COMP0012
Compilers

Coursework Part II: Code Optimisation
Due on 26/04/2019, 14:00 GMT

Aim

Using Java and BCEL (Byte Code Engineering Library), implement peephole optimisation as much as possible. The Java compiler already does *some* of this. For example, the following Java code

Listing 1: Constant Folding Example: Java Code

```
...
int a = 429879283 - 876987;
// no assignment to a in the middle
System.out.println(527309 - 1293 + 5 * a);
...
```

produces the following constant pool in the corresponding `.class` file:

Listing 2: Bytecode Constant Pool

```
...
const #2 = int  429002296; // this would be variable a
...
const #7 = int  526016; // this would be the subexpression inside println
...
```

We can see that `javac` has performed two arithmetic operations to achieve constant folding: $429879283 - 876987 = 429002296$, and $527309 - 1293 = 526016$. However, if there is no assignment to the variable a, it can be folded further by propagating the value of a. Your peephole optimisation should identify such patterns and perform constant folding as much as possible.

Evaluation

This coursework will be number-graded (0 to 10) and contribute to 10% of the overall course outcome. Each sub-goal will result in specific patterns of bytecode after compilation. Your task is to implement a single bytecode optimiser that will identify and optimise all three of them. Marks will be awarded based on the achievement of each sub-goal:

- **Simple Folding (3 marks):** The first sub-goal is to perform constant folding for the values of type `int`, `long`, `float`, and `double`, in the bytecode constant pool. Note that the Java compiler usually performs this for you. You will be provided with an artificial class file, which contains an un-optimised constant pool.
- **Constant Variables (3 marks):** The second sub-goal requires you to optimise uses of local variables of type `int`, `long`, `float`, and `double`, whose value does not change throughout the scope of the method (i.e. after the declaration, the variable is not reassigned). You need to propagate the initial value throughout the method to achieve constant folding. For example, you will be targeting something like the following:

Listing 3: Example of Constant Variable Folding Target

```
public int optimiseMe(){
        int a = 534245;
        int b = a - 1234;
        // the following argument can be folded into a constant
        System.out.println((120298345 - a) * 38.435792873);
        for(int i = 0; i < 10; i++){
                // the subexpression (b - a) can be folded into a constant
                System.out.println((b - a) * i);
        }
        // the return value can be folded into a constant
        return a * b;
}
```

- **Dynamic Variables (3 marks):** This sub-goal requires you to optimise uses of local variables of type `int`, `long`, `float`, and `double`, whose value *will* be reassigned with a different constant number during the scope of the method. You still need to propagate the value of the variable, but for specific intervals: starting from the assignment (or initialisation) until the next assignment.

- **Additional peephole optimisation (1 mark):** The final sub-goal requires you to implement any additional peephole optimisation (e.g., dead code removal) so that your bytecode optimiser is able to fully optimise the code given as target.

Listing 4: Example of Dynamic Variable Folding Target

```
public int optimiseMe(){
        int a = 123456789;
        // the following arguments can be folded into a constant
        System.out.println((120298345 - a) * 38.435792873);
        System.out.println((120298345 / a) + 99.8398761);
        a = 987654321;
        // the return value can also be folded but using a different value
        return a * a;
}
```

Directory Structure

The coursework files provide a skeleton project with `ant` build script and relevant Java libraries. Download `comp0012-coursework2.zip` from Moodle. Uncompressed, it contains configuration and other files as well as the Java source code directory `src` as follows:

```
comp0012-coursework2
├── src ........................ Java source for optimisation and test targets
│   └── comp0012
│       ├── main
│       │   ├── Main.java ........................ Main recursively processes .class files
│       │   └── ConstantFolder.java .............. Optimises individual .class files
│       └── target
│           ├── SimpleFolding.j .. Jasmin code that generates artificial target for sub-goal 1
│           ├── ConstantVariableFolding.java ..... Target for sub-goal 2
│           └── DynamicVariableFolding.java ...... Target for sub-goal 3
├── test ......................................... JUnit test case sources
├── lib .......................................... Library jars
└── build.xml .................................... ant build script
```

The `src` directory contains a `Main.java` that takes two arguments when compiled:

`java comp0012.main.Main -in` [root of input folder] `-out` [root of output folder]

It will recursively visit all subfolders under the input folder, optimise `*.class` files, and write the optimised classfiles in the corresponding directory structure under the output folder. Currently `ConstantFolder.java` does not do any optimisation: it simply writes unoptimised, unmodified classfiles. The `ant` build script contains a task called `optimise`, which will execute `Main` class. The input folder will be set to the compiled output of the source code; the output folder will be set to `optimised` under the base directory.

Three target files, each corresponding to one of the sub-goals, are provided, along with unit tests. Moreover, the `ant` build script contains `test.original` and `test.optimised` tasks. The first will run the unit tests using the original class files; the second, however, will set the classpath to the `optimised`, thereby testing the optimised classes.

Deliverables

Each group should submit the following deliverables by the submission deadline:

- **Implementation**: a Java implementation of the optimisation. Use the given directory structure and build script in the skeleton files (available from Moodle).
- **Report**: include a written report that contains detailed descriptions of your optimisation algorithm. Describe the optimisation you have implemented in as much detail as possible. There is no page limit.

You should implement the constant folding optimisation in the given `ConstantFolder.java`, while preserving the directory structure. The build script is designed so that simply issuing `ant` will compile the source code and unit test, generate `SimpleFolding.class`, execute the optimisation, and unit test both the original and the optimised classes. Your deliverable should support this process out of the box, and should follow the directory structure below:

```
groupXX .................................. Name the directory as with your group name
├── report.pdf ............................ Your report, either in PDF or Word format
├── src ................................... Java source for optimisation and test targets
│   └── comp0012
│       ├── main
│       │   ├── Main.java ................. Main recursively processes .class files
│       │   └── ConstantFolder.java ....... With your implementation added
│       └── target
│           ├── SimpleFolding.j .. Jasmin code that generates artificial target for sub-goal 1
│           ├── ConstantVariableFolding.java ........ Target for sub-goal 2
│           └── DynamicVariableFolding.java ......... Target for sub-goal 3
├── test .................................. JUnit test case sources
├── lib ................................... Library jars
└── build.xml ............................. ant build script
```

Compress the top level directory (which should be named as `lsagroupXX` where XX is your group number) and submit it through Moodle. The report should be included inside the top level directory. Make sure that you follow this directory structure instructions. **Violations will result in reduction of points**.

Guidelines

This coursework is compulsory. It will be graded on 0 to 10 scale and contribute 10% of the overall marks for the module. You should submit it online through Moodle by 26/04/2019, 14:00 GMT.

Submissions received after this deadline will be considered late and the UCL late submission penalties (`https://www.ucl.ac.uk/srs/academic-manual/c4/module-assessment/#3.11`) will be applied (i.e. to reduce the mark awarded).

Make sure your submission is self-contained. It should not depend on any file outside the submitted directory, such as files on your own hard drive or online. We expect `ant` under `lsagroupXX` directory simply to work, straight out of the box. Testing the submission on the department Linux machines is **strongly** recommended: this is a good way to catch any platform-dependent fault in your script and configurations.

Tools

These are the relevant tools. BCEL is part of the coursework requirement (i.e. you are requested to use this to modify the bytecode); you will probably find the others helpful during the development.

- **Bytecode Engineering Library**: In order to make changes to the bytecode, it is better to use a library (a direct, byte-level access is possible but would be very painful). Apache BCEL (Byte Code Engineering Library: `http://commons.apache.org/proper/commons-bcel/`) is a widely used tool. It comes with a detailed tutorial: `http://commons.apache.org/proper/commons-bcel/manual.html`.
- **Bytecode Disassembler**: Java comes with a disassembler, `javap`, which takes a `.class` file and prints out the corresponding byte code representation. Try `javap -c -verbose YOURCLASS` to see the contents of `YOURCLASS.class`. This will allow you to observe bytecode patterns created by various program structures in Java.
- **Jasmin**: Jasmin(`http://jasmin.sourceforge.net`) is a Java assembler - you can write bytecode source codes in a way similar to writing MIPS assembler language, and Jasmin can compile it into Java classes. If you want to create a fine-tuned bytecode instructions to test your optimisation, this is the right tool (combined with bytecode disassembler). Read `SimpleFolding.j` to get started, along with the online documentation.
- **Java Bytecode Reference:** for the full list of bytecode instructions in Java, see `http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings`.