COMP207P Compilers – Coursework Part II: Code Optimisation

Simple Folding

For this sub-goal we had to perform constant folding for values of types int, long, float and double in the bytecode constant pool. This was done through code in the optimizeMethod method in the ConstantFolder.java file.

First, we iterated over every single InstructionHandle in the method and checked the instance of the instruction to filter the results by operation. There were six different operations for each type, 24 in total, that our code handled. The operations were addition, subtraction, multiplication, division, remainders and negation. The names of these instances were easily understandable such as IADD for integer addition and FDIV for float division. Once we have the InstructionHandles of the operators, we obtained the previous two instruction handles and created methods (getIntValue, getFloatValue, etc) to obtain their values. The operation is then performed on the values and the result of the operation is then inserted as a new instruction into the instruction list. An LDC is used for an int and float whilst LDC2_W is used for a long or double.

The changes in the Java Bytecode from doing this can be seen when running the javap command. In the original build class, five different instructions (getstatic, ldc, ldc, iadd, invokevirtual) were required to add up two values and print them out. This was done by first passing two ldc parameters and then adding them together before printed. However, in the optimised version, this only required three instructions (getstatic, ldc, invokevirtual) to print out the value. Only one ldc with the value being the sum of the two original ldc values in the original build was used in the optimised version. This was then printed out.

Constant Variable Folding

In this sub-goal the aim was to optimise the local variables of type int, long, float and double whose values does not change throughout the scope of the method after declaration. This was done by propagating the initial value of the variable throughout the method to allow constant folding of variables.

When constants are saved into local variables i.e. int x = 5; Java will represent this as iconst_5, istore_1. When the variable is called it is called as iload_1. So in Java Bytecode we removed iload instructions and instead load the value of the variable directly from the constant pool and perform operations on it directly. For example, iconst_5, istore_1, <other code>, iload_1 becomes iconst_5, istore_1, <other code>, ldc (5). We managed this by, whenever we encounter an iload instruction, we went through all the instructions and checked for an istore instruction with the same index, and obtained the value from the previous instruction.

This now constant value can then be folded for operations using the method we used in simple folding. In the Java Bytecode, the length of the methods were shortened significantly by removing the need to constantly push, store and load variables with their operators. Instead through constant folding, the results of each equation were calculated first and then stored as an ldc, negating the need to store every value obtained from each step of the equation. For example, method one was shortened from 12 bytecode lines in the original version to 6 bytecode lines in the optimised version.

Dynamic Variable Folding

For this sub-goal we had to optimise the uses of local variables whose values will be reassigned with a different constant during the scope of the method. To make this work, when loading the value of an iload instruction, we have to modify our code to check the last istore assignment of the same index before the istore instruction. This ensures that the value being loaded is the value of the variable for the specific interval of the iload instruction. We also had to consider iinc increments.

When a variable is modified e.g. int x = 5; x = 2 + x; what normally would happen is: iconst_5, istore_1, iload_1, iconst_2, iadd, istore_1. In our optimisation method, we would perform the arithmetic operation and push the result to the stack then store it in the variable instead. Using the example above, our optimisation would output the following bytecode instead: iconst_5, istore_1, ldc (7), istore_1, shortening it by two bytecode lines.

A major challenge that we had was in the case of loops, where we had to update the StackMapTable and avoid folding variables which will be modified inside the same loop (e.g. for loop iterators). We managed the second part by creating a method checkLoopModification which checks whether the value of an index being loaded by an iload instruction is within a loop and being modified by another instruction within the same loop (we detect loops by detecting goto instructions that point backwards). For the StackMapTable we modified our code to get the StackMapTable from the methodGen and modified the delta offset values based on the targets of branch instructions within the optimised code.