COMP2010 Coursework 2 Report

Lecturer: Dr Shin Yoo

By Sam Fallahi, Rohan Kopparapu and David Lipowicz

Contents

Intro

With constant folding being the main task for this coursework, first the bytecode was examined to find all the store instructions. On finding a store instruction, if the value being stored to a register could be evaluated as a number, the following steps were taken:

- All load instructions for that register till the end of the program were replaced with one of the following instructions:
    - `BIPUSH`
    - `SIPUSH`
    - `LDC`
    - `LDC2_W`

By using this idea as the basis for the optimisation algorithm, the remaining bytecode was examined to check for other cases which could be optimised.

Algorithm Structure

The method **optimizeMethod** traverses the InstructionList by going through each InstructionHandle at a time. The following sections explain the algorithm used for the bytecode optimisation.

Increment operators

While traversing the InstructionList, every time an increment operator IINC is found, it is replaced with the following instructions:

- `BIPUSH`
- `ILOAD`
- `IADD`
- `ISTORE`

This ensures that the second step, which handles store instructions does not required a special case to handle increment operators. After the instructions are added to the InstructionList, the InstructionHandle is redirected at the newly added BIPUSH and the IINC is removed.

Store instructions

In the case of StoreInstructions, the last number value on the stack is forwarded to the method **handleStoreInstructions** along with the InstructionList and InstuctionHandle

```
private boolean handleStoreInstructions(InstructionList, InstructionHandle,
Number)
```

If the last stack value (explained further below) cannot be computed by simple folding, or is not a dynamic or static constant, the method returns false. If the value is computable, it traverses the InstructionList from the current StoreInstruction till the end of the code or till another StoreInstruction is found for the same register (i.e. the same variable has changed in value).

During this traversal, every instance of a corresponding load instruction is replaced with an explicit value. Depending on the type of store instruction (`ISTORE`, `FSTORE`, `DSTORE` or `LSTORE`), and the size of the value (8, 16, 32 or 64 bits), one of the following instructions are added:

- `BIPUSH`
- `SIPUSH`
- `LDC`
- `LDC2_W`

In the case of `LDC` and `LDC2_W` instructions, the value is put into the constant pool and loaded every time the value is required.

The last stack value (last value pushed to the stack) is obtained by using the method getLastStackPush.

```
private Number getLastStackPush(InstructionList, InstructionHandle)
```

This method traverses the InstructionList from the current instruction, backwards (towards the beginning of the list) till a stack changing instruction is found, using the following method:

```
private Boolean stackChangingOp(InstructionHandle)
```

This method determines whether the instruction is one that adds to the stack. If the instruction is one that adds to the stack (i.e. `BIPUSH`, `SIPUSH`, `ICONST`, `DCONST`, `FCONST`, `LCONST`, `LDC` or `LDC2_W`) the value being added is obtained, the instruction deleted and the value returned.

```
private void safeInstructionDelete(InstructionList, InstructionHandle)
```

This method is used to delete instructions by redirecting the branches to the instruction before it and removing the given instruction handle.

Type conversions

For all conversion instructions found while calculating the last stack value (except for `I2C`), the numbers are converted by using the following method:

```
private Number convertNumber(InstructionHandle, Number)
```

Arithmetic instructions

If the last stack changing operation is found to be an ArithmeticInstruction, the algorithm then recursively searches for previous stack pushes. If the values found can be computed, it deletes the ArithmeticInstruction and returns the result after applying the operation to the values.

(Note: 1 value in the case of unary operators and 2 values in the case of binary operators).

In addition, the following instructions are also handled by the algorithm in the same way that binary ArithmeticInstructions are handled:

- `FCMPG`
- `FCMPL`
- `DCMPG`
- `DCMPL`

The returned value is then added to the constant pool gen and either an `LDC` or `LDC2_W` (for doubles and longs) are added to the InstructionList.

NOP

If `NOP` instructions were found in the bytecode, they were simply removed as they do nothing.

Notes

- `InstructionList.setPositions`() was called each time the bytecode was altered to ensure the integrity(?) of the InstructionList
- `BIPUSH` and `SIPUSH` were used when `LDC` was not required (i.e. when the Number is of size 8 bits or 16 bits, respectively).
- All the direct known subclasses of ArithmeticInstruction have been accounted for
- All the direct known subclasses of ConversionInstruction (except `I2C`) have been accounted for
- All the direct known subclasses of LocalVariableInstruction have been accounted for
- `DCMPG, DCMPL, DCONST, FCMPG, FCMPL, FCONST, ICONST and LCONST` have been accounted for
- `NOP` instructions were handled as an extra!