COMP207P Coursework 2 – Optimisation Report

Group 27 Taolun Li and Xiangyi Yin

Optimisation starts with the optimize method which loops through all methods of the class and optimizes each using the optimizeMethod method.

First Steps

Loops

In the optimizeMethod(), before any optimization is done, our code will first check the method for any loops that may exist. This is done by traversing the InstructionList to find all GOTO instructions that target an instruction which is before the GOTO instruction. Furthermore, the loops are checked every time when optimizations are made as the position of the loop may be changed.

GOTO Instructions

If the GOTO instruction is targeting an instruction before itself, we would consider that there is a loop in the method. The position of the loop is stored in an ArrayList called loops.

If the GOTO instruction is targeting an instruction after itself, it is not a loop. In this case, if the GOTO instruction is targeting an IINC instruction, we would do the following:

1. Insert an instruction that pushes the increment amount onto the stack (i.e. BIPUSH #inc) before the IINC instruction.
2. Insert a LoadInstruction that loads the value from the variable with the index in the IINC instruction before the IINC instruction.
3. Insert an IADD instruction before the IINC instruction.
4. Insert a StoreInstruction that stores the value to the variable with the index in the IINC instruction before the IINC instruction.
5. Update the target of the GOTO instruction to the first inserted instruction (the BIPUSH instruction).
6. Delete the IINC Instruction.

Handle IINC instructions

After loops are found and any IINC targeted by a GOTO instruction that is before the IINC instruction is processed as shown above, all other IINC instructions that are not in the loop are optimised as follows:

The InstructionList is looped through and if an IINC instruction is reached, its index is passed into the method getPrevVal which searches the InstructionHandles above to find the nearest StoreInstruction/IINC instruction that contains the same index. If an IINC instruction is found, nothing is done back in the loop that called the getPrevVal method and the program breaks out of the loop. If a StoreInstruction is found, the code looks at the instruction before the StoreInstruction. If that instruction is any instruction that pushes a direct value to the stack (i.e. ICONST, BIPUSH, LDC, etc.), that value is returned. If it is not, then null is returned. Back in the loop that called the getPrevVal method, if the return value is not null, the following step is taken:

1. Add the increment amount of the IINC instruction to the value returned by the getPrevVal method.
2. Depending on the magnitude of the result from step 1, insert a BIPUSH/SIPUSH/LDC instruction with that result before the IINC instruction.
3. Insert a ISTORE instruction with the index of the IINC instruction before the IINC instruction.
4. Redirect any references to the IINC instruction to the instruction inserted at step 2.
5. Delete the IINC instruction.

If the return value is null, meaning no StoreInstruction can be found above or that there is no instruction above that StoreInstruction that directly pushes a value onto the stack, the following steps are taken:

1. Insert an instruction that pushes the increment amount onto the stack (i.e. BIPUSH #inc) before the IINC instruction.
2. Insert a LoadInstruction that loads the value from the variable with the index in the IINC instruction before the IINC instruction.
3. Insert an IADD instruction before the IINC instruction.
4. Insert a StoreInstruction that stores the value to the variable with the index in the IINC instruction before the IINC instruction.
5. Redirect any references to the IINC instruction to the instruction inserted at step 1.
6. Delete the IINC Instruction.

Then

After the first steps, the program will loop through the modified InstructionList in order to search for more optimisation opportunities.

Handle StoreInstructions and LoadInstuctions

When a StoreInstruction is reached, our program would first check the instruction above the StoreInstruction to see if it is an instruction that provides a direct value (e.g. BIPUSH, LDC, ICONST). If not, nothing else is done and the program moves on to the next instruction in the InstructionList. If yes, pass that value and the index of the StoreInstruction to the method convertLoadInst which attempts to convert LoadInstructions of the same index to a simple BIPUSH/SIPUSH/ICONST/DCONST/FCONST/LCONST/LDC/LDC2_W. The method convertLoadInst would go through all InstructionHandles after the StoreInstruction.

Case without a loop in the code:

Until the next StoreInstruction or an IINC instruction with the same index is found, all the LoadInstructions of the same index in the middle are replaced with a push instruction (BIPUSH/SIPUSH/ICONST/DCONST/FCONST/LCONST/LDC/LDC2_W) with the same value/index as the passed value. The StoreInstruction is then deleted along with the instruction above it (which provided the passed value). Any references to the instruction above the StoreInstruction is redirected to the instruction after the StoreInstruction before being deleted.

Case with a loop in the code:

There are four cases:

1. There is an IINC instruction of the same index in the loop. Do not change LoadInstructions in the loop to direct value instructions (BIPUSH/SIPUSH/ICONST/DCONST/FCONST/LCONST/LDC/LDC2_W) and keep the original StoreInstruction and the direct value instruction above it.

2. No StoreInstruction or IINC instruction of the same index is in the loop. LoadInstructions are handled the same as no loop in the method.

3. There is a StoreInstruction of the same index in the loop, but it is before any LoadInstruction of the same index. LoadInstructions are handled the same as no loop in the method.

4. There is a StoreInstruction of the same index in the loop, but it is after a LoadInstruction of the same index. Do not change LoadInstructions in the loop to direct value instructions (BIPUSH/SIPUSH/ICONST/DCONST/FCONST/LCONST/LDC/LDC2_W) and keep the original StoreInstruction and the direct value instruction above it.

Handle ArithmeticInstructions

When an ArithmeticInstruction is reached, the method getArithmeticRes is called to calculate the result obtained in the arithmetic operation. In details, if the ArithmeticInstruction is a binary operation, the program will look at two instruction directly before the ArithmeticInstruction using the method getArithmeticVals. If the ArithmeticInstruction is a unary operation, the program will only look at the instruction directly above the ArithmeticInstruction. If one of the instruction(s) above is LoadInstructions or other ArithmeticInstructions, null is returned to the getArithmeticRes method which will return null as a result to the main loop. If the instruction(s) above are all instructions that have a direct value (e.g. ICONST, LDC, BIPUSH, etc.), the values will be returned to the method getArithmeticRes which would then handle the calculation depending on the type of ArithmeticInstruction. The getArithmeticRes method returns the result to the main. If the result is null, the program moves on to the next instruction in the InstructionList. Otherwise, the result is stored into the Constant Pool and a LDC/LDC2_W instruction with the index of that value is inserted before the ArithmeticInstruction. The ArithmeticInstruction is deleted along with the instructions above it which provided the direct value needed for the arithmetic operation. Any references to the first instruction that provided the direct values needed for the arithmetic operation is redirected to the LDC/LDC2_W inserted.

Handle ConversionInstructions

ConversionInstructions are optimised in a similar same way as ArithmeticInstructions.

When a ConversionInstruction is reached, the method getConversionRes is called which in turn calls the method getConversionVals. The getConversionVals method looks at the instruction directly above the ConversionInstruction and checks if it is an instruction that provides a direct value (e.g. ICONST, BIPUSH, LDC, etc.). If yes, the value of that instruction is returned to getConversionRes. If not, null is returned. If the result returned is null, getConversionRes will also return null, if not, getConversionRes will perform type casting on the value returned from getConversionVals depending on what type of ConversionInstruction it is and returns the type casted value back to the main loop. If the value returned is null, nothing else is done and the program looks at the next instruction in the InstructionList. If the value returned isn't null, it is stored into the Constant Pool and a LDC/LDC2_W instruction is inserted with the index of the value in the Constant Pool. The ConversionInstruction is deleted along with the instruction above it which provided the direct value needed for conversion. Any references to that instruction which provided the direct values needed for conversion is redirected to the LDC/LDC2_W inserted.

Handle LCMP, DCMPG, DCMPL, FCMPG and FCMPL Instructions

The two values needed for comparison are found in the same way as a binary ArithmeticInstruction using the getArithmeticVals method. If one of the values that getArithmeticVals returned is null, nothing is done and our program will move on to the next instruction in the InstructionList. If both values are not null, our program will compare the values and calculate the result received from the comparison which is one of the values from

-1, 0, 1. The program will then insert an ICONST instruction with the result value. The comparison instruction is deleted along with the two instructions that provided the direct values for comparison. Any references to the first instruction that provided the direct values for comparison is redirected to the ICONST instruction added.