# Lab Guide 4

# Dynamic compilation in Java

**Objectives:**

- Understand dynamic compilation process, benefits and limitations
- Identification of performance constrains of the Java language

**Introduction**

There are many modern programming languages that use dynamic compilation to support multiple platforms. Java is a language that uses dynamic compilation. The process of compiling Java code starts by transforming the code into an intermediate language called bytecode. The bytecode is cross-platform compatible, the machine only needs a Java Virtual Machine to run the code. The Virtual Machine allows the execution of the program in two different ways: interpretation or compilation. The code execution starts with interpretation (of the program bytecodes). Code compilation is performed during the program execution (dynamic compilation or Just-In-Time compilation, JIT) for chunks of code with high computational costs. In many cases, chunks of code are recompiled to apply more complex optimisations. If the optimisations are no longer valid, previously compiled versions will be used. In this lab session, the students will analyse the Java compilation process.

For the lab session, the students will need a Java implementation of the matrix multiplication. This and other files required for this session are available on e-learning or in the cluster at `/share/cpar/PL04-codigo`.

**Exercise 1 - Java Profiling**

a) Compilation process: Use `-XX:+PrintCompilation` in program execution to identify the methods which were compiled to native instructions. How many times was `mmult` compiled? Why?

b) Identify code executed: In the `MULT.s` file, students will find the assembly code generated by the JIT during program execution. The `perfreport.txt` file contains the execution profile of the same run. Identify the assembly code that was executed.

c) Analyse code: The Java program requires more instructions than C program to perform the matrix multiplication. Analyse assembly code generated from Java (kernel of matrix multiplication) and identify optimisations present in the code. These optimisations were also applied in the C version?

d) Measure compiler impact: Open the file `perfreport2.txt` and analyse if there is a negative impact of dynamic compilation on performance.

e) Garbage Collector: The `Java Virtual Machine` uses a `Garbage Collector` to manage memory at runtime. The `Garbage Collector` sometimes needs to interrupt the program execution. Identify if the cost of the `Garbage Collector is` relevant? Compare the file `perfreport2.txt` (run of the `MULT` class) and `perfreport3.txt` (run of the `MULT_2` class). Why?

f) (*) Measure and compare Java with C: Fill the table with performance data collected with `perf stat -M cpi ./a.out`. Calculate the performance gain between the C and Java versions and comment the results.

| N | Version | Time | CPI | #I (inst_retired.any) | L1_DMiss | Miss/#I |
|---|---|---|---|---|---|---|
| 512 | C | | | | | |
| | Java | | | | | |

**Commands to be used:**

Compile Java program:

```
javac MULT.java
```

Run program:

```
java MULT
```

Print Java assembler (needs `hsdis-amd64.so` lib):

```
java -XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly MULT > MULT.s
```

Run Java with perf:

```
perf record -k 1 java -XX:+UnlockDiagnosticVMOptions -XX:+PreserveFramePointer -
XX:+DebugNonSafepoints -agentpath:.../libperf-jvmti.so MULT

perf inject -i perf.data --jit -o perf.data.jitted

perf report -i perf.data.jitted -n –stdio
```