Translating JVM Code to MIPS Code

Last part of successfully translating and running a Java (or and other JVM language) code is so called “HotSpot™ compilation”. The goal is to create small, executable, pieces of code that can be both heavily optimized and eventually reused when running them repeatedly. Running a native code is faster than interpreting instructions and compiling “at runtime” gives the compiler more info necessary for optimization. Another reason to compile JVM code while running is to allow loading new classes at runtime. This compilation consists of several necessary steps: register allocation, optimization, instruction selection and run-time support.

This text focuses on translating JVM code into a specific instruction set called MIPS. This is a RISC architecture based instruction set, having 32 general-purpose register and set of simple instructions to work with them. Specifically the text targets SPIM, a simulator able to interpret MIPS code in general environment.

On a more fine-grained look, the translation happens in several discrete steps. Firstly, we split JVM Code in separate blocks that are referred to as “high-level intermediate representation” (HIR). Edges in the HIR graph are reflecting ability of one block to transfer control to another one.

Several standard optimizations can be applied to HIR code. Among them is inlining; process of replacing a method call with the method contents. This can be used effectively only for static and short methods. Another example is constant folding and constant propagation.

Following HIR optimization, the code is translated into low-level intermediate representation (LIR). This is matching assembly language 1 to 1. The code is still aligned into blocks from HIR representation with jump labels representing each one. Register usage has to be converted into real, i.e. non-abstract, registers. Objects, arrays and character strings share a common memory representation here. For each a 12 B header is prepended, notably containing a link to table containing method links, thus allowing virtual method calls/inheritance hierarchy. In the end, an actual SPIM code representation can be produced and executed.

Authors:

Dominika Koroncziova(xkoron00)

Lukas Lichota(xlicho00)