Compilers Coursework Part 2

Code Optimisations

COMP207P – Compliers

Team 38

Kazuma Hochin	zcabkho@ucl.ac.uk
Sam Pham	zcabsph@ucl.ac.uk
William Lam	zcabwhy@ucl.ac.uk

Table of Contents

1 – Code Optimisations

1.1 – Introduction

In the following report, we will discuss about the algorithm and implementation we used in the COMP207P Compliers coursework. The aim of the algorithm is to optimise code using peephole methods of optimisation. In this project, we used Java and Apache's BCEL (Byte Code Engineering Library).

Why optimise code? Because, the instruction stack that is directly generated from code can sometimes be inefficient. Wasting CPUs cycles, memory and program space. And the larger the program, the more significant it becomes. Optimisations means, to change the instruction stack in such a way that it is smaller and takes less time to process, whether that means to delete instructions or replace them, it is up to the algorithm.

1.2 – Peephole Optimisations

The purpose of peephole optimisations is to perform optimisations over a small segment of the instruction stack by replacing them with smaller and faster sections of instructions. There are a number of different techniques used in peephole optimisations but the one we aim to implement is constant folding where by certain calculations are instead processed and stored when compiling instead of being calculated at run-time.

In the coursework, we aim to process three different situations in constant folding:

- Simple Folding – is constant folding for values such as integers, floats, longs and doubles, resulting in one value that is stores and so doesn't have to be run at runtime. Such as a = 1 + 2, instead of running it at runtime, you can store it as 3 and have 3 at runtime instead
- Constant Variables – is where you replace variables with the value it currently holds so that at runtime, you don't need to access the stack to get the variable value. These variables are assigned once and not reassigned in the scope of the variable
- Dynamic Variables – is similar to the constant variables except that they can be reassigned in the same scope

2 – Optimisation Algorithm

To achieve code optimisation, an algorithm was made and implemented. The optimisation algorithm is split into three different phases:

1. Creating the ForLoop table and the associated HashMap
2. Constant Folding, adding to the constant pool, replacing the instruction and adding to the delete table
3. Condensing the instruction stack by using the delete table and remove the unnecessary instructions

2.1 – ForLoop Table and HashMap

This is the first phase and the aim of this is to create a HashMap which details where in the instruction stack you should you be wary off with constant folding. There are two reasons for this. The first is you don't want to fold the variable that it used in the for loop to iterate as this would potentially mean that the loop goes on forever or never starts. Secondly, the for loop has an if condition and if conditions can be folded to either true of false, if done on the condition in the loop, it would mean the loop would either go on forever, regardless of the change in the iterator variable.

So to avoid these problems, we have made a table, where each entry is a single for-loop, detailing where the loop starts and ends using the index on the instruction stack and what is the reference to the iterator variable in the local variables table. The algorithm will only go through the instruction stack once and create entries in the table. More specifically, the end is where the GoTo instruction is in the stack and the start is the instruction of the for-loop starts, specifically the StoreInstruction before the LoadInstruction which is the target of the GoToInstruction. The table is then used to create a HashMap where the keys are the start and end values and the value is an ArrayList of references to be careful of.

First Phase Pseudocode

```
firstMethod(CGen, CGen, Method)
{
      ForAll(Instruction in the InstructionStack)
      {
            If (Instruction is a GOTO Instruction)
            {
                  Int end = Instruction.getPosition
                  GoToInstruction goTo = Instruction
                  Int target = goTo.getTarget.getPosition
                  Int index = getLocalVariableIndex(goTo)
                  If (target < end)
                  {
                        Int start = goTo.getTarget.getPrev.GetPosition
                        Add to ForLoopTable(start, end, index)
                  }

            }
      }
}

tableToHash()
{
      ForAll(Entries in the ForLoopTable)
      {
            HashMap.Add(Entry.Start , [-1])
            HashMap.Add(Entry.End , [-1])
      }
      ForAll(Entries in the ForLoopTable)
      {
            For(int i = Entry.start while end < Entry.end, temp1++)
            {
                  if (HashMap.keyExists(i))
                        if (HashMap.Value(i) == [-1])
                              HashMap.Value(i) = [Entry.index]
                        Else
                              Add Entry.index to HashMap.Value(i)
            }

      }

}
```

2.2 - Folding, Constant Pool and Replacing Instructions

The aim of the second phase is to partially fold the instruction stack. So, while traversing through the instruction stack, we calculate the values for folding, then add the calculated values into the constant pool and then replace the appropriate instruction with an LDC or LDC2_W to read the value from. But we don't delete the other instructions that are no longer needed. Instead we store the start and end of the segment we wish to delete in a deleteTable and pass it onto the final phase which is discussed later on, hence why this phase is partially folding.

The main two reasons to leave the deletion until after are, one, deleting instructions would shift the index values of the instructions after it, so the HashMap values would also have to change to accommodate the shift. Secondly, removing targeted instructions will cause a TargetLostException which is further explained in the final phase and since this phase only traverses once and we didn't want to backtrack, the only option left was to create another phase.

Second Phase Pseudocode

```
SecondMethod(CGen, CGen, Method)
{
	ForAll(Instruction in the InstructionStack)
	{
		If (Instruction is Arithmetic|| If||InvokeVirtual||GetStatic||Goto||Return)
		{
			DeleteTable delete = new DeleteTable()
			Number number = arithmeticMethod(arg1,arg2)
			CGen.addConstantPool(number)
			if(number is long or double)
			{
				instruction.replaceInstructionMethodWith(LDC2_W(number))
			}
			else
			{
				instruction.replaceInstructionMethodWith(LDC(number))
			}
			delete.add(startDeleteIndex,endDeleteIndex)
		}
	}
}

class deleteTable
{
	ArrayList a = new ArrayList()
	addEntry(int start, int end)
	{
		a.add(start,end)
	}
	getEntry(int index)
	{
		return a.get(index)
	}

}
```

2.3 - Condensing the Instruction Set

This final phase is to remove the instructions that have been detailed in the delete table and it goes through the instruction stack twice.

The first traversal is to remove the Branch instructions such as the GoTo and If statements. The reason for this is if you were to first remove an instruction which is a target of a Branch instruction, a TargetLostException will be thrown so to avoid this, the Branch instructions are removed first and the delete table is updated accordingly.

```
16: ifle 23 (Branch Instruction – Target Handle 23)

[java] REMOVED:   23: iconst_0[3](1)
[java] org.apache.bcel.generic.TargetLostException: {   23: iconst_0[3](1) }
```

Figure 1. An example of the TargetLostException

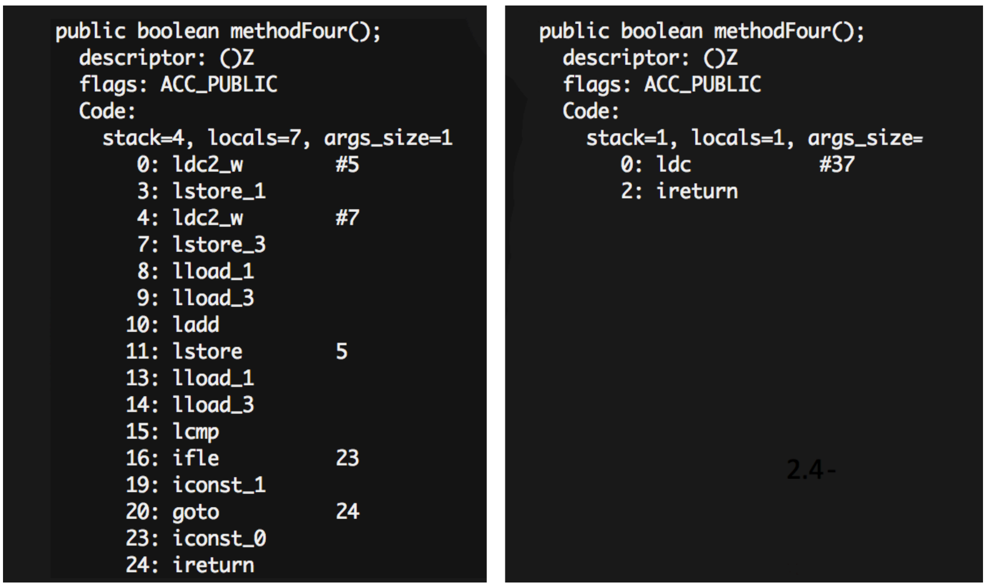
The second traversal is to remove the rest but this traversal starts at the end of the stack and goes backwards. This is to avoid having to edit the values of the delete table. By the end of this traversal, the algorithm is finished and the instruction stack is optimised.

Third Phase Pseudocode

```
thirdMethod(InstructionTable, CGen, DeleteTable)
{
        ForAll(Entries in DeleteTable, Forward Traversal)
        {
                ForAll(Instructions between Start and End)
                {
                        If Instruction is a BranchInstruction
                                Remove Instruction
                                Edit DeleteTable
                }
        }
        ForAll(Entries in DeleteTable, Backward Traversal)
        {
                ForAll(Instructions between End and Start)
                {
                                Remove Instruction
                }
        }
}
```

2.4– Results

Below is the instruction stack before and after optimisation for methodFour() in the ConstantVariable class.



```
public boolean methodFour();
  descriptor: ()Z
  flags: ACC_PUBLIC
  Code:
    stack=4, locals=7, args_size=1
       0: ldc2_w        #5
       3: lstore_1
       4: ldc2_w        #7
       7: lstore_3
       8: lload_1
       9: lload_3
      10: ladd
      11: lstore        5
      13: lload_1
      14: lload_3
      15: lcmp
      16: ifle          23
      19: iconst_1
      20: goto          24
      23: iconst_0
      24: ireturn
```

```
public boolean methodFour();
  descriptor: ()Z
  flags: ACC_PUBLIC
  Code:
    stack=1, locals=1, args_size=
       0: ldc           #37
       2: ireturn
```

Figure 2. Before and after optimisation for methodFour() in ConstantVariableFolding

Below is the instruction stack before and after optimisation for methodTwo() in the DynamicVariable class.

```
public boolean methodTwo();
  descriptor: ()Z
  flags: ACC_PUBLIC
  Code:
    stack=3, locals=3, args_size=1
       0: sipush        12345
       3: istore_1
       4: ldc           #2
       6: istore_2
       7: getstatic     #3
      10: iload_1
      11: iload_2
      12: if_icmpge     19
      15: iconst_1
      16: goto          20
      19: iconst_0
      20: invokevirtual #4
      23: iconst_0
      24: istore_2
      25: iload_1
      26: iload_2
      27: if_icmple     34
      30: iconst_1
      31: goto          35
      34: iconst_0
      35: ireturn
```

```
public boolean methodTwo();
  descriptor: ()Z
  flags: ACC_PUBLIC
  Code:
    stack=3, locals=1, args_size=1
       0: sipush        12345
       3: getstatic     #3
       6: ldc           #50
       8: invokevirtual #4
      11: ldc           #50
      13: ireturn
```

Figure 3. Before and after optimisation for methodTwor() in DynamicVariableFolding

3 – Implementations

3.1– Constant Folding

For constant folding, we used two temporary variables of type Number to deal with constant folding. Our method goes through each instruction in the instruction list and obtains a value from the instruction if it's one of the following:

- LDC
- LDC2_W
- ConstantPushInstruction
- StoreInstruction
- LoadInstruction

The two temporary variables will always hold the two most recent values. The arithmetic operations will be applied to these two temporary variables and the resultant value will be pushed onto the stack, when a certain instruction is seen, as LDC for integers and floats or LDC2_W for longs and doubles.

3.2– Local Variables

We implemented our own local variable table using HashMap. Local variable table is used to add a variable whenever it sees a StoreInstruction and returns a variable whenever it sees a LoadInstruction. By using a HashMap it deals with both constant and dynamic folding as it checks whether a variable already exists and replaces the old variable with new variable if it exists.

3.3– If Statements

For any if statements in the bytecode, we first retrieve the two values from the two temporary variables which have the two values needed to be compared. Then, it checks the type of the two Number objects to change them to the correct primitive type. Afterwards, it compares the two variables by checking the instruction to see which operator should be used. A Boolean is then returned stating whether the condition is true or false. The if statement will then be removed from the instruction list provided that it is not part of a loop.

3.4– For-Loop

For the for-loop, we first created a table detailing the start and end of each loop. We then go through the loop in the second phase of the algorithm and tries to fold as much as possible. However, we did not remove any code regarding the looping variable as otherwise the for-loop would be broken. Furthermore, we made sure the if comparison, the GoTo instruction, the increment instruction and the push of the loop ending number are not removed.

4 – GitHub

The project has been developed using a Git repository which can be accessed with the following link:

https://github.com/kazuchan237/CodeOptimisation