Optimisation Report

Team 18
Junmin Park, SN **20011190**
Nafiz Zaman, SN **20014446**
Yousef Essam Yousri Zaher, SN **20018291**

Data Structures:

These are the data structures that we used throughout our optimisation programme. These are used to store data that will be useful for some of the operations. *variable* HashMap is only used for the regular optimization whereas *variableInstruction*s and *variableLoad* HashMaps are only used for the PeepHoleOptimization method.

HashMaps:

- *variables* (**Integer**, **Number**): to store the value of a given variable.
- *variableInstructions* (**Integer**, **InstructionHandle[]**): to store the Load and Store Instructions for a variable.
- *variableUsed* (**Integer**, **Boolean**): to indicate whether the variable has or hasn't been used through the program.

Stacks:

- *valuesStack* (**Number**): holds the numerical values caused by instructions
- *loadInstructions* (**InstructionHandle**): holds the instructions that load values onto the stack

ArrayLists:

- *loopBounds* (**InstructionHandle**): holds the first instruction of each loop in indices 2n, and final instruction of each loop in indices 2n + 1, where $n \in \mathbb{N}$**,** $n \leqslant$ number of loops.

Other:

- *deleteElseBranch* (**Boolean)**: indicates whether to delete the else branch when in a GOTO
- *blockOperationIfInLoop* (**Boolean**): indicates whether to skip an operation.

Running Different Optimisations:

Optimize method calls two optimization methods:

- *runRegularOptimization*, which runs simple folding, constant variable folding, and dynamic variable folding.
- *runPeepHoleOptimization*, which runs the peephole optimisation that deletes dead code, and keeps running it over and over until there is no more dead code detected.

Regular Optimisation:

For each method, we run the regular optimization with *runRegularOptimizaiton*, which creates an *instructionList* containing the list of instructions from the method code. Then for each instruction, it calls *handleInstruction*.

handleInstruction calls different handling methods depending on the class/interface the instruction is a subclass of/implements. The different handling methods are listed below:

Handle Arithmetic Operation

The *handleArithmetic* method pops the two values which are involved in the operation from the stack and the load instructions of those values are also popped from the *loadInstruction* stack, in order to delete them from the *InstructionList*.

When the desired operation is performed, the result is pushed onto the stack, and a new load instruction for that value is created, which is pushed onto the *loadInstruction* stack, and replaces the operation instruction in the InstructionList. By doing this, the mathematical operations are done by the compiler and only the instruction that loads the result remains.

Handle Long Comparison

handleLongComparison is the method that handles comparisons for the *LCMP* instruction. It pops two values from *valuesStack*, converts them to longs and compares the values. The two load instructions are popped from *loadInstructions* and deleted from the *InstructionList*.

The result of the comparison, 0 (if equal), 1 (if the second value is larger) or -1 (if the second value is smaller), are pushed to *valuesStack*.

A new load instruction that contains the result replaces the *LCMP* instruction and is also pushed onto the loadInstruction stack.

Handle Comparison

handleComparison is used when the instruction is an *IfInstruction*. If the instruction is comparing the value with zero, then only one value is popped from the valueStack, but when comparing two values, two values are popped from the *valueStack*.

If the result of the comparison is *true*, then the *deleteElseBranch* variable is set to *true*. If *false*, then the instructions from and including the comparison instruction up to and including the last instruction before the target of the comparison instruction, are removed from the instructionList.

This method optimises branch instructions by planning the deletion of either the if or else branch. If *deleteElseBranch* variable is *true* then, when reaching the goto instruction, *handleGoTo* deals with deletion of the else branch.

Handle Conversion
For conversion instructions, we pop the value from the top of the stack and convert it to another type depending on the instruction. Then push the result back onto the top of the stack. We also pop the loadInstruction, and delete it from the InstructionList, as well as replace the conversion instruction with a load instruction that loads the new result.

Handle GoTo
In GoToInstructions, if *deleteElseBranch* is *true*, instructions from and including the *GoToInstruction*, up to but excluding the target instruction are deleted from the *InstructionList*. The method also sets *deleteElseBranch* to *false* in order to prevent unintended deletions of else branches of different if statements.

Handle Store
For store instructions, the value and load instructions are popped from the *valueStack* and *loadInstructions*, and using the index extracted from the instruction as the key, the value is put in the *variables* HashMap for that key.

Handle Constant Load
Load instructions of constants are dealt by pushing the extracted value from the instruction onto the *valueStack*, and pushing the instruction to the loadInstruction stack.

Handle Variable Load
Load the value from the *variables* HashMap, and then push that value and the load instruction onto the *valueStack* and *loadInstructions* stack respectively.

Locating Loop Bounds
We locate a loop if there exists a GoToInstruction that targets another instruction that has a position less than the GoToInstruction (i.e. referring to an instruction before it, hence creating a loop). We then place the target of the GoToInstruction into the *loopBounds*, and then place the GoToInstruction into the list, such that the target appears first in the list.

Handling Dynamic Variables
When handling variable load instructions, if *blockOperationIfInLoop* is *false* then the code checks if the value of the variable changes within the loop, and if the value does change, then *blockOperationIfInLoop* is set to *true*, we do this in order to stop the simplification of values, as the value in the variable could change. *blockOperationIfInLoop* is set to *false*, after the next instruction is handled, if that instruction is not a load or conversion instruction.

Optimisation of the following instructions is not performed if *blockOperationIfInLoop* is *true*:

- Arithmetic Operations,
- Comparisons,
- Conversions.

Peephole Optimisation

peepHoleOptimization is run under a while loop and is repeated until the code does not change. The idea of this method is to first get the code and generate the *InstructionList*. It will then loop through the *InstructionList* and get a handle of each instruction, then check it using *checkInstruction*, which similarly to *handleInstruction* decides which method to choose according to which Instruction it implements or is a subclass of After all the instructions have been checked we then return a boolean that is true if there is no dead code to remove.

checkVariableLoad

This method's main purpose is to set the key in the *variableUsed* HashMap to true, in order to keep it so that *removeDeadCode* does not remove it from the *InstructionList*.

checkLoad

Pushes the constant load instruction onto the loadInstruction stack, so that if a StoreInstruction is found, we can locate the instruction used to load the value.

checkStore

Places the variable into the *variableUsed* HashMap, and sets its value to *false*, as it has not been used yet. It then also places the InstructionHandles of the StoreInstruction and the most recent LoadInstruction in an array., which is then placed inside the HashMap for the variable. This is used in the *removeDeadCode* to delete those instructions if the variable has not been used.

removeDeadCode

This method will go through the instructionList and then remove variables that are not in use. This cleans up the code and improves the efficiency of it. It will then only return *true* back to the *peepHoleOptimization* method, if there is no more dead code to be removed.

Contributions
Junmin : Brainstorming, report, finalising report
Nafiz : Brainstorming, report, finalising report
Yousef :Leader, Brainstorming, programming, finalising report