

Compilers Optimisation Coursework Report

Group Members: Galen Han, Alexander Xu, Jamie Law

Overview

This report will deal with explaining the core algorithms and heuristics of our optimisation methods for implementing constant folding optimisation on Java bytecode instructions.

Core Algorithms

Initialisation

After parsing the class file which is to be optimised, we obtain the methods from the ClassGen. We then iterate through each method and call upon our *optimiseMethod()* function.

optimiseMethod()

In this section, we perform our optimisation by iterating over each method of the class (method one, method two etc.).

For each method we get an array of instruction handles. A method generator is initialised with the original method as the baseline (in preparation for creating the optimised method).

We apply peephole optimisations on the instruction list repeatedly, until a single iteration is unable to find any additional optimisations. This is done by using a counter and a while loop to perform **arithmetic** and **comparison** optimisations. This counter is incremented by the method return types; arithmetic optimisation and comparison optimisation both return an integer indicating how many optimisations have been made. This counter is reset every time the loop is re-entered. If the counter stays at 0 after an iteration, then the loop is exited as this indicates that no more optimisations can be made.

Following this, we ensure jump handles are all within the current method. Then, we set max stack/max locals using the aforementioned method generator. The method is replaced in the original class with optimised instructions by getting a new method from the method generator and replacing the original method with the optimised method.

optimiseArithmeticOperations()

This method handles the peephole optimisation for arithmetic operations. A counter is used to indicate how many instructions have been optimised (needed in *optimiseMethod()*). In order to identify an arithmetic operation, we use the *InstructionFinder* in BCEL with the following regular expression:

```
(ConstantPushInstruction|LDC|LDC2_W|LoadInstruction)
(ConversionInstruction)?

(ConstantPushInstruction|LDC|LDC2_W|LoadInstruction)
(ConversionInstruction)?
ArithmeticInstruction
```

This regular expression matches all arithmetic operations (multiplication, division, addition and subtraction). There are additional clauses for optional *ConversionInstruction*, which occur in the bytecode when constants of different types are operated on.

Once an arithmetic operation has been matched, we have an array of *InstructionHandle*. The instructions (left instruction, right instruction and operation instruction) are obtained first by allocating each instruction to an index of the array, which is dependent on whether there is a conversion for the left and right instructions. We can then obtain the actual operation by using *getInstruction()* from the operation instruction. From the left and right instructions, we use our *ValueLoader* to get the respective values (see [Utility Algorithms](#) below).

In order to preserve the semantic meaning of the instructions, we check if the left or right value is located in a for loop using the [utils.ForLoopChecker.checkIfForLoop\(InstructionHandle h\)](#) method. The algorithm for checking if the given *InstructionHandle* is manipulated in a for loop is described later on in this report. If either the left or right value is in a for loop, we are unable to fold the constant value in, as it is a variant in the loop. We therefore skip to the next match of *InstructionFinder*.

If the value of *leftInstruction* and *RightInstruction* are not affected in any for loop, we fold the value of the two by using the *Utilities.foldOperation(ArithmeticInstruction operation, Number left, Number right)*.

This method will perform the respective operation on the two *Number* parameters and return the value as a *double*.

The result will then be inserted into the constant pool using *ConstantPoolInserter.insert(Double value, char type, ConstantPoolGen cpngen)*. It will ensure that *double* is converted to the correct type, depending on the highest hierarchy of the *leftInstruction* and *rightInstruction*, in accordance with the Java specification for type promotion. E.g. if the *leftInstruction* was an **ILOAD** and right instruction was an **FLOAD**, the folded value would be inserted into the pool as a float using the *addFloat()* method.

Note that each time we fold a value, a new constant is inserted into the constant pool. This simplifies the implementation of dynamic variable folding significantly. For a real-world optimiser we would check the constant pool for unused constants that have no reference to them, after all peephole optimisations have been made and delete them to save memory space and reduce the size of the program, however, this is outside the scope of the assignment.

Finally we delete the now redundant load instructions and the **ArithmeticOperation**, replacing it with a single **LDC** or **LDC2_W** instruction to load the constant from the pool.

We loop through any additional pattern matches and return the number of optimisations made.

optimiseComparisons()

This method handles the peephole optimisation for comparisons. A counter is used to indicate how many instructions have been optimised (needed in *optimiseMethod()*). In order to identify a comparison, we use the *InstructionFinder* in BCEL with the following regular expression:

```
(ConstantPushInstruction|LDC|LDC2_W|LoadInstruction)
(ConversionInstruction)?
```

```
(ConstantPushInstruction|LDC|LDC2_W|LoadInstruction)
(ConversionInstruction)?
```

```
(LCMP|DCMPG|DCMPL|FCMPG|FCMPL)?
IfInstruction
(ICONST GOTO ICONST)?
```

This regular expression matches all comparisons by identifying two instructions (for getting values) and an *IfInstruction*. There are additional clauses for optional *ConversionInstruction* and non integer comparisons (*LCMP*, *DCMPG* etc.), which occur in the bytecode when constants of different types are operated on. It also accounts for constant comparisons e.g. ($x < y$) with the (*ICONST GOTO ICONST*)?.

Once a comparison has been matched, we have an array of *InstructionHandle*. We use a similar method to *optimiseArithmeticOperations()*. The instructions (left instruction, right instruction and comparison instruction) are obtained first by allocating each instruction to an index of the array, which is dependent on whether there is a conversion for the left and right instructions. For the comparison instruction, we also need to identify whether there is a non-integer comparison before the *IfInstruction*. We can then obtain the actual comparison by using *getInstruction()* from the comparison instruction. From the left and right instructions, we use our *ValueLoader* to get the respective values (see [Utility Algorithms](#) below).

The same method for identifying for loops (in *optimiseArithmeticOperations()*) is implemented here. In order to preserve the semantic meaning of the instructions, we check if the left or right value is located in a for loop using the *utils.ForLoopChecker.checkIfForLoop(InstructionHandle h)*

method. The algorithm for checking if the given *InstructionHandle* is manipulated in a for loop is described later on in this report. If either the left or right value is in a for loop, we are unable to fold the constant value in, as it is a variant in the loop. We therefore skip to the next match of *InstructionFinder*.

If the value of *leftInstruction* and *RightInstruction* are not affected in any for loop, we use the *ComparisonChecker* to obtain the result from the comparison (two comparisons are needed for non-integer comparisons due to the conversion comparison, *checkFirstComparison()* and *checkSecondComparison()*).

The result will then replace the left instruction to point to the new index. Due to the instruction interpretation, 0 and 1 are inverted.

Finally, we use the result to delete dead code instructions. If the result returns true, then the if statement (*ComparisonInstruction*) is deleted while retaining the instructions within. Otherwise, the whole set of instructions is deleted (as this if statement would never be entered, with the exclusion of comparing dynamic variables in loops, which have been dealt with as mentioned previously). This also includes deleting instructions within else statements accordingly, if any exist.

We continue to loop through any additional pattern matches and return the number of optimisations made.

Utility Algorithms

utils.ForLoopChecker.checkIfForLoop(InstructionHandle h)

This method allows us to check whether a variable is dynamic, such as a variable used to iterate through a for-loop. This is done by passing a load instruction we intend to check into the method, then iterating through the list of preceding instructions to identify a GOTO instruction. When a GOTO instruction is found, the instruction previous to the GOTO instruction is checked to identify whether it is an increment or store instruction (which likely indicates that the variable is a variant in a loop). If all conditions pass as true, we fetch the target of the GOTO instruction.

From here, we can determine whether the GOTO instruction targets the load instruction we are checking. If so, the method returns true to indicate that the variable from the load instruction is indeed being manipulated inside a for loop

and no further folding will occur. Otherwise, the method will return false and folding operations will continue as normal.

This prevents the following code to be folded incorrectly:

```
int a = 534245;
int b = a - 1234;
for(int i = 0; i < 10; i++){
    System.out.println((b - a) * i);
}
```

We correctly detect that variable *i* is a variant in the loop and thus will correctly fold it from `System.out.println((b - a) * i);` to `System.out.println(-1234 * i);`

public static Number ValueLoader.getValue(InstructionHandle h, ConstantPoolGen cpGen) throws UnableToFetchValueException

This method allows us to fetch the value of any instruction that pushes a numerical value onto the stack. We distinguish whether the instruction is a constant value or a load instruction.

If the instruction has its value stored in the constant pool, i.e. it is an instance of *LDC* or *LDC2_W*, we simply load the value stored in the constant pool and return it as a *Number*.

It becomes slightly more complicated when we need to get the value for a *LoadInstruction* such as *ILOAD*. We exhaustively iterate over previous *InstructionHandles* until we find a *StoreInstruction* such as *ISTORE*. While we iterate, if we find any *IINC* instructions along the way, we check if the *IINC* instructions are within a for loop, again with the *utils.ForLoopChecker.checkIfForLoop(InstructionHandle h)* method. If the *IINC* instruction is in a for loop, we throw an *UnableToFetchValueException*, as this means that the value does not remain constant. Otherwise, if the *IINC* is not in a for loop, we get the value of the increment and this to an accumulator value that is initialised to 0. If we finally find a *StoreInstruction*, we get the value that is pushed onto the stack from the previous instruction and return the result of this value and the accumulator value.

public static int

ComparisonChecker.checkIntComparison(IfInstruction comparison, Number leftValue, Number rightValue)

This method returns 0 or 1 by identifying the IfInstruction comparison type (for integers only) and performing the respective comparison with the *intValue()* of the left and right values.

public static int

ComparisonChecker.checkFirstComparison(InstructionHandle comparison, Number leftValue, Number rightValue)

This method returns -1 or 1 (or 0 for long comparison) by identifying the comparison type (distinguishing between doubles, floats and longs) and performing the respective comparison with the *doubleValue()/floatValue()/longValue()* with the left and right values. For these non-integer type comparisons, they include *DCMPG*, *DCMPL* (Greater than, less than for doubles) *FCMPG*, *FCMPL* (Greater than, less than for floats) which would return -1 or 1 accordingly. However, for long comparison *LCMP*, it also identifies whether values are equal, which would return a value of 0.

public static int

ComparisonChecker.checkSecondComparison(IfInstruction comparison, int value)

This method is similar to *checkIntComparison()*, except it is a follow-up from non-integer comparisons (*checkFirstComparison()*). This method takes in the value from the first comparison and the IfInstruction comparison type to return 1 or 0 depending on the value.