Unbekannte Kostbarkeiten des SDK Heute: Just-in-Time-Compilation

Bernd Müller, Ostfalia

Das Java SDK enthält eine Reihe von Features, die wenig bekannt sind. Wären sie bekannt und würden sie verwendet, könnten Entwickler viel Arbeit und manchmal sogar zusätzliche Frameworks einsparen. Wir wollen in dieser Reihe derartige Features des SDK vorstellen: die unbekannten Kostbarkeiten.

Der Just-in-Time-Compiler leistet seit zwei Dekaden Unglaubliches: Wir müssen uns über performanten Code (fast) keine Gedanken machen und haben trotzdem eine meist hoch performante Ausführung unserer Systeme garantiert. Der Just-in-Time-Compiler (kurz JIT) arbeitet im Verborgenen und erspart uns aufgrund der JVM-Ergonomics eine manuelle Konfiguration. Unter Ergonomics versteht man die von der JVM an Anwendung und Umgebung angepassten, automatisch durchgeführten Optimierungen zur Performanzsteigerung. Dazu gehören Einstellungen für die Garbage Collection, die Heap-Größe, aber eben auch der JIT-Compilation. Dieser Artikel gibt einen kleinen Einblick in die sehr unübersichtliche Welt der JIT-Compilation und hinterlässt (hoffentlich) das gute Gefühl, eine sehr reife Ablaufumgebung für unsere Applikationen zu haben.

Unbekannte Kostbarkeiten reloaded

Von 2011 bis 2015 haben wir in unserer Kolumne *Unbekannte Kostbarkeiten* insgesamt 14 Artikel in der Java aktuell veröffentlicht [1]. Mit dem Release von Java 8 änderte sich sowohl das Leserinteresse als auch die Publikationslandschaft: Features wie Lambdas und Streams dominierten die Medien und Konferenzen, der Rest war

Datentyp	Flag-Name	Flag-Wert	Kategorie
bool	C1ProfileInlinedCalls	true	C1 product
bool	DebugInlinedCalls	true	C2 diagnostic
intx	FreqInlineSize	325	pd product
bool	IncrementalInline	true	C2 product
bool	Inline	true	product
ccstr	InlineDataFile	-	pd product
intx	InlineSmallCode	2000	pd product
bool	InlineSynchronized-Methods	true	C1 product
intx	MaxInlineLevel	9	product
intx	MaxInlineSize	35	product
intx	MaxRecursiveInlineLevel	1	product
intx	Tier23InlineeNotify-FreqLog	20	product
bool	UseInlineCaches	true	product
bool	UseInlineDepthFor-SpeculativeTypes	true	C2 diagnostic
bool	UseOnlyInlinedBimorphic	true	C2 product

Tabelle 1: VM-Flags für das Inlining

eher außen vor. Auch mit Java 9 war eine Konzentration der Medien auf das Thema *Modul-System* klar zu erkennen. Andere Themen waren von weniger Interesse. Wir haben daher mit unseren *„Unbekannten Kostbarkeiten"* eine kleine Pause eingelegt. Jetzt, mit den schnellen Versionswechseln von Java 10 und danach, ist die Zeit reif, sich wieder mit den unbekannten Kostbarkeiten zu beschäftigen: Zu groß ist die Gefahr, dass wir tolle Features, die uns Java zur Verfügung stellt, nicht verwenden, weil wir sie nicht kennen.

Unser heutiger, erster Beitrag in den *„Unbekannten Kostbarkeiten reloaded"* besitzt eine Sonderstellung. Wir müssen nichts tun, um den JIT-Compiler zu verwenden, er ist immer da und funktioniert auch ohne unsere Mitwirkung ausgesprochen gut. Dieser Artikel ist also eher ein Blick hinter die Kulissen als ein Tipp oder eine Anleitung für die explizite Verwendung.

Die Anfänge des JIT

Als Java im Jahr 1995 das Licht der Welt erblickte, veröffentlichte Sun das Whitepaper *„The Java Language: An Overview"*, in dem man die folgende Aussage lesen konnte: *„The bytecodes can be translated on the fly (at runtime) into machine code for the particular CPU the application is running on"*. Obwohl die Java-Implementierung 1.0 des Jahres 1995 den Java-Quellcode in Byte-Code kompilierte und dieser dann in der JVM interpretiert wurde, existierte die Idee der Just-in-Time-Kompilierung bereits. Im Dezember 1998 war die Zeit dann reif. Mit der Version 1.2 erhielt die JVM einen JIT-Compiler und Sun veröffentlichte folgende Pressemitteilung: *„Java HotSpot Performance Engine turbocharges the Java 2 platform – performance is no longer an issue"*. Die JVM wurde mit dem Zusatz „Hot Spot" versehen, da die JIT-Kompilierung nur für „heiße", also besonders häufig aufgerufene Code-Stücke erfolgte. Verglichen mit heute war das damalige Wissen um die effiziente Implementierung von JIT-Mechanismen geradezu prähistorisch. Was der JIT-Compiler heute alles leistet, schauen wir uns nun an. Wir können allerdings nur an der Oberfläche kratzen und beschränken uns auf ein paar wenige, wie wir hoffen aber interessante Themen: Inlining, On-Stack Replacement und Escape Analysis.

Inlining

Unter Inlining versteht man das Ersetzen des Codes eines Methodenaufrufs durch den Methodenrumpf. Dies erspart zur Laufzeit den Verwaltungsaufwand für die Methodenaufrufe (Stichwort Call Stack) und ermöglicht weitere Optimierungen, da der neu kombinierte Code eventuell Vereinfachungen zulässt, die sonst nicht möglich gewesen wären. Moderne JVMs unterstützen das Inlining in verschiedenen Ausprägungen, die durch Kommandozeilen-Flags parametrisiert sind.

Die *Tabelle 1* zeigt die Ausgabezeilen eines VM-Aufrufs mit den Parametern `-XX:+UnlockDiagnosticVMOptions` und `-XX:+PrintFlagsFinal`, die „Inline" enthalten. Also die VM-Flags, die etwas mit Inlining zu tun haben.

Die erste Spalte gibt den Datentyp innerhalb der VM an, die zweite Spalte den Namen des Flag. Die dritte Spalte nennt den aktuellen Wert des Flag und die vierte Spalte dessen Kategorie. Offizielle VM-Optionen, die also auf allen Java-VMs definiert sind, sind mit "product" gekennzeichnet, plattformabhängige Flags mit "pd" (platform dependent), da sie sich zum Beispiel auf Intel/AMD und ARM unterscheiden. Debugging-relevante Flags sind mit "diagnostic" gekennzeichnet. Die Unterscheidung, zu welchem JIT-Compiler das Flag gehört, erfolgt über C1 (Client) und C2 (Server). Dazu später mehr.

Wie ist das zu interpretieren beziehungsweise zu nutzen? Das Inlining häufig aufgerufener Methoden wird bis zu einer Größe von 325 Bytes vorgenommen (Flag `FreqInlineSize`). Für selten aufgerufene Methoden wird das Inlining nur durchgeführt, falls sie kleiner als 35 Bytes (Flag `MaxInlineSize`) sind. Soll dies auch bei größeren Methoden durchgeführt werden, so muss die VM zum Beispiel mit der Option `XX:MaxInlineSize=70` aufgerufen werden, was die Methodengröße verdoppelt. Wie wir am Anfang des Artikels bereits erwähnt haben, ist die Verwendung der JIT-Optionen für die meisten Entwickler nicht empfehlenswert, da die JVM-Ergonomics in der Regel besser sind als unsere vermeintlichen Verbesserungsversuche. Hunt und John *[2]* raten sogar explizit von der Verwendung der Optionen ab.

On-Stack Replacement

Anders als beim Inlining werden ganze Methoden vollständig in Maschinen-Code übersetzt, wenn sie „hot" sind, also besonders häufig aufgerufen wurden. Im Client-Compiler ist der Schwellenwert 1.500, beim Server-Compiler 10.000 Aufrufe. Wird eine Methode derart übersetzt, erfolgt der *nächste* Aufruf der Methode über den Maschinen-Code, nicht den Byte-Code. Was passiert aber bei sehr lange laufenden Schleifen? Hier kann nicht auf den nächsten Aufruf des Schleifen-Codes gewartet werden, der eventuell in sehr weiter Zukunft liegt oder sogar nie erfolgt. Der Übergang von der Ausführung vom Byte-Code einer Schleife zum Maschinen-Code wird *„On-Stack Replacement"* genannt. Neben einem Zähler für die Aufrufhäufigkeit einer Methode gibt es auch einen Zähler, der die Rücksprünge innerhalb einer Schleife zählt, den sogenannten *„BackEdge Counter"*.

Wird dieser übersprungen, kompiliert der JIT-Compiler nicht eine gesamte Methode, sondern nur die Schleife, und beginnt die Ausführung des kompilierten Codes. Die Schwellenwerte für Methodenaufrufe und Schleifenrücksprünge lassen sich mit `XX:CompileThreshold=N` und `XX:BackEdgeThreshold=N` setzen.

Escape Analysis

Ein alloziertes Objekt *„escaped"*, wenn es nach seiner Allokation in einem Thread A durch einen anderen Thread B gesehen (zugegriffen) werden kann. Falls dies nicht der Fall ist, kann der Compiler eine oder mehrere der folgenden Optimierungen durchführen:

- Object Explosion: Anlegen der Felder des Objekts an verschiedenen Stellen und potenzieller Verzicht auf die Objektallokation selbst
- Scalar Replacement: Speichern der Felder des Objekts in CPU-Registern
- Thread Stack Allocation: Speichern der Felder des Objekts in einem Stack-Frame
- Synchronisierung eliminieren
- GC Read/Write-Barriers eliminieren

Die Escape Analysis wurde in Java 6u14 eingeführt und ist seit Java 7u4 der Default. Gesteuert wird sie über das VM-Flag `XX:+DoEscapeAnalysis`. Das Flag ist boolesch und wird mit `+` ein-, mit `-` ausgeschaltet.

C1, C2, Graal

Nachdem wir uns drei zentrale Optimierungsmöglichkeiten des JIT-Compilers angeschaut haben, wollen wir einen Blick auf den JIT-Compiler selbst werfen. Es gibt nicht nur einen, es gibt drei: C1, C2 und Graal. C1 und C2 sind der sogenannte Client- und der Server-Compiler und seit vielen Jahren in der VM enthalten. Die zentrale Idee der Differenzierung ist, dass Client-Anwendungen – zu der Zeit der Einführung zum Beispiel AWT-("Abstract Window Toolkit"), später Swing-Anwendungen auf dem Client – schnell hochfahren müssen, um den Benutzer nicht warten zu lassen. Der Schwellenwert für die Methoden-Compilation beträgt daher 1.500 Methodenaufrufe und findet ohne weiteres Monitoring der Aufrufe statt. Bei Server-Anwendungen ist die Startzeit nicht so kritisch, sodass zunächst Methodenaufrufe beobachtet und analysiert werden und der JIT darauf basierend dann besseren Code erzeugen kann als ohne diese Monitoring-Informationen. Hier beträgt der Schwellenwert 10.000 Methodenaufrufe. Der Client-Compiler war zunächst der Client-VM, also der 32-Bit-VM vorbehalten. Genauso verhielt es sich mit dem Server-Compiler, der in Server-VMs mit 64 Bit enthalten war. Später wurden beide Compiler zugleich verbaut und die sogenannte *Tiered Compilation* eingeführt, bei der zunächst der Client-, später der Server-Compiler denselben Code noch einmal übersetzt hat. Dieses Verhalten ist seit Java 8 der Default.

Mit dem JEP 317 *[3]* wurde ein zunächst experimenteller JIT-Compiler, genannt *Graal*, eingeführt. Motivation für Graal war zum einen die schlechte Wartbarkeit des in C++ geschriebenen C2-Compilers, zum anderen die Realisierung eines Ahead-of-Time-Compilers mit JEP 295 *[4]*. Graal ist seit JDK 9 in Linux, seit JDK 10 auch in Windows verfügbar. Er ist jedoch als experimentell gekennzeichnet und muss explizit freigeschaltet werden. Hierzu wurde mit JEP 243 *[5]* ein VM-Compiler-Interface realisiert, um beliebige JIT-Compiler über einen Plug-in-Mechanismus der VM zur Verfügung zu stellen. Der JEP ist mit *JVM Compiler Interface* überschrieben, sodass die entsprechende Option `XX:+UseJVMCICompiler` lautet. Durch den experimentellen Charakter des Compilers müssen die experimentellen Optionen zusätzlich durch `XX:+UnlockExperimentalVMOptions` verfügbar gemacht werden.

Zusammenfassung

Wir haben uns die drei Just-in-Time-Compiler C1, C2 und Graal angeschaut. Man muss allerdings gestehen, dass dies nur sehr oberflächlich und exemplarisch an den drei Themenkomplexen Inlining, On-Stack Replacement und Escape Analysis erfolgt ist. Javas JIT-Compiler sind sehr hochentwickelte Komponenten des Java-Laufzeitsystems, die, basierend auf Ergonomics, ohne unser Zutun sehr performanten Code erzeugen. In der Regel müssen wir uns daher keine Gedanken über performanten Code machen und können uns auf performante Algorithmik und gute Architektur konzentrieren.

Referenzen

[1] *https://www.pdbm.de/unbekannte-kostbarkeiten.html*

[2] Charlie Hunt und Binu John. Java Performance. Addison-Wesley, 2012.

[3] JEP 317: Experimental Java-Based JIT Compiler. *http://openjdk.java.net/jeps/317*

[4] JEP 296: Ahead-of-Time Compilation. *http://openjdk.java.net/jeps/295*

[5] JEP 243: Java-Level JVM Compiler Interface. *http://openjdk.java.net/jeps/243*

Bernd Müller
bernd.mueller@ostfalia.de

Nach seinem Studium der Informatik und der Promotion arbeitete Bernd Müller für die IBM und die HDI Informationssysteme. Er ist Professor, Geschäftsführer, Autor mehrerer Bücher zu den Themen JSF und JPA, sowie Speaker auf nationalen und internationalen Konferenzen. Er engagiert sich im iJUG und speziell in der JUG Ostfalen.