

Java Bytecode Optimization

Variable folding optimization for Java bytecode

COMP2010 – Compilers

Group members: James Xue, Kiran Batthi, Volodymyr Siedlecki, Patrick Steffens
Lecturer: Dr Shin Yoo

Contents

1	Introduction	3
2	Classification of instruction types	3
3	Implementation description	4
3.1	optimize()	4
3.2	performFolding(ClassGen gen, ConstantPoolGen cpgen, Method method)	4
3.2.1	Data Structures	4
3.2.2	Other Variables and Data Structures	5
3.2.3	Code Description	5
3.3	performReduction(Deque<InstructionHandle> instructionStack, InstructionList instList, Deque<Integer> pushInstrIndexStack, int instrPointer)	8
3.4	cleanUpInstructionList(Map<Integer,ArrayList<InstructionHandle> > map, InstructionList instList, InstructionHandle newHandle)	9
4	Dynamic folding example	10

1 Introduction

The `optimize` method is called and within it a new `ClassGen` object is created. Then the `ConstantPool` and the unoptimized methods are retrieved from the `ClassGen` object. Then, `optimize` iterates over these methods and calls `performFolding` for each one. Once there are no more methods remaining, the optimized code is generated. The general process of this optimisation is depicted in figure 1.

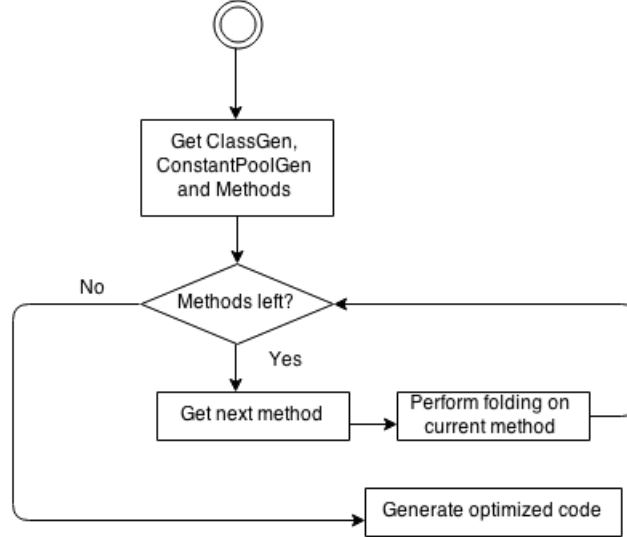


Figure 1: The general process of optimisation

In the following section 2, we will provide further explanation on how we classified the different bytecode instruction types. It is followed by a detailed description of our most important functions (section 3). Since every in section 2 described instruction type is handled differently, every particular explanation is supported by an appropriate flow chart diagram. The report closes with an example showing the difference between unoptimized and optimized code (section 4).

2 Classification of instruction types

In order to distinguish between the different bytecode instructions, we classified the most important ones. Since the scope of this coursework only included arithmetic and logic operations, only the data types `int`, `long`, `float` and `double` are considered. We decided to have five different classes of instructions: Push, Conversion, Store, Arithmetic Operation and Comparison.

- **Push:** Instructions that push constants onto the local constant stack (`sipush`, `bipush`, `i/f/l/dload`, `ldc`, `ldc_w`, `ldc2_w`, `i/f/l/dconst`).
- **Conversion:** Instructions that convert the top most constant of the constant stack into a different type (`x2y` where $x, y \in \{i, l, d, f\}$ and $x \neq y$).

- **Store:** Instructions that pop a constant from the stack and store it in the local variable table (`i/f/l/dstore_x` where $x \in [0, 3]$)
- **Arithmetic Op.:** Operation that performs an arithmetic or logical calculation (`i/l/f/dadd/sub/mul/div/rem/neg, i/lshl, i/lshr, iushr, i/lor, i/land, i/lxor`).
- **Comparison:** Instructions that perform a comparison and either push the result on the stack or jump to a certain destination (`d/fcmpg, d/fcmpl, lcmp, ifne, ifle, iflt, ifge, ifgt, ifeq, if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt, ic_imple`).

3 Implementation description

3.1 optimize()

Brief: Calls `performFolding` for every method.

It iterates over all methods and calls `performFolding`, passing the current method, the `ConstantPoolGen` and the `ClassGen` object. After the optimization is done for all methods, the optimized byte code is generated.

3.2 performFolding(ClassGen gen, ConstantPoolGen cpgen, Method method)

Brief: Performs simple, constant and dynamic folding on method `m`, invoking itself recursively until no further optimization is possible.

3.2.1 Data Structures

It uses five major data structures to always keep track of the algorithm's and method code's state:

- **constantStack:** Simulates the constant stack and contains only the most recent values.
- **instructionStack:** Contains the actual processed instructions and all those that will be removed at the end.
- **pushInstrIndexStack:** Contains indices of all push instructions of the current optimisation step in the order they appeared in the bytecode.
- **instructionMap:** Instead of completely removing the store operation, it is saved in this data structure in case it is needed at a later point in the code.

- **constantMap**: When a store instruction is read, the topmost constant is popped from the **constantStack** and stored in a local variable table. This **constantMap** simulates that table, so that we always have access to the most recent value of a variable.

3.2.2 Other Variables and Data Structures

- **methodCode**: Code of the method, containing a header and the **instList**.
- **instList**: List consisting of references to all instructions (**InstructionHandles**) in the method's code
- **remove**: Flag that indicates if an instruction can be removed, i.e. an interaction with the **instructionStack** is necessary.
- **changed**: Flag that indicates whether the instructions have been optimized (i.e. the original code has changed)
- **instrPointer**: Indicates the number of instructions on the **instructionStack**. Incremented only when an instruction is added and decremented when instructions are popped from the stack.

3.2.3 Code Description

The **performFolding** method does all three types of folding: simple, constant, and dynamic. It first retrieves the method's code from the method object and then receives all instructions as a list (**instList**). Then it iterates over the instruction list using the instruction **handle** (pointer to specific instruction in **instList**). Each handle is checked to see if it is a valid instruction - if it is not, then the instruction is ignored and the next handle is addressed. The algorithm determines an instruction's type by making use of Java's **instanceof** operator.

When the loop is finished or has been interrupted by an optimization, the **changed** flag is checked. If **changed** is **true**, the actual reduction step is performed (**performReduction**, subsection 3.3), followed by a clean up (**cleanUpInstructionList**, subsection 3.4) of the **instList**. The latter is necessary to get rid of store-related instructions that do not have an appropriate load and are therefore useless. After the clean up, a new method is created which replaces the current one, and **performFolding** is invoked with this new method (i.e. the optimized code of the original method) as actual parameter.

As explained before, it checks whether each instruction is of type Push (direct/indirect), Conversion, Store, Arithmetic Operation or Comparison (see section 2 for further explanation).

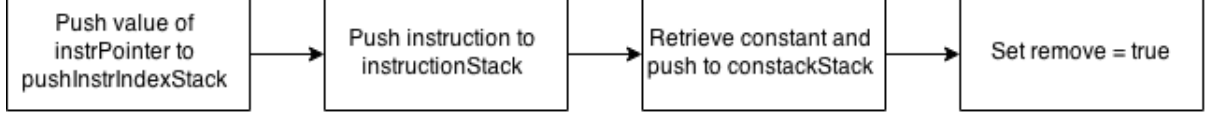


Figure 2: Control flow for push instructions

Push instructions. First, the instruction is pushed to the `instructionStack`, and the loaded constant is pushed to the `constantStack`. The flag `remove` is also set to `true` which indicates that all following instructions must be taken into account. In the next iteration, if `remove` is set to `true`, it checks the next instruction. If the instruction is not of type Push, it executes specific code according to the instruction type.

It is worth mentioning, that we distinguish between *direct* and *indirect* push operations. Direct push instructions immediately push constants to the stack. Indirect push instructions first load the constant from the local variable table and then push it to the constant stack. The only indirect push instructions are those of type `load`.

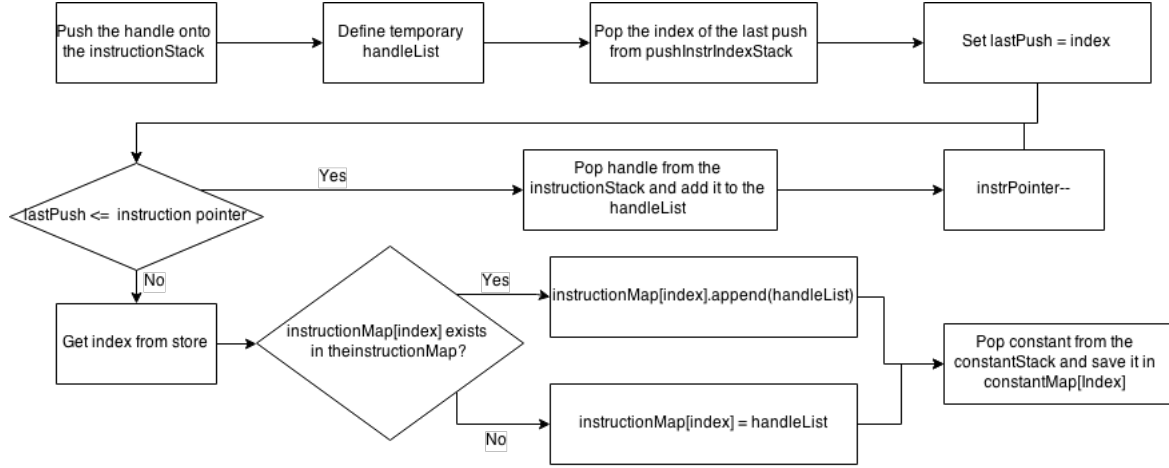


Figure 3: Control flow for store instructions

Store instruction. Firstly, the instruction's `handle` is added to the `instructionStack` and a temporary handle list (`instructionHandles`) is defined. Then, a variable called `lastPush` is created and initialized with the index of the last push operation, i.e. the topmost element of the `pushInstrIndexStack`. Then a while loop iterates over the `instructionStack` and in each iterations pops the topmost instruction and adds it to `instructionHandles`. Finally, `instrPointer` is decreased. This repeats until all handles that have been added since the last push operation are popped from the `instructionStack`. In this way, the algorithm can even consider conversion instructions which are placed between the last push and the current store instruction. Hence, it pops all handles that are necessary for this store instruction ¹. Once the loop has

¹This might not even be necessary. However, due to lack of time, we kept it in our implementation, because it works fine.

ended, it stores the value in the `constantMap`, where the key is the `store` instruction's reference index. It also saves the `handleList` in the `instructionMap`, again using the reference index as key. In case of dynamic folding, the `handleList` is added to the existing one in `instructionMap`. In case anything goes wrong, the containers will be cleared and `remove` is set to false, meaning that the pattern matching process will start from new in the next iteration.

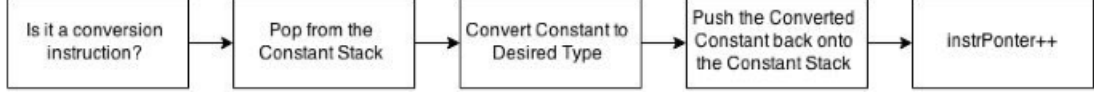


Figure 4: Control flow for conversion instructions

Conversion instructions. The topmost constant on the `constantStack` is popped and converted to the desired type. It is then pushed back to the `constantStack` and the particular instruction handle is pushed to the `instructionStack`. Finally, the `instrPointer` is incremented.

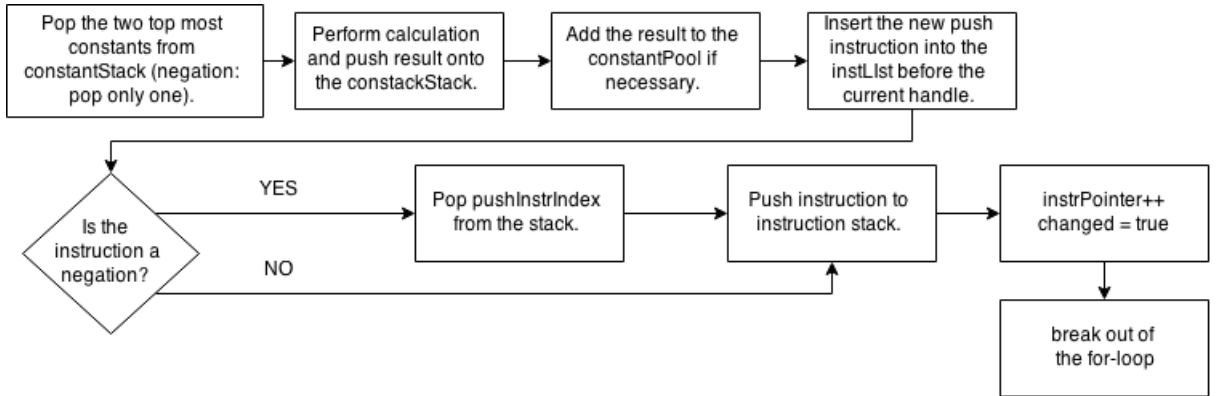


Figure 5: Control flow for arithmetic operations

Arithmetic instructions. If it is an arithmetic instruction, the two topmost constants are popped from the `constantStack`. In case of a negation, only the one topmost constant is popped. The desired calculation is then performed and the result is pushed to the `constantStack` and, if necessary, added to the general constant pool. Next, a new push instruction is inserted within the instruction list directly before the current handle. The type of this instruction depends on the size of the variable. In that way, the algorithm does not add constants to the constant pool when they can be also pushed directly (using `sipush`, `bipush`, `i/l/d/fconst`).

If the instruction is not a negation, the topmost push instruction index is popped from `pushInstrIndexStack`. This is because two push operations are involved in the arithmetic operation and therefore the removal should not stop when reaching the last, but when reaching the second last push instruction on the `pushInstrIndexStack`. After

that, the instruction is pushed to the `instructionStack` and the `instrPointer` is incremented by one. Also, the `changed` flag is set to `true`. Finally, the function breaks out of the loop, since the algorithm only performs one optimization at a time.

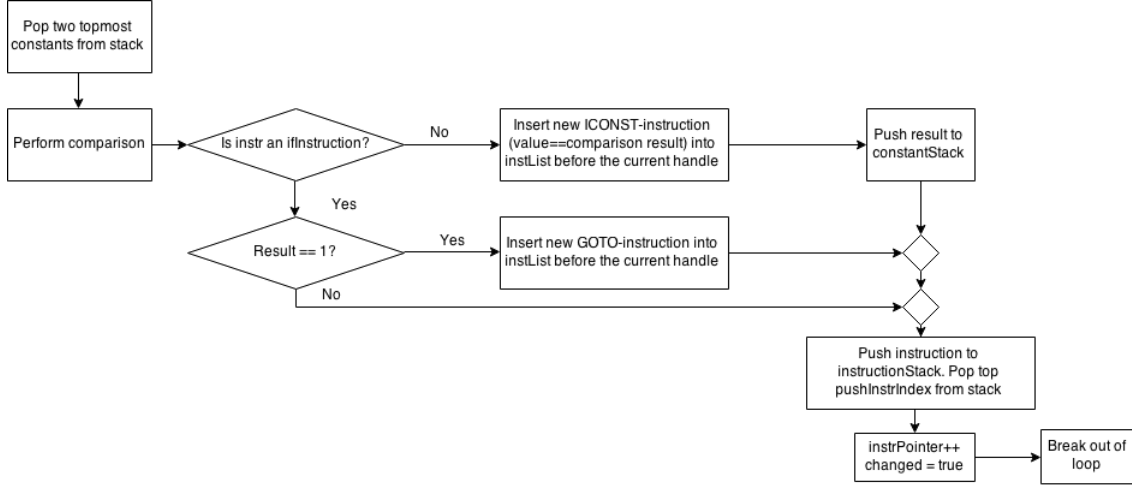


Figure 6: Control flow for comparison operations

Comparison instructions. If the instruction’s type is a comparison, the topmost constants are popped from the stack before the comparison is performed. Since particularly if-instructions imply a jump if the comparison evaluates to `true`, in that particular case the algorithm checks if the result is equal to 1, i.e. `true`. If this is the case, a new `goto` instruction is inserted into the `instList` before the current handle and the instruction is pushed to the `instructionStack`. This means nothing more than: the current instruction is replaced by a `goto` instruction. If the result is equal to 0, no new instruction is added and the current one will simply be removed.

If the instruction is not an if-instruction, the current `handle` is replaced by an `iconst` instruction. Afterwards, the `instrPointer` is incremented and the `changed` flag is set to `true`. Finally, the function breaks out of the loop, since the algorithm only performs one optimization at a time.

```

3.3 performReduction(
    Deque<InstructionHandle> instructionStack,
    InstructionList instList,
    Deque<Integer> pushInstrIndexStack,
    int instrPointer)

```

Brief: Removes all instructions that are not needed any more once the optimization was successful.

This method deletes all instructions in the `instList` that are on the `instructionStack` between the last push and the top. If one of the deleted handles is still targeted by branch instructions, their targets are set to the parameter `newHandle` and the `TargetLostException` therefore handled accordingly.

```
3.4 cleanUpInstructionList(  
    Map<Integer,ArrayList<InstructionHandle> > map,  
    InstructionList instList,  
    InstructionHandle newHandle)
```

Brief: Removes all instructions that are not needed any more once the optimization was successful.

This deletes all unneeded instructions from the instruction list. This is necessary to get rid of store related instructions that do not have an appropriate load and are therefore not needed any more.

First, a list is defined which will store all the entries that will be removed (`removeEntries`). Then the algorithm iterates over all entries in the `instructionMap` and checks whether the current `instList` contains a load with the same reference as the current `entry`'s key. If not, the `entry` is stored in `removeEntries` for later removal.

If one of the deleted handles is still targeted by branch instructions, their targets are set to the parameter `newHandle` and the `TargetLostException` therefore handled accordingly.

4 Dynamic folding example

The following example illustrates how well our optimization performs. It implies dynamic and constant folding.

Given the following Java code:

```
int a = 42;
int b = (a + 764) * 3;
a = 22;
return b + 1234 - a;
```

Our optimization algorithm is able to perform all calculations at compile time and replaces almost all of the instructions of the unoptimized code with a single push instruction:

0: bipush	42	0: sipush	3630
2: istore_1		21: ireturn	
3: iload_1			
4: sipush	764		
7: iadd			
8: iconst_3			
9: imul			
10: istore_2			
11: bipush	22		
13: istore_1			
14: iload_2			
15: sipush	1234		
18: iadd			
19: iload_1			
20: isub			
21: ireturn			
(a) Unoptimized		(b) Optimized	

Figure 7: Unoptimized and optimized Java bytecode