SUPERBLAME

Variable folding optimization for Java code

COMP2010 – Compilers

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

Contents

1 Introduction

This algorithm uses five principle data structures and a recursive method to reduce the given bytecode as much as possible. In brief overview, a reduction is found, the instructions are changed, and the the process repeats until no more changes and can occur and hence no further optimization is possible. It's important to note that all three simple, constant, and dynamic folding occur within the same method performFolding. This report is organized by the descriptions of the data structures and then explanations of the crucial methods. This algorithm uses five principle data structures and a recursive method to reduce the given bytecode as much as possible. In brief overview, a reduction is found, the instructions are changed, and the the process repeats until no more changes and can occur and hence no further optimization is possible. It's important to note that all three simple, constant, and dynamic folding occur within the same method performFolding. This report is organized by the descriptions of the data structures and then explanations of the crucial methods.

2 Classification of instruction types

3 Implementation description

3.1 Data structures

We used 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 need 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.

3.1.1 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 Function description

3.2.1 `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.

3.2.2 `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.

The `performFolding` method does all three types of optimizations, simple, constant, and dynamic. It gets the code from the method and then receives all of instructions as a list (`instList`). 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`), followed by a clean up (`cleanUpInstructionList`) 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.

Missing figure

Control flow for method `performFolding(...)`

Direct 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 checked. In the next iteration, if `remove` is set to `true`, it checks the next instruction not of the type push after first push, or a series of sequential push operations, and executes specific code according to the instruction type.

Missing figure

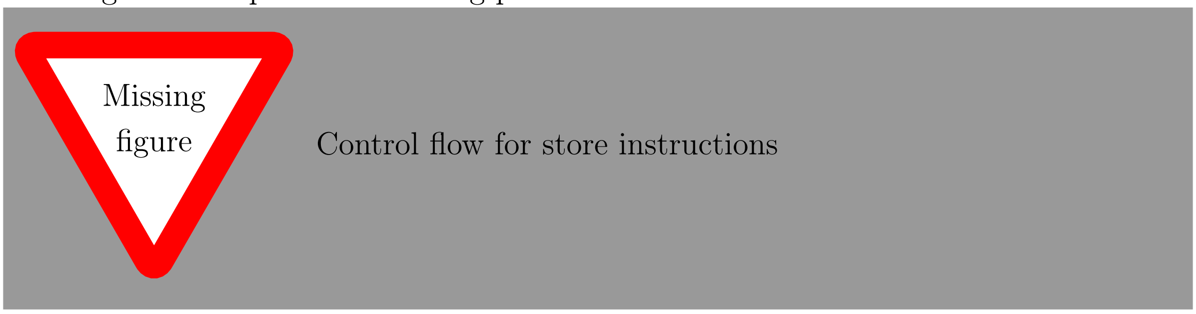
Control flow for direct push instructions

Indirect push instruction (`load`). The `load` instruction is classified as an indirect push operation and therefore goes through the same steps as if it were a direct push operation. They are not handled in the same `if`-block because the way of getting the constant from a load instruction differs from the way of getting it from direct push operations. The reason for this is, that first the constant must be loaded from the constant pool and then be pushed to the stack.

Missing figure

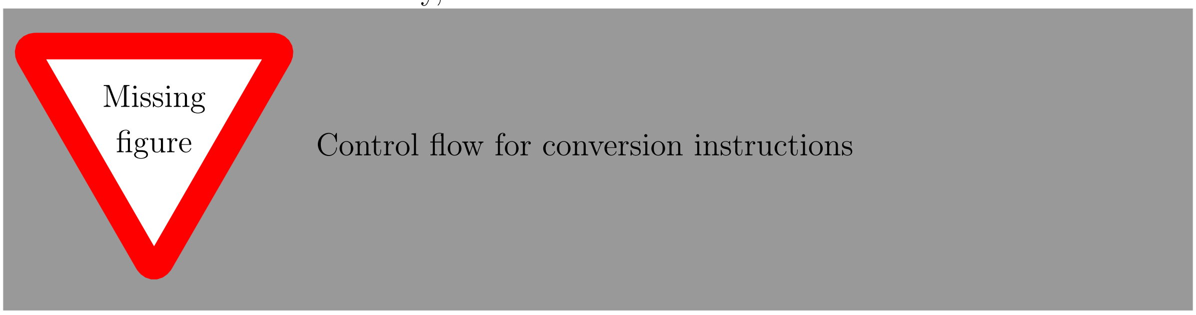
Control flow for indirect push instructions

Store instruction. Firstly, the instruction's `handle` is added to the `instructionStack` and a temporary handle list (`instructionHandles`) is defined. Then, a variable called count is created and initialized with the index of the last push operation. Immediately following is a while loop that iterates over the `instructionStack`, 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 conversions which are placed between the last push and the current store instruction. Hence, it pops all necessary handles for the storing. Once the loop ends, it stores the value in the `constantMap`, where the key is the `store` instruction's reference. It also saves the `handleList` in the `instructionMap`, again using the reference index as key. In case of dynamic folding, the temporary handle is added to the existing array in instruction map. 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.



Control flow for store instructions

Conversion instructions. The value is taken from the instruction and converted to the desired type. It is then pushed to the `constantStack` and the handle is pushed to the `instructionStack`. Finally, the `instrPointer` is incremented.



Control flow for conversion instructions

Arithmetic instructions. If it is an arithmetic instruction then the two topmost constants are popped from the `constantStack`. The desired calculation is performed and the result is pushed to the `constantStack` and also added to the general constant pool. Next, a new `ldc`-instruction is inserted within the instruction list directly before the current handle. If the instruction is not a negation, the topmost push instruction index is popped from `pushInstrIndexStack`. This is because it is one of two push operations involved in the arithmetic operation and therefore the removal should not stop before reaching the first push involved. Then the instruction is pushed to the `instructionStack` and the `instrPointer` is incremented by one and the changed flag is set to true. Finally, the function breaks out of the loop, since the algorithm only performs one calculation at a time.

Control flow for arithmetic instructions

Comparison instructions. If the instruction's type is a comparison, pop the topmost constants from the stack before the comparison is performed. Since particularly if-instructions imply a jump if the comparison evaluates to `true` the algorithm checks in that particular case 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.

Control flow for comparison instructions

3.2.3

```
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 referenced by a branch instruction, this instruction is being updated and the error therefore handled accordingly.

3.2.4 cleanUpInstructionList(
Map<Integer,ArrayList<InstructionHandle> > map,
InstructionList instList)

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 instruction 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.

4 Dynamic folding example

Java code:

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

```
0: bipush        42
2: istore_1
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

```
0: ldc   #48              //int 3630
21: ireturn
```

(b) Optimized

Figure 1: Unoptimized and optimized Java bytecode