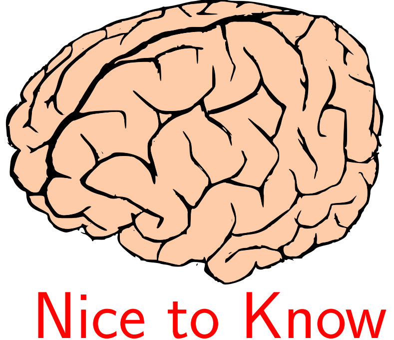
Compilers Everywhere

Bernd Müller

Ostfalia

Vortragstypus ???

Vortragstypus ???



Nice to Know

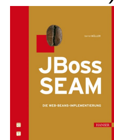
Vortragstypus ???

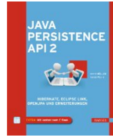
Agenda

- Was ist Just-in-Time-Compilation ?
- Beispiele (kleiner Auszug)
- Kommandozeilenparameter
- Demo

Vorstellung Referent

- Prof. Informatik (Ostfalia, HS Braunschweig/Wolfenbüttel)
- Buchautor (JSF, Seam, JPA, ...)



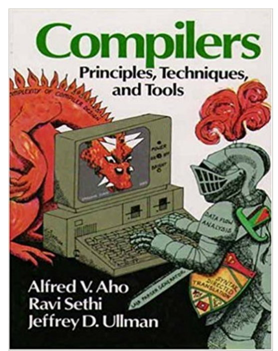


- Mitglied EGs JSR 344 (JSF 2.2) und JSR 338 (JPA 2.1)
- Geschäftsführer PMST GmbH
- JUG Ostfalen (Mitorganisator)
- ...
- bernd.mueller@ostfalia.de
- @berndmuller

(Java) Compiler Basics

Compilers

A compiler is a program that reads a program written in one programming language – the source language – and translates it into an equivalent program in another language – the target language.



Compiler (from Wikipedia)

...

The translation process influences the design of computer languages which leads to a preference of compilation or interpretation.

...

In practice, an interpreter can be implemented for compiled languages and compilers can be implemented for interpreted languages.

...

https://en.wikipedia.org/wiki/Compiler

Compiler (from Wikipedia)

...

The translation process influences the design of computer languages which leads to a preference of compilation or interpretation.

...

In practice, an interpreter can be implemented for compiled languages and compilers can be implemented for interpreted languages.

...

https://en.wikipedia.org/wiki/Compiler

The Java Language: A White Paper / An Overview, 1995

Section *Interpreted*

The Java interpreter can execute Java bytecodes directly on any machine to which the interpreter has been ported. And since linking is a more incremental and lightweight process, the development process can be much more rapid and exploratory.

The Java Language: A White Paper / An Overview, 1995

Section *High Performance*

While the performance of interpreted bytecodes is usually more than adequate, there are situations where higher performance is required. The bytecodes can be translated on the fly (at runtime) into machine code for the particular CPU the application is running on. For those accustomed to the normal design of a compiler and dynamic loader, this is somewhat like putting the final machine code generator in the dynamic loader. The bytecode format was designed with generating machine codes in mind, so the actual process of generating machine code is generally simple.
In interpreted code we're getting about 300,000 method calls per second on an Sun Microsystems SPARCStation 10. The performance of bytecodes converted to machine code is almost indistinguishable from native C or C++.

The Java Language: A White Paper / An Overview, 1995

Section *High Performance*

While the performance of interpreted bytecodes is usually more than adequate, there are situations where higher performance is required. The bytecodes can be translated on the fly (at runtime) into machine code for the particular CPU the application is running on. For those accustomed to the normal design of a compiler and dynamic loader, this is somewhat like putting the final machine code generator in the dynamic loader. The bytecode format was designed with generating machine codes in mind, so the actual process of generating machine code is generally simple.
In interpreted code we're getting about 300,000 method calls per second on an Sun Microsystems SPARCStation 10. The performance of bytecodes converted to machine code is almost indistinguishable from native C or C++.

HotSpot-Ankündigung 1999

PARIS–(BUSINESS WIRE)–April 27, 1999

Sun Microsystems, Inc. today announced the release of the Java HotSpot(TM) Performance Engine, the fastest software to date for Java(TM) technology performance. The Java HotSpot performance engine breaks new ground in software design and raises the bar by providing 100% higher performance than the previous Java platform. The Java HotSpot Performance Engine will be available free of charge for download at http://java.sun.com/products/hotspot/index.html.

"Java HotSpot Performance Engine turbocharges the Java 2 platform – performance is no longer an issue," said Jon Kannegaard, Vice President and General Manager, Java Platform at Sun Microsystems, Inc.'s Java Software. This release maintains uncompromising compatibility while dramatically boosting speed. It's now the obvious choice for deploying full-scale Java applications in the enterprise."

URL

HotSpot-Ankündigung 1999

PARIS–(BUSINESS WIRE)–April 27, 1999

Sun Microsystems, Inc. today announced the release of the Java HotSpot(TM) Performance Engine, the fastest software to date for Java(TM) technology performance. The Java HotSpot performance engine breaks new ground in software design and raises the bar by providing 100% higher performance than the previous Java platform. The Java HotSpot Performance Engine will be available free of charge for download at http://java.sun.com/products/hotspot/index.html.

"Java HotSpot Performance Engine turbocharges the Java 2 platform – performance is no longer an issue," said Jon Kannegaard, Vice President and General Manager, Java Platform at Sun Microsystems, Inc.'s Java Software. This release maintains uncompromising compatibility while dramatically boosting speed. It's now the obvious choice for deploying full-scale Java applications in the enterprise."

URL

Just-in-Time-Compilation

Hot-Spot-Compilation oder JIT-Compilation

- Idee: Performanz hängt hauptsächlich davon ab, wie schnell *häufig* ausgeführter Code ist

Hot-Spot-Compilation oder JIT-Compilation

- Idee: Performanz hängt hauptsächlich davon ab, wie schnell *häufig* ausgeführter Code ist
- Daher: späte (nicht zum Programmstart), optionale Compilierung sinnvoll

Hot-Spot-Compilation oder JIT-Compilation

- Idee: Performanz hängt hauptsächlich davon ab, wie schnell *häufig* ausgeführter Code ist
- Daher: späte (nicht zum Programmstart), optionale Compilierung sinnvoll
  - Compilierung benötigt Zeit, diese evtl vergeudet, falls Code selten aufgerufen

Hot-Spot-Compilation oder JIT-Compilation

- Idee: Performanz hängt hauptsächlich davon ab, wie schnell *häufig* ausgeführter Code ist
- Daher: späte (nicht zum Programmstart), optionale Compilierung sinnvoll
  - Compilierung benötigt Zeit, diese evtl vergeudet, falls Code selten aufgerufen
  - JVM benötigt Informationen über Ausführung, um optimalen Code zu erzeugen

Hot-Spot-Compilation oder JIT-Compilation

- Idee: Performanz hängt hauptsächlich davon ab, wie schnell *häufig* ausgeführter Code ist
- Daher: späte (nicht zum Programmstart), optionale Compilierung sinnvoll
  - Compilierung benötigt Zeit, diese evtl vergeudet, falls Code selten aufgerufen
  - JVM benötigt Informationen über Ausführung, um optimalen Code zu erzeugen
- Beispiel: `obj1.equals(obj2)`

Hot-Spot-Compilation oder JIT-Compilation

- Idee: Performanz hängt hauptsächlich davon ab, wie schnell *häufig* ausgeführter Code ist
- Daher: späte (nicht zum Programmstart), optionale Compilierung sinnvoll
  - Compilierung benötigt Zeit, diese evtl vergeudet, falls Code selten aufgerufen
  - JVM benötigt Informationen über Ausführung, um optimalen Code zu erzeugen
- Beispiel: `obj1.equals(obj2)`
  - Falls `obj1` die letzen Male immer String war, `String.equals()` verwenden und nicht dynamisch dispatchen

Hot-Spot-Compilation oder JIT-Compilation

- Idee: Performanz hängt hauptsächlich davon ab, wie schnell *häufig* ausgeführter Code ist
- Daher: späte (nicht zum Programmstart), optionale Compilierung sinnvoll
  - Compilierung benötigt Zeit, diese evtl vergeudet, falls Code selten aufgerufen
  - JVM benötigt Informationen über Ausführung, um optimalen Code zu erzeugen
- Beispiel: `obj1.equals(obj2)`
  - Falls `obj1` die letzen Male immer String war, `String.equals()` verwenden und nicht dynamisch dispatchen
  - Aber: `obj1` könnte mal von anderem Typ sein, muss also beobachtet werden

Die beiden JIT-Compiler

- Client-Compiler, auch C1 genannt

Die beiden JIT-Compiler

- Client-Compiler, auch C1 genannt
- Server-Compiler, auch C2 genannt

Die beiden JIT-Compiler

- Client-Compiler, auch C1 genannt
- Server-Compiler, auch C2 genannt
- Warum zwei ?

Die beiden JIT-Compiler

- Client-Compiler, auch C1 genannt
- Server-Compiler, auch C2 genannt
- Warum zwei ?
  - Client-Compiler optimiert, um Start-Up-Zeit zu minimieren

Die beiden JIT-Compiler

- Client-Compiler, auch C1 genannt
- Server-Compiler, auch C2 genannt
- Warum zwei ?
  - Client-Compiler optimiert, um Start-Up-Zeit zu minimieren
  - Server-Compiler optimiert, um auf lange Sicht bessere Performanz zu haben

Die beiden JIT-Compiler

- Client-Compiler, auch C1 genannt
- Server-Compiler, auch C2 genannt
- Warum zwei ?
  - Client-Compiler optimiert, um Start-Up-Zeit zu minimieren
  - Server-Compiler optimiert, um auf lange Sicht bessere Performanz zu haben
- Warum kann der Server-Compiler evtl. besser sein?

Die beiden JIT-Compiler

- Client-Compiler, auch C1 genannt
- Server-Compiler, auch C2 genannt
- Warum zwei ?
  - Client-Compiler optimiert, um Start-Up-Zeit zu minimieren
  - Server-Compiler optimiert, um auf lange Sicht bessere Performanz zu haben
- Warum kann der Server-Compiler evtl. besser sein?
  - Weil er den ausgeführten Code länger beobachtet und gründlicher analysiert hat und damit gezieltere Optimierungen einbauen kann

Tiered Compilation

Tiered Compilation

- Zu Beginn Client-Compiler verwenden, um schnellen Startup zu bekommen

Tiered Compilation

- Zu Beginn Client-Compiler verwenden, um schnellen Startup zu bekommen
- Wenn Code „*hot*“ wird und genügend Laufzeitinformationen gesammelt wurden, nochmals durch Server-Compiler übersetzen

Tiered Compilation

- Zu Beginn Client-Compiler verwenden, um schnellen Startup zu bekommen
- Wenn Code „*hot*“ wird und genügend Laufzeitinformationen gesammelt wurden, nochmals durch Server-Compiler übersetzen
- Zu setzen mit `-XX:+TieredCompilation`

Tiered Compilation

- Zu Beginn Client-Compiler verwenden, um schnellen Startup zu bekommen
- Wenn Code „*hot*“ wird und genügend Laufzeitinformationen gesammelt wurden, nochmals durch Server-Compiler übersetzen
- Zu setzen mit `-XX:+TieredCompilation`
- Seit Java 8 der Default

Tiered Compilation

- Zu Beginn Client-Compiler verwenden, um schnellen Startup zu bekommen
- Wenn Code „*hot*“ wird und genügend Laufzeitinformationen gesammelt wurden, nochmals durch Server-Compiler übersetzen
- Zu setzen mit `-XX:+TieredCompilation`
- Seit Java 8 der Default
- Tiered Compilation setzt Server-Compiler voraus.
  `java -client -XX:+TieredCompilation` läuft daher ohne Tiered Compilation

2 Compiler — 5 Ausführungs-Level (bei Tiered Compilation)

- Level 0: interpreted code

2 Compiler — 5 Ausführungs-Level (bei Tiered Compilation)

- Level 0: interpreted code
- Level 1: simple C1 compiled code (with no profiling)

2 Compiler — 5 Ausführungs-Level (bei Tiered Compilation)

- Level 0: interpreted code
- Level 1: simple C1 compiled code (with no profiling)
- Level 2: limited C1 compiled code (with light profiling)

2 Compiler — 5 Ausführungs-Level (bei Tiered Compilation)

- Level 0: interpreted code
- Level 1: simple C1 compiled code (with no profiling)
- Level 2: limited C1 compiled code (with light profiling)
- Level 3: full C1 compiled code (with full profiling)

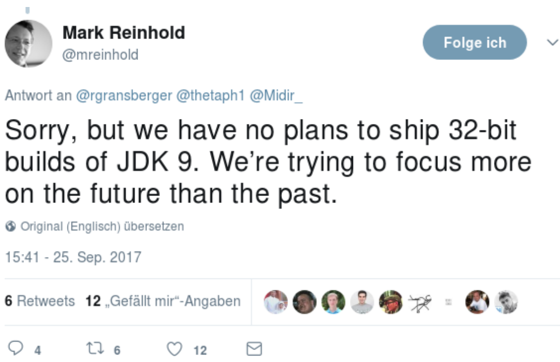
2 Compiler — 5 Ausführungs-Level (bei Tiered Compilation)

- Level 0: interpreted code
- Level 1: simple C1 compiled code (with no profiling)
- Level 2: limited C1 compiled code (with light profiling)
- Level 3: full C1 compiled code (with full profiling)
- Level 4: C2 compiled code (uses profile data from the previous steps)

2 Compiler — 5 Ausführungs-Level (bei Tiered Compilation)

- Level 0: interpreted code
- Level 1: simple C1 compiled code (with no profiling)
- Level 2: limited C1 compiled code (with light profiling)
- Level 3: full C1 compiled code (with full profiling)
- Level 4: C2 compiled code (uses profile data from the previous steps)
- Normalerweise: $0 \Rightarrow 3 \Rightarrow 4$

Seit Java 9 keine 32-Bit-JVMs mehr



Code Cache

Code Cache

- Beim Compilieren wird Assembler-Code in einem Code-Cache gehalten

Code Cache

- Beim Compilieren wird Assembler-Code in einem Code-Cache gehalten
- Dessen Größe ist einstellbar:
  `-XX:InitialCodeCacheSize=N`
  `-XX:ReservedCodeCacheSize=N`

Code Cache

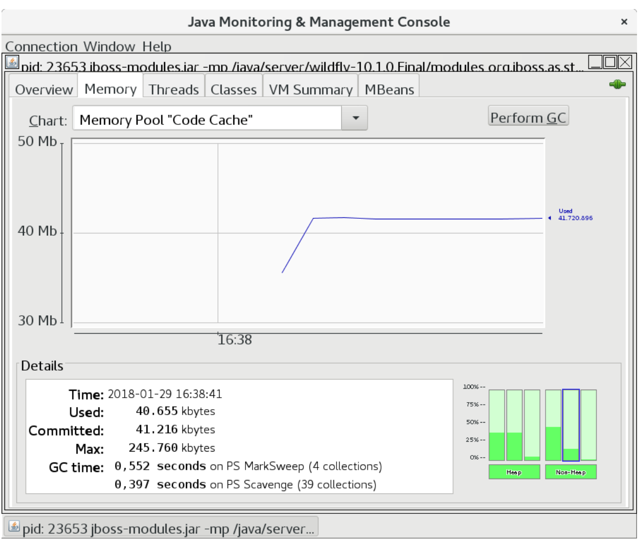
- Beim Compilieren wird Assembler-Code in einem Code-Cache gehalten
- Dessen Größe ist einstellbar:
  `-XX:InitialCodeCacheSize=N`
  `-XX:ReservedCodeCacheSize=N`
- Defaults in Java 8 Server in der Regel ausreichend

Code Cache

- Beim Compilieren wird Assembler-Code in einem Code-Cache gehalten
- Dessen Größe ist einstellbar:
  `-XX:InitialCodeCacheSize=N`
  `-XX:ReservedCodeCacheSize=N`
- Defaults in Java 8 Server in der Regel ausreichend
- Falls nicht, Meldung der Art:

```
Java HotSpot(TM) 64-Bit Server VM warning: CodeCache is full.
      Compiler has been disabled.
Java HotSpot(TM) 64-Bit Server VM warning: Try increasing the
      code cache size using -XX:ReservedCodeCacheSize=
CodeCache: size=... used=... max_used=... free=...
bounds ...
total\_blobs=... nmethods=... adapters=...
compilation: disabled (not enough contiguous free space left)
```

Code-Cache kann mit jconsole beobachtet werden



Compilation Thresholds

Compilation Thresholds

- Was ist *häufig* ausgeführter Code?

Compilation Thresholds

- Was ist *häufig* ausgeführter Code?
- Zwei Zähler
  - Anzahl Aufrufe einer Methode
  - Anzahl Rücksprünge innerhalb einer Schleife
- Setzen mit `-XX:CompileThreshold=N`
- Setzen mit `-XX:BackEdgeThreshold=N`
- Defaults in Client-Compiler: 1500 in Server-Compiler 10000
- In der Regel selten bis nie zu änder laut [Oaks], da mit diesen Werten Info für optimale Compilierung vorhanden

Nur zur Info ein paar Thresholds . . . (Java 8, 64 Bit)

```
 intx CompileThreshold                          = 10000     {pd product}
uintx IncreaseFirstTierCompileThresholdAt       = 50        {product}
 intx Tier2CompileThreshold                     = 0         {product}
 intx Tier3CompileThreshold                     = 2000      {product}
 intx Tier4CompileThreshold                     = 15000     {product}

 intx BackEdgeThreshold                         = 100000    {pd product}
 intx Tier2BackEdgeThreshold                    = 0         {product}
 intx Tier3BackEdgeThreshold                    = 60000     {product}
 intx Tier4BackEdgeThreshold                    = 40000     {product}
```

Bei Interesse: advancedThresholdPolicy.hpp

On-Stack Replacement

On-Stack Replacement

- Was tun, wenn eine Methode eine Schleife enthält, die sehr lange läuft?
- Oder eine Schleife, die nie terminiert?

On-Stack Replacement

- Was tun, wenn eine Methode eine Schleife enthält, die sehr lange läuft?
- Oder eine Schleife, die nie terminiert?
- Dann muss Schleife übersetzt werden, ohne auf den Methodenaufruf warten zu können

On-Stack Replacement

- Was tun, wenn eine Methode eine Schleife enthält, die sehr lange läuft?
- Oder eine Schleife, die nie terminiert?
- Dann muss Schleife übersetzt werden, ohne auf den Methodenaufruf warten zu können
- Wenn Back-EdgeThreshold erreicht, Compilation anstoßen

On-Stack Replacement

- Was tun, wenn eine Methode eine Schleife enthält, die sehr lange läuft?
- Oder eine Schleife, die nie terminiert?
- Dann muss Schleife übersetzt werden, ohne auf den Methodenaufruf warten zu können
- Wenn Back-EdgeThreshold erreicht, Compilation anstoßen
- Und compilierte Version anstoßen, solange Schleife noch läuft

On-Stack Replacement

- Was tun, wenn eine Methode eine Schleife enthält, die sehr lange läuft?
- Oder eine Schleife, die nie terminiert?
- Dann muss Schleife übersetzt werden, ohne auf den Methodenaufruf warten zu können
- Wenn Back-EdgeThreshold erreicht, Compilation anstoßen
- Und compilierte Version anstoßen, solange Schleife noch läuft
- Nennt man *On-stack Replacement*

On-Stack Replacement

- Was tun, wenn eine Methode eine Schleife enthält, die sehr lange läuft?
- Oder eine Schleife, die nie terminiert?
- Dann muss Schleife übersetzt werden, ohne auf den Methodenaufruf warten zu können
- Wenn Back-EdgeThreshold erreicht, Compilation anstoßen
- Und compilierte Version anstoßen, solange Schleife noch läuft
- Nennt man *On-stack Replacement*
- Anstoßen der Compilation relativ ausgeklügelt:

```
int limit = (CompileThreshold *
        (OnStackReplacePercentage - InterpreterProfilePercentage)) / 100;
```

On-Stack Replacement

- Was tun, wenn eine Methode eine Schleife enthält, die sehr lange läuft?
- Oder eine Schleife, die nie terminiert?
- Dann muss Schleife übersetzt werden, ohne auf den Methodenaufruf warten zu können
- Wenn Back-EdgeThreshold erreicht, Compilation anstoßen
- Und compilierte Version anstoßen, solange Schleife noch läuft
- Nennt man *On-stack Replacement*
- Anstoßen der Compilation relativ ausgeklügelt:

```
int limit = (CompileThreshold *
        (OnStackReplacePercentage - InterpreterProfilePercentage)) / 100;
```

- Zusammenhänge noch komplizierter und für mich ;-) und den Talk zu kompliziert

Escape Analysis

Escape Analysis

- Jeder Heap-Zugriff erfolgt über Field-Name oder (Array-)Index

Escape Analysis

- Jeder Heap-Zugriff erfolgt über Field-Name oder (Array-)Index
- Wenn Objekt in Methode erzeugt und nur dort verwendet wird, kann optimiert werden

Escape Analysis

- Jeder Heap-Zugriff erfolgt über Field-Name oder (Array-)Index
- Wenn Objekt in Methode erzeugt und nur dort verwendet wird, kann optimiert werden
- Man sagt: „*the object does not escape*“

Escape Analysis

- Jeder Heap-Zugriff erfolgt über Field-Name oder (Array-)Index
- Wenn Objekt in Methode erzeugt und nur dort verwendet wird, kann optimiert werden
- Man sagt: „*the object does not escape*“
- Mögliche Optimierungen:

Escape Analysis

- Jeder Heap-Zugriff erfolgt über Field-Name oder (Array-)Index
- Wenn Objekt in Methode erzeugt und nur dort verwendet wird, kann optimiert werden
- Man sagt: „*the object does not escape*“
- Mögliche Optimierungen:
  - *Automatic Stack Allocation*, damit kein GC nötig

Escape Analysis

- Jeder Heap-Zugriff erfolgt über Field-Name oder (Array-)Index
- Wenn Objekt in Methode erzeugt und nur dort verwendet wird, kann optimiert werden
- Man sagt: „*the object does not escape*“
- Mögliche Optimierungen:
  - *Automatic Stack Allocation*, damit kein GC nötig
  - *Scalar Replacement*, Fields auf Stack oder CPU-Register

Escape Analysis

- Jeder Heap-Zugriff erfolgt über Field-Name oder (Array-)Index
- Wenn Objekt in Methode erzeugt und nur dort verwendet wird, kann optimiert werden
- Man sagt: „*the object does not escape*“
- Mögliche Optimierungen:
  - *Automatic Stack Allocation*, damit kein GC nötig
  - *Scalar Replacement*, Fields auf Stack oder CPU-Register
- Switch `-XX:+DoEscapeAnalysis`, Default seit 7u4

Escape Analysis Source

```
// Adaptation for C2 of the escape analysis algorithm described in:
// [Choi99] Jong-Deok Shoi, Manish Gupta, Mauricio Seffano,
//          Vugranam C. Sreedhar, Sam Midkiff,
//          "Escape Analysis for Java", Procedings of ACM SIGPLAN
//          OOPSLA  Conference, November 1, 1999
...
...
typedef enum {
  UnknownEscape = 0,
  NoEscape      = 1, // An object does not escape method or thread and it i
                     // not passed to call. It could be replaced with scala
  ArgEscape     = 2, // An object does not escape method or thread but it i
                     // passed as argument to call or referenced by argumen
                     // and it does not escape during call.
  GlobalEscape  = 3  // An object escapes the method or thread.
} EscapeState;
```

Bei Interesse: escape.hpp

Deoptimization

Deoptimization

- Manche (einige/viele) Optimierungen müssen rückgängig gemacht werden
- Z.B. Unterklasse wird geladen, kann Methode überschreiben
- Nächste Tier hat compiliert (C2 muss C1-Code ersetzen)
- Ausgaben im Log:
  - made not entrant
  - made zombie

Informationen über den Compilier-Vorgang

-XX:+PrintCompilation

- `java -XX:+PrintCompilation`
- Attribute:
  - %: On Stack Replacement
  - s: Method is synchronized
  - !: Method has Execption Handler
  - b: Compilation in Blocking Mode // nicht im Hintergrund
  - n: Compilation for Wrapper to native Method
  - Compilation Tier
  - Methodenname
  - Anzahl Bytes Byte-Code
  - Deoptimization (made not reentrant, made zombie)

`-Xint` und `-Xcomp`

- `-Xint`: nur interpretieren
- `-Xcomp`: alles sofort compilieren

`-XX:+PrintInlining`

- `java -XX:+UnlockDiagnosticVMOptions -XX:+PrintInlining`
- Zeigt Infos über Inlining (kleiner Methoden)
- Durchaus nicht trivial (PrintFlagsFinal grep „Inline“)

`java -XX:+CITime`

- `java -XX:+CITime`
- Statistiken zur Compilierung werden nach JVM-Ende ausgegeben

`-XX:+LogCompilation`

- `java -XX:+UnlockDiagnosticVMOptions -XX:+LogCompilation`
- *Sehr* detaillierte Informationen werden in Datei `hotspot_pidXYZ.log` geschrieben

`-XX:+PrintOptoAssembly`

- `java -XX:+UnlockDiagnosticVMOptions -XX:+PrintOptoAssembly`
- Zeigt erzeugten Assembler an
- Benötigt Debug-VM

-XX:+CompileCommand=<method>

- `java -XX:+UnlockDiagnosticVMOptions -XX:CompileCommand=print,*Waste1.waste`
- Detailierte Informationen über Methode(n)

Ahead of Time Compilation

JEP 295: Ahead-of-Time Compilation

- JEP 295: Ahead-of-Time Compilation

JEP 295: Ahead-of-Time Compilation

- JEP 295: Ahead-of-Time Compilation
- **Summary** Compile Java classes to native code prior to launching the virtual machine.

JEP 295: Ahead-of-Time Compilation

- JEP 295: Ahead-of-Time Compilation
- **Summary** Compile Java classes to native code prior to launching the virtual machine.
- **Goals**
  - Improve the start-up time of both small and large Java applications, with at most a limited impact on peak performance.

JEP 295: Ahead-of-Time Compilation

- JEP 295: Ahead-of-Time Compilation
- **Summary** Compile Java classes to native code prior to launching the virtual machine.
- **Goals**
  - Improve the start-up time of both small and large Java applications, with at most a limited impact on peak performance.
  - Change the end user's work flow as little as possible.

Ahead-of-Time Compilation: so gehts!

- Neuer Compiler `jaotc` ab JDK 9 (nur Linux)
- Ab JDK 10 auch in Windows
- Verwendung:
  - `jaotc --output libWaste.so de.pdbm.Waste1`
  - `jaotc --output libjava.base.so --module java.base`
  - `java -XX:AOTLibrary=./libWaste.so,./libjava.base.so de.pdbm.Waste1`

Graal und Java Virtual Machine Compiler Interface (JVM CI)

Java Virtual Machine Compiler Interface (JVM CI)

- JEP 243: Java-Level JVM Compiler Interface
- **Summary** Develop a Java based JVM compiler interface (JVMCI) enabling a compiler written in Java to be used by the JVM as a dynamic compiler.

Java Virtual Machine Compiler Interface (JVM CI)

- JEP 243: Java-Level JVM Compiler Interface
- **Summary** Develop a Java based JVM compiler interface (JVMCI) enabling a compiler written in Java to be used by the JVM as a dynamic compiler.
- **Goals**
  - Allow a Java component programmed against the JVMCI to be loaded at runtime and used by the JVM's compile broker.
  - Allow a Java component programmed against the JVMCI to be loaded at runtime and used by trusted Java code to install machine code in the JVM that can be called via a Java reference to the installed code.

Java Virtual Machine Compiler Interface (JVM CI)

- JEP 243: Java-Level JVM Compiler Interface
- **Summary** Develop a Java based JVM compiler interface (JVMCI) enabling a compiler written in Java to be used by the JVM as a dynamic compiler.
- **Goals**
  - Allow a Java component programmed against the JVMCI to be loaded at runtime and used by the JVM's compile broker.
  - Allow a Java component programmed against the JVMCI to be loaded at runtime and used by trusted Java code to install machine code in the JVM that can be called via a Java reference to the installed code.
- Umgangssprachlich: Schreib deinen eigenen Compiler und „plug it in“

Graal

- C1 und C2 in JVM enthalten und in C++ geschrieben,

Graal

- C1 und C2 in JVM enthalten und in C++ geschrieben,
- Einarbeitungs- und Wartungsaufwand Albtraum

Graal

- C1 und C2 in JVM enthalten und in C++ geschrieben,
- Einarbeitungs- und Wartungsaufwand Albtraum
- Neuer Compiler *Graal* in Java geschrieben

Graal

- C1 und C2 in JVM enthalten und in C++ geschrieben,
- Einarbeitungs- und Wartungsaufwand Albtraum
- Neuer Compiler *Graal* in Java geschrieben
- JEP 317: Experimental Java-Based JIT Compiler

Graal

- C1 und C2 in JVM enthalten und in C++ geschrieben,
- Einarbeitungs- und Wartungsaufwand Albtraum
- Neuer Compiler *Graal* in Java geschrieben
- JEP 317: Experimental Java-Based JIT Compiler
- Integriert über allgemeines Java Virtual Machine Compiler Interface (JVM CI)

Graal

- C1 und C2 in JVM enthalten und in C++ geschrieben,
- Einarbeitungs- und Wartungsaufwand Albtraum
- Neuer Compiler *Graal* in Java geschrieben
- JEP 317: Experimental Java-Based JIT Compiler
- Integriert über allgemeines Java Virtual Machine Compiler Interface (JVM CI)
- AOT bereits mit Graal realisiert, also in JDK 9 Linux enthalten

Graal

- C1 und C2 in JVM enthalten und in C++ geschrieben,
- Einarbeitungs- und Wartungsaufwand Albtraum
- Neuer Compiler *Graal* in Java geschrieben
- JEP 317: Experimental Java-Based JIT Compiler
- Integriert über allgemeines Java Virtual Machine Compiler Interface (JVM CI)
- AOT bereits mit Graal realisiert, also in JDK 9 Linux enthalten
- Ab JDK 10 offiziell (experimental) verwendbar:
  `-XX:+UnlockExperimentalVMOptions -XX:+UseJVMCICompiler`

GraalVM

- https://www.graalvm.org/: GraalVM is a universal virtual machine for running applications written in JavaScript, Python, Ruby, R, JVM-based languages like Java, Scala, Kotlin, and LLVM-based languages such as C and C++.

GraalVM

- https://www.graalvm.org/: GraalVM is a universal virtual machine for running applications written in JavaScript, Python, Ruby, R, JVM-based languages like Java, Scala, Kotlin, and LLVM-based languages such as C and C++.
- Modifizierte/erweiterte JVM mit Truffle und Substrate VM

GraalVM

- https://www.graalvm.org/: GraalVM is a universal virtual machine for running applications written in JavaScript, Python, Ruby, R, JVM-based languages like Java, Scala, Kotlin, and LLVM-based languages such as C and C++.
- Modifizierte/erweiterte JVM mit Truffle und Substrate VM
- Enthält `native-image`: generate an image that contains ahead-of-time compiled Java code

GraalVM

- https://www.graalvm.org/: GraalVM is a universal virtual machine for running applications written in JavaScript, Python, Ruby, R, JVM-based languages like Java, Scala, Kotlin, and LLVM-based languages such as C and C++.
- Modifizierte/erweiterte JVM mit Truffle und Substrate VM
- Enthält `native-image`: generate an image that contains ahead-of-time compiled Java code
- Zu beachten:
  - Reflection (Graal SDK unterstützt)
  - `sun.misc.Unsafe`: Field-Offsets müssen im Image neu berechnet werden
  - Nicht vollständiger Classpath: AOT muss alle referenzierten Klassen eager laden, um übersetzen zu können. Ohne AOT kann Programm laufen, wirft aber evtl. Exception. Kann mit Option `--report-unsupported-elements-at-runtime` verhindert werden

Es geht immer weiter . . .

- Zum Abschluss die gute Nachricht — es geht immer weiter ;-)
- JEP draft: JWarmup precompile java hot methods at application startup (5/2018)

Es geht immer weiter ...

- Zum Abschluss die gute Nachricht — es geht immer weiter ;-)
- JEP draft: JWarmup precompile java hot methods at application startup (5/2018)
- **Summary** JWarmup overcomes Java application warmup performance problem caused by JIT threads compete with normal java threads for CPU resource at same time when both the application (requests) loads up at peak and JIT kicks in for compiling tasks. By precompiling java hot methods during warmup, JWarmup can successfully improve peak time performance degradation.

Es geht immer weiter ...

- Zum Abschluss die gute Nachricht — es geht immer weiter ;-)
- JEP draft: JWarmup precompile java hot methods at application startup (5/2018)
- **Summary** JWarmup overcomes Java application warmup performance problem caused by JIT threads compete with normal java threads for CPU resource at same time when both the application (requests) loads up at peak and JIT kicks in for compiling tasks. By precompiling java hot methods during warmup, JWarmup can successfully improve peak time performance degradation.
- **Goals** Pre-compile java hot methods to reduce CPU usage for java application at load up peak time.

Es geht immer weiter . . .

- Zum Abschluss die gute Nachricht — es geht immer weiter ;-)
- JEP draft: JWarmup precompile java hot methods at application startup (5/2018)
- **Summary** JWarmup overcomes Java application warmup performance problem caused by JIT threads compete with normal java threads for CPU resource at same time when both the application (requests) loads up at peak and JIT kicks in for compiling tasks. By precompiling java hot methods during warmup, JWarmup can successfully improve peak time performance degradation.
- **Goals** Pre-compile java hot methods to reduce CPU usage for java application at load up peak time.
- Kurzgefasst: Vor Produktion Testlauf, der Daten sammelt und speichert. Diese beim richtigen Lauf für JIT nutzen. Produktion läuft dann mit nativem Code

Und was nützt mir das jetzt alles ?

Seeing is believing . . . Demo Time !

Shell-Script zur Diagrammerzeugung:

Fragen und Anmerkungen

Referenzen

[Oaks] Scott Oaks, Java Performance — The Definitive Guide. O'Reilly, 2014.

Und Google ;-)