Just-In-Time Compilation

Thiemo Bucciarelli

Institute for Software Engineering and Programming Languages

18. Januar 2016

Agenda

Definitions: Compiler

Definition: Compiler

Translation of source code from one programming language to another.

- Lexical Analysis
  Split the input into atomic units.
- Syntax Analysis
  Check if the syntax is correct (parser).
- Semantic Analysis
  E.g. type checking. Typically results in an intermediate representation.
- Optimization
  Increase the effectivity of the code without changing the semantics.

Definitions: AOT Compiler

Definition: Ahead-of-Time (AOT) Compiler

- Compilation before runtime
- Typically called by the programmer
- Often produces native code for direct execution
- Mostly platform dependent!

Definitions: Interpreters

Definition: Interpreter

- Does not create any output
- Processes at runtime
- Directly executes the code on the processor (=interpreting)
- Mostly platform independent!

Just-In-Time Compilation

Goals:

- Tries to fill the gap between Interpreter and AOT compilers
- Efficiency and platform independency

Structure:

- Performs compilation during runtime
- *Trace-based* or *method-based*
- Has to be as fast as possible
- Can use additional information for better optimizations
- Knows the underlying hardware

Just-In-Time Compilation

Method-based:

- Only analyses the calls, no further analysis
- Similar to static AOT compilers
- Less efficient, but less overhead

Trace-based:

- Analyses what paths (*traces*) are often used, and to which methods they belong
- Can effectively be combined with an interpreter
- Allows targeted optimizations
- More efficient, more overhead

Comparing JIT, AOT and Interpreters

AOT-Compilers:

- $+$ Can be arbitrary slow and thus perform time-consuming optimizations
- $+$ No overhead at runtime
- $-$ Little platform independence

Interpreters:

- $+$ Platform independence
- $\pm$ Little overhead at runtime
- $-$ Inefficient for repetitive executions
- $-$ Little optimization possibilities

JIT-Compilers:

- $+$ Platform independence
- $+$ Highly optimized and efficient code
- $+$ Runtime optimizations
- $-$ High overhead at runtime

Profiling

- Necessary for the optimization
- Analysis of the execution and collection of information
- A good profiling is crucial for efficient optimizations
- *static* or *dynamic*
- Has to produce as less overhead as possible (especially for JIT)

Next slides:

- How could a profiler be implemented?
- What information could be collected?

Profiling: How?

Sampling-based:

- Statistical approach: Take samples of the execution state
- Less precise, but very efficient
- Does not affect memory management much
- Drawback: A method could be completely undetected

Event-based:

- Profiler is triggered by certain events
- Very specified

Profiling: How?

Instrumentation:

- Injection of profiling code
- Very flexible, better performance than profiling by another thread

Profiling: What?

Call graph:

- Graph showing the call-dependencies of the methods
- Edges contain the amount of times this call was performed

Execution Trace:

- A trace representing the execution, including timestamps.
- Most precise profile

Optimization

Optimization:

- *Static* or *dynamic/adaptive*
- JIT compilers can use both types (but not every technique, because of their overhead)

Examples for static optimization:

- Dead code elimination
- Constant folding and propagation
- Loop-invariant code motion

Dynamic optimization on the basis of the *Java Virtual Machine*

JVM: Introduction

- Here: *HotSpot* and *JRockit*
- AOT compilation of Java source-code to Java bytecode
- JRockit uses JIT and HotSpot uses JIT& Interpreter
- Bytecode is assembly-like, and easy to process during runtime

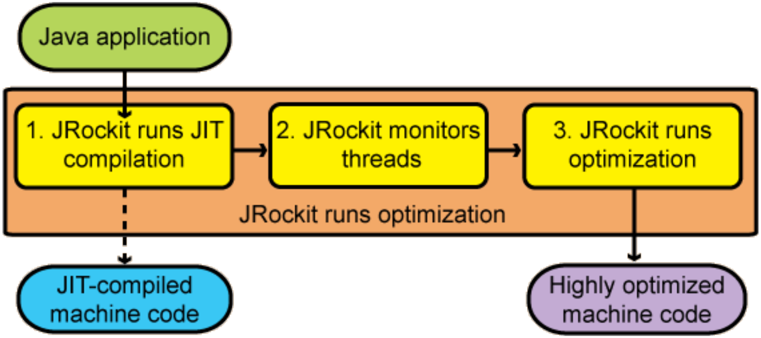
JVM: Java Bytecode

What Java bytecode looks like

```
Compiled from "test.java"
public class test {
  public test();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<
      init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: iconst_5
       1: istore_1
       2: return
}
```

JRockit structure


Java application
1. JRockit runs JIT compilation
2. JRockit monitors threads
3. JRockit runs optimization
JRockit runs optimization
JIT-compiled machine code
Highly optimized machine code

JVM: HotSpot Optimizations

Hotspot Detection:

- Selective optimization: Only optimize parts of the code
- Assumption: the execution mostly resides in a small part of the code (80/20 rule)
- For 80% of the code , interpreting is probably more efficient than compiling
- Detect hot spots and focus on these for compilation and optimization

JVM: HotSpot Optimizations

Method Inlining:

- Method invokations are time consuming
- Copy the code of frequently invokated methods inside their caller-methods
- Reduces overhead, allows more optimization

Problems:

- Recursive calls (infinite inlining, the optimizer has to set a maximum for the inlining of recursive calls)
- Overriding (see next slide)

JVM: HotSpot Optimizations

Dynamic deoptimization:

- Not every method can be inlined
- If a method is not final, it could possibly be overridden at runtime
- HotSpot speculatively inline methods which are not final (but not overridden at this moment)
- In certain cases, the inlining has to be undone, this is called dynamic deoptimization

JVM: Hotspot Optimizations

Range check elimination:

- Java requires strict array bounding checks
- Reading, changing and overwriting a value would need two checks
- the JVM can check if it is possible for the index value to change between operations
- Reduces the range check amount

JVM: When to use JIT?

- As stated in HotSpot Detection, interpreting is in some cases better than JIT
- But: How to decide whether a method should be JIT-compiled or interpreted?
- HotSpot uses heuristic approaches

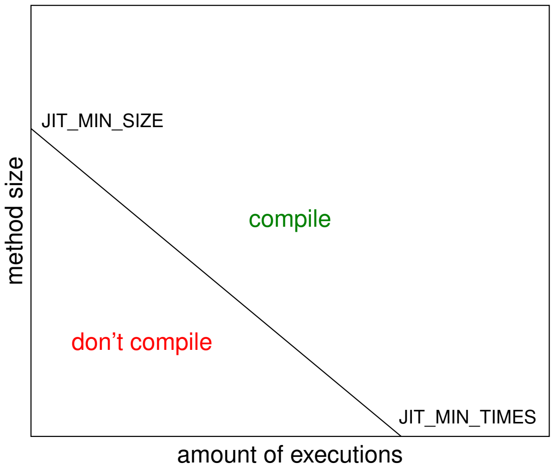
JVM: When to use JIT?

The simple heuristics

- Often executed methods should be compiled
- Large methods should also be compiled, even when they are rarely executed
- Decision is based on the amount of executions and size of the methods

Example

Example for a simple decision graph:



Conclusion

- JIT compilation gives compilers the ability to be highly platform independent
- Optimizations mostly based on assumptions, which could also have negative effects
- Important decisions: Optimize? JIT or interpret?

Questions?

Thank you for your attention.
Questions?