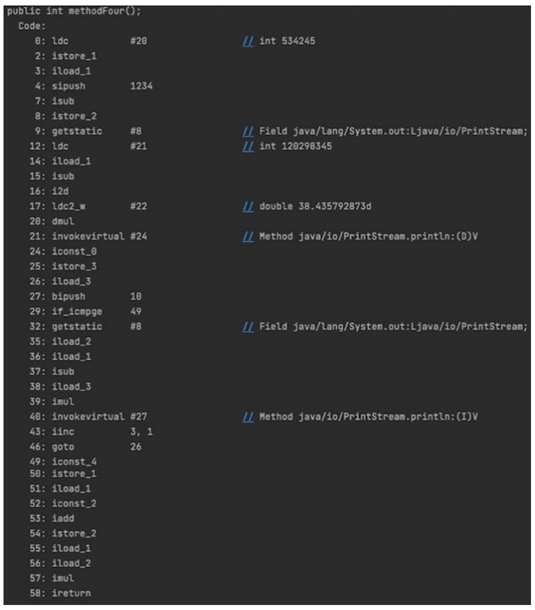


```
public class ConstantVariableFolding {
    public ConstantVariableFolding() {
    }

    public int methodOne() {
        return 3650;
    }

    public double methodTwo() {
        return 1.67D;
    }

    public boolean methodThree() {
        return false;
    }

    public boolean methodFour() {
        return true;
    }
}
```

As the class file shows, all store instructions are removed in the dynamic variable folding method. Therefore, all the variable assignments are removed.

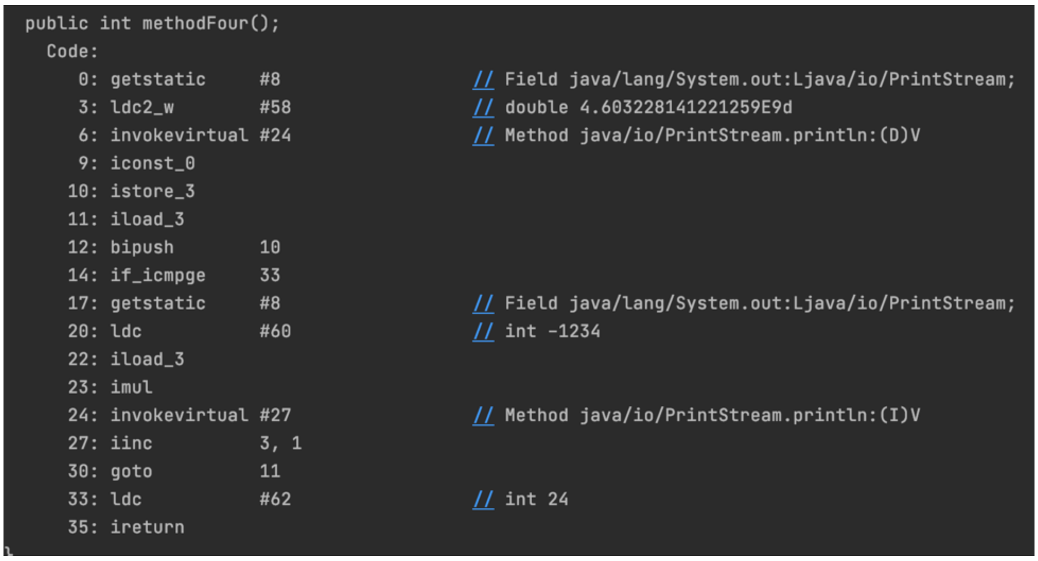
Dynamic variable folding:

Before optimization:



```
public int methodFour();
  Code:
     0: ldc           #20                 // int 534245
     2: istore_1
     3: iload_1
     4: sipush        1234
     7: isub
     8: istore_2
     9: getstatic     #8                  // Field java/lang/System.out:Ljava/io/PrintStream;
    12: ldc           #21                 // int 120298345
    14: iload_1
    15: isub
    16: i2d
    17: ldc2_w        #22                 // double 38.435792873d
    20: dmul
    21: invokevirtual #24                 // Method java/io/PrintStream.println:(D)V
    24: iconst_0
    25: istore_3
    26: iload_3
    27: bipush        10
    29: if_icmpge     49
    32: getstatic     #8                  // Field java/lang/System.out:Ljava/io/PrintStream;
    35: iload_2
    36: iload_1
    37: isub
    38: iload_3
    39: imul
    40: invokevirtual #27                 // Method java/io/PrintStream.println:(I)V
    43: iinc          3, 1
    46: goto          26
    49: iconst_4
    50: istore_1
    51: iload_1
    52: iconst_2
    53: iadd
    54: istore_2
    55: iload_1
    56: iload_2
    57: imul
    58: ireturn
```

After optimization:



```
public int methodFour();
  Code:
     0: getstatic     #8                  // Field java/lang/System.out:Ljava/io/PrintStream;
     3: ldc2_w        #58                 // double 4.603228141221259E9d
     6: invokevirtual #24                 // Method java/io/PrintStream.println:(D)V
     9: iconst_0
    10: istore_3
    11: iload_3
    12: bipush        10
    14: if_icmpge     33
    17: getstatic     #8                  // Field java/lang/System.out:Ljava/io/PrintStream;
    20: ldc           #60                 // int -1234
    22: iload_3
    23: imul
    24: invokevirtual #27                 // Method java/io/PrintStream.println:(I)V
    27: iinc          3, 1
    30: goto          11
    33: ldc           #62                 // int 24
    35: ireturn
```

As the result shows, the store and load instructions are removed and replaced with the LDC value. The GOTO instruction is redirected to prevent target loss.

Conclusion:

In conclusion, in simple folding, used LDC and arithmetic and logic operations are replaced with the calculated LDC value. Dynamic folding replaces store and load instructions with the literal values of the variables. As a result, constant folding is more simplified compared to the single simple folding method.

Team contribution:

Wenxuan and Phoenix implement simple folding; Phoenix implements constant variable folding, all team members implement dynamic folding, and the report is mainly written by Jackson. All team members contributed to the editing of the report.

Works Cited

Apache Commons BCEL™ – The BCEL API, commons.apache.org/proper/commons-bcel/manual/bcel-api.html.

“BCEL Class APIdoc” Index of /Proper/Commons-Bcel/Apidocs/Org/Apache/Bcel, 2004, commons.apache.org/proper/commons-bcel/apidocs/org/apache/bcel/.