Coursework Part II: Code Optimisation

Carla Hyenne, Minttu Alakuijala, Nathalie Von Huth

Introduction

Our implementation for optimising Java Byte Code with variable folding works as follows. For a given class, we iterate through each method, and for each method, we iterate through every instruction. For each Instruction, we check for it's type and handle it as appropriate. The instruction types we are optimising are:

- Local Variable Instructions
    - Store Instructions
    - Load Instructions
- Arithmetic Instructions
- Conversion Instructions
- Compare Instructions

To do so we implemented methods to handle the operand stack, and more specifically to retrieve number values from instructions that produce variables.

Once an `Instruction` is optimised, the method is updated with a new `Instruction List`. Then we reiterate through the instructions from the start, until every instruction is optimised and the method returns.

The idea is that whenever possible, instructions are replaced with a `PUSH Instruction`. A `PUSH Instruction` is a BCEL class, which determines what instruction from `BIPUSH, SIPUSH, LDC, LDC2_W, xCONST_n` is most appropriate. This way, when an instruction is processed and removed, its resulting value is pushed to the top of the stack and can be accessed by future instructions.

Explained in this report is how we are handling variables on the stack, and how each instruction type is optimised for variable folding.

Stack Handling Methods

Popping values from the stack (1)

Targetting Java Byte Code instructions: `BIPUSH, SIPUSH, ICONST, DCONST, LCONST, FCONST, LDC, LDC2_W.`

The method `popStack(InstructionList, InstructionHandle, ConstantPoolGen)` is used within optimising methods to access the value(s) on top of of the operand stack. This is done by iterating backwards through the instruction list until an instruction which has pushed a value onto the stack is found.

When such an instruction is found and it's value returned, the instruction is deleted.

Pushing values to the stack (2)

When possible in our optimisation, we will insert and delete instructions. When the result of an operation (for example, the `Number` result of `IADD`) needs to be pushed to the stack, we take the following steps:

- Add a new `Constant` to the `ConstantPool`
- Insert a `PUSH` Instruction.
- Remove the initial instruction

Thus, the initial instruction has been replaced with a `PUSH` instruction. This is so the result can be accessible as a value on the stack for the next instructions.

Local Variable Instructions

Store Instructions

Java Byte Code instructions: `DSTORE, FSTORE, ISTORE, LSTORE`

When a Store Instruction is found, we retrieve its index. We iterate through the remaining instructions after the storing, and as long as there is no Store Instruction with the same index to overwrite the value in the register, every Load Instruction with the index is replaced by a `PUSH` instruction.

If a Store Instruction with the same index is encountered, the iteration breaks (this is the case for Dynamic Folding when variables change values). This Store Instruction will be evaluated later on. If there are no more instructions to evaluate, the iteration breaks.
Then, the original Store Instruction can safely be removed since all corresponding Load Instructions have been optimised.

Load Instructions

Java Byte Code instruction: `ILOAD, LLOAD, DLOAD, FLOAD`

As explained above in Store Instructions, a Load Instruction is optimised using the technique of pushing a value onto the stack (2) in the method:
`optimiseLoadingOp(InstructionList, InstructionHandle, MethodGen, ConstantPoolGen, Number);`

Arithmetic Instructions

Java Byte Code instructions: `IADD, ISUB, IMUL, IDIV, DADD, DSUB, DMUL, DDIV, LADD, LSUB, LMUL, LDIV, FADD, FSUB, FMUL, FDIV, INEG, DNEG, FNEG, LNEG, IREM, DREM, FREM, LREM`

When an Arithmetic Instruction is found, we call:
`optimiseArithmeticOp(InstructionList, InstructionHandle, MethodGen, ConstantPoolGen);`
Within this method, to perform the operation we call:
`evaluateArithmeticOp(InstructionList, InstructionHandle, ConstantPoolGen);`
It performs the operation by finding the last value(s) on top of the stack. The result is pushed to the stack (2) so it can be accessed by the next instructions.

Conversion Instructions

Java Byte Code instructions: `I2D, L2D, F2D, D2I, L2I, F2I, I2L, D2L, F2L, I2F, L2F, D2F`

When a Conversion Instruction is found, we call:
`optimiseNumberConversion(InstructionList, InstructionHandle, MethodGen, ConstantPoolGen);`
This method retrieves the value on top of the stack and pops it from the stack (1). The value is appropriately converted, and the instruction replaced with a `PUSH` Instruction so the new value is added to the stack (2).

Compare Instructions

Java Byte Code instructions: `DCMPG, DCMPL, FCMPG, FCMPL, LCMP`

When a Compare Instruction is found, we call:
`optimiseCompareOp(InstructionList, InstructionHandle, MethodGen, ConstantPoolGen);`
The last two values are popped off the stack (1) and compared. The compare operation returns the appropriate result, and it is replaced with a `PUSH` Instruction.