Java Just-in-Time Compilation

Seminar: Java vs. C++

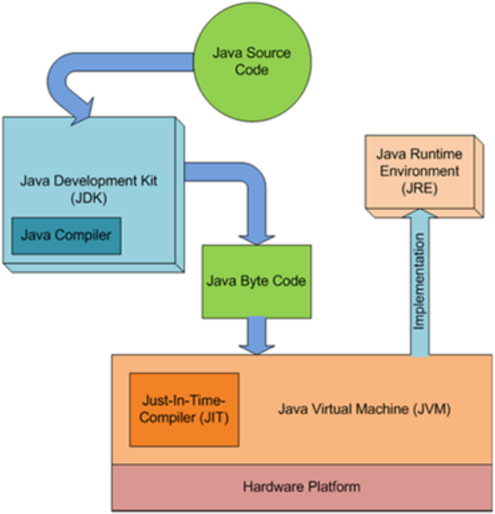
Jan Kelch
kelchj@informatik.uni-freiburg.de

Albert-Ludwigs-University Freiburg
Department of Computer Science
Chair of Algorithms and Data Structures

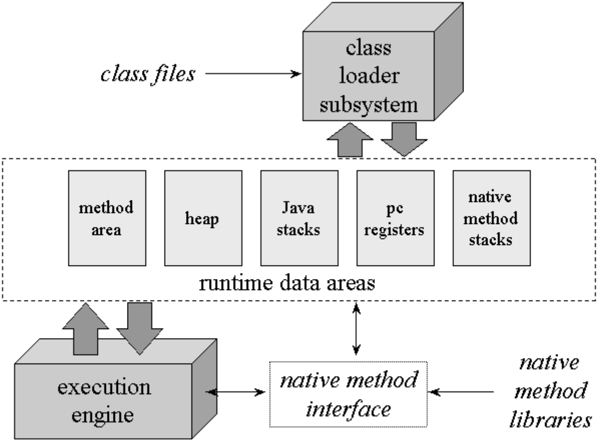
November 24, 2010

Outline

Overview – the Java components



JVM internal architecture



Execution Engine

- behavior is defined in terms of an instruction set (bytecode)
- specification describes in detail `what` todo but little about `how`

Execution Engine

- behavior is defined in terms of an instruction set (bytecode)
- specification describes in detail `what` todo but little about `how`

How could an implementation execute bytecode?

- interpret
- just-in-time compile
- execute natively in silicon
- use a combination of these
- or ... maybe someone comes up with some new techniques

Recall Compiler Structure

Frontend

1. lexical analysis (scanner)
2. syntactical analysis (parser)
3. semantical analysis

Recall Compiler Structure

Frontend

1. lexical analysis (scanner)
2. syntactical analysis (parser)
3. semantical analysis

Backend

1. generate intermediate representation (IR)
2. optimization
3. assembly code generation

Interpreter

Does only the frontend part

- lexical analysis (scanner)
- syntactical analysis (parser)
- semantical analysis

Interpreter

Does only the frontend part

- lexical analysis (scanner)
- syntactical analysis (parser)
- semantical analysis

Workflow

- reads bytecode by bytecode in a loop
- calls function associated to op-code
- or use TemplateTable (openJDK)

TemplateTable – Interpreter

- the interpreter is generated at runtime
- there are two dispatch tables
- 1. is the normal mode table
- 2. is used to bring interpreter to a safepoint
  (e.g. when a GC should be made, or synchronization)
- TT holds generator functions for each kind of bytecode
- per bytecode generate and dispatch assembly code

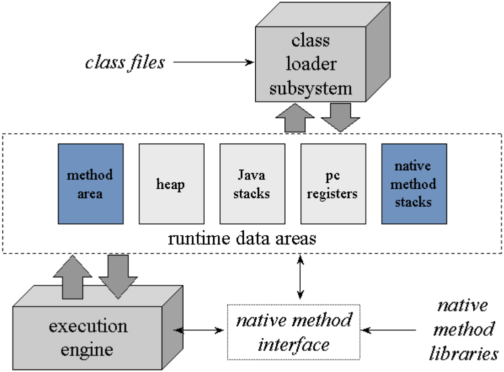
TemplateTable – Interpreter

- the interpreter is generated at runtime
- there are two dispatch tables
- 1. is the normal mode table
- 2. is used to bring interpreter to a safepoint
  (e.g. when a GC should be made, or synchronization)
- TT holds generator functions for each kind of bytecode
- per bytecode generate and dispatch assembly code

Code example – generator function

```
void TemplateTable::iconst(int value) {
 transition(vtos, itos);
 if (value == 0) { __xorl(rax, rax);
 } else { __movl(rax, value);
}}
```

Just-In-Time Compiler



- compiles bytecode to assembly
- compiles per method
- dynamic bind compiled code

Recall – Method Area

Stores method information

- the method's name
- the method's return type (or void)
- the number and types (in order) of the method's parameters
- the method's modifiers (some subset of public, private, protected, static, final, synchronized, `native`, abstract)
- the method's bytecode (in case modifier is not native or abstract)

Problems

Interpreter

- slow because of line by line model

Problems

Interpreter

- slow because of line by line model

Just-in-time compiler

- tradeoff compilationtime vs. runtime

Combination – Adaptive Compilation

Workflow of the Sun HotspotVM

1. interpreting bytecode
2. profiles code-usage
3. find hotspots
4. just-in-time compile hotspot code while still interpreting
5. caching compiled code
6. switching to/reuse compiled code

Hot Spot Detection

Hotspots

- application spends 80% of time in 20% of code
- compilation from many loop-iteration on
- compilation from many method-calls on
- many $:=$ 10.000

Method Inlining

What does it do?

- replace method call with corresponding method block

Method Inlining

What does it do?

- replace method call with corresponding method block

And why?

- JIT compilation is performed per method
- for small method reduce method invocation overhead (e.g. method which only returns a value)
- compiler gets larger blocks which significantly increases optimization

Example – Method Inlining

Code example

```
class A {
 final int foo() { return 3; }
 }
```

Example – Method Inlining

Code example

```
class A {
 final int foo() { return 3; }
 }
```

Benefit – inlining a.foo()

- no method call
- no dynamic dispatch
- possible to constant-fold the value
  (`a.foo()+2` becomes 5 with no code executed at runtime)
- because of dynamic deoptimization JVM can inline without `final`-keyword

Dynamic Deoptimization

What is the intension?

- OO-language are dynamic
  (dynamic dispatch or virtual method invocation)
- so compiled code can become incorrect til runtime

Dynamic Deoptimization

What is the intension?

- OO-language are dynamic
  (dynamic dispatch or virtual method invocation)
- so compiled code can become incorrect til runtime

What does it do?

- jit-compiler records all of the assumptions that the code makes
- so JVM can undo compilation(+optimization) to get bytecode (for interpretation/recompilation)
- JVM switch back from native to bytecode while method is still running

Example – Dynamic Deoptimization

Code example

```
class B {
 int foo() { return 3; }
 }
class C extends B {
 int foo() { return 6; }
 }
```

Example – Dynamic Deoptimization

Code example

```
class B {
 int foo() { return 3; }
 }
class C extends B {
 int foo() { return 6; }
 }
```

Result

- as long no override for `int foo()` everything is fine
- problem arises when `class C` is dynamical loaded
- code with inlined `B.foo()` is incorrect
- variable in the code of type B can point to objects of either class B or C

On Stack Replacement

What is the intension?

- hotspots (like loop-iterations) could be in functions which will be called only once
- so the compiled version would never be executed

On Stack Replacement

What is the intension?

- hotspots (like loop-iterations) could be in functions which will be called only once
- so the compiled version would never be executed

What does it do?

- the exact opposite of dynamic deoptimization
- JIT compiles code
- interpreted frame is turned into a compiled frame while method is still running

Example – On Stack Replacement

Code example

```
public class D {
 public static void main(String[] arg) {
  int sum = 0;
  for (int index = 0; index < 10*1000*1000; index += 1) {
   sum += index;
}}}
```

Example – On Stack Replacement

Code example

```
public class D {
 public static void main(String[] arg) {
  int sum = 0;
  for (int index = 0; index < 10*1000*1000; index += 1) {
   sum += index;
}}}
```

Timeline without OSR

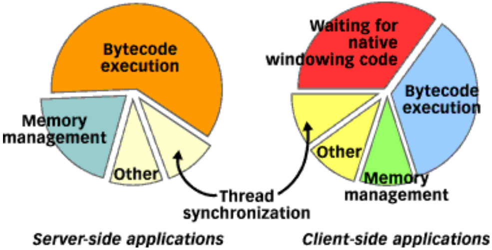
- Interpreter starts interpreting `main()` method
- Counter hits 10.000, and compilation begins, but still interpreting `main()`
- Compilation finishes, still interpreting `main()`
- `main()` finishes

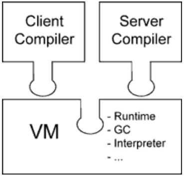
Example – On Stack Replacement

Timeline with OSR

- Interpreter starts interpreting `main()` method
- Counter hits 10.000, and compilation begins, but still interpreting `main()`
- Compilation finishes, still interpreting `main()`
- Counter hits 14.000, and interpreting stops
- `main()` is compiled a second time via OSR to allow entry in the middle of the loop
- `main()` resumes in the compiled code
- `main()` finishes

JVM time spend, Sun HotspotVM





Sun HotspotVM Compiler

Client-Compiler

is a simple, fast three-phase compiler

1. front end constructs high-level intermediate representation (HIR) from BCs

   HIR uses static single assignment (SSA)
2. platform-specific back end generates low-level intermediate representation (LIR) from HIR
3. performs register allocation on LIR and generates machine code from it

Sun HotspotVM Compiler

Server-Compiler

is a high-end fully optimizing compiler

- uses an advanced static single assignment (SSA)-based IR for optimizations
- optimizations: dead code elimination, loop invariant hoisting, common subexpression elimination, constant propagation, global value numbering, and global code motion
- java specific: null-check and range-check elimination, optimization of exception throwing paths
- register allocator is a global graph coloring allocator

Sun HotspotVM Compiler

Compiler Optimizations of both JITCs

- Deep inlining and inlining of potentially virtual calls
- Fast instanceof/checkcast
- Range check elimination
- Loop unrolling
- Feedback-directed optimizations

Time measurement

Code example

```
public class E {
 public static void main(String[] arg) {
  int sum = 0;
  for (int i = 0; i < 10*1000*1000*1000; i += 1) {
   sum += i;
  }
 }
}
```

Time measurement

Code example

```
public class E {
 public static void main(String[] arg) {
  int sum = 0;
  for (int i = 0; i < 10*1000*1000*1000; i += 1) {
   sum += i;
  }
 }
}
```

Assesments?!

```
[jan@hyperBox ~]$ vim E.java
[jan@hyperBox ~]$ javac E.java
[jan@hyperBox ~]$ time java E

real    0m1.462s
user    0m1.273s
sys     0m0.143s
[jan@hyperBox ~]$ time java -XX:+PrintCompilation -Djava.compiler=NONE E

real    0m23.500s
user    0m23.018s
sys     0m0.307s
[jan@hyperBox ~]$ time java -XX:+PrintCompilation -client E
  1       java.lang.String::hashCode (64 bytes)
  1%      E::main @ 4 (21 bytes)

real    0m1.466s
user    0m1.270s
sys     0m0.160s
[jan@hyperBox ~]$ time java -XX:+PrintCompilation -server E
  1%      E::main @ 4 (21 bytes)

real    0m0.196s
user    0m0.037s
sys     0m0.117s
[jan@hyperBox ~]$
```

Choosing Compiler – Sun HotspotVM

Using . . . compiler

- none: java -Djava.compiler=NONE [classfile]
- client: java -client [classfile]
- server: java -server [classfile]

Choosing Compiler – Sun HotspotVM

Using . . . compiler

- none: java -Djava.compiler=NONE [classfile]
- client: java -client [classfile]
- server: java -server [classfile]

Some compiler information

- java -XX:+PrintCompilation [classfile]

See what the compiler do

Tools – part of Hotspot

- IdealGraphVisualizer – tool for examining IR of server compiler
- LogCompilation tool – parse LogCompilation output of the JVM
- hsdis – disassembler used by hotspot for debugging

See what the compiler do

Tools – part of Hotspot

- IdealGraphVisualizer – tool for examining IR of server compiler
- LogCompilation tool – parse LogCompilation output of the JVM
- hsdis – disassembler used by hotspot for debugging

Tools – 3rd party

Client Compiler Visualizer: Tool for examining the HIR, LIR, and linear scan register allocation of the client compiler

References I

Bill Venners.
Book: Inside the Java Virtual Machine, Ch. 5.
http://www.artima.com/insidejvm/ed2/jvm.html.

Joe's blog.
Differentiate JVM JRE JDK JIT.
http://javapapers.com/core-java/differentiate-jvm-jre-jdk-jit/.

Oracle.
Wiki: Hotspot Tools.
http://wikis.sun.com/display/HotSpotInternals/HotSpot+Tools.

Oracle Sun Developer Network (SDN).
Article: The Java HotSpot Performance Engine: An In-Depth Look.
http://java.sun.com/developer/technicalArticles/Networking/HotSpot/index.html.

Oracle Sun Developer Network (SDN).
Tutorials & Code Camps: Chapter 8 Continued: Performance Features and Tools.
http://java.sun.com/developer/onlineTraining/Programming/JDCBook/perf2.html.

References II

Oracle Sun Developer Network (SDN).
White Paper: The Java HotSpot Performance Engine Architecture.
http://java.sun.com/products/hotspot/whitepaper.html.

osdir.com.
Sun Hotspot JVM Part 1: The Interpreter.
http://osdir.com/ml/attachments/pdf5YlauhYAr5.pdf.