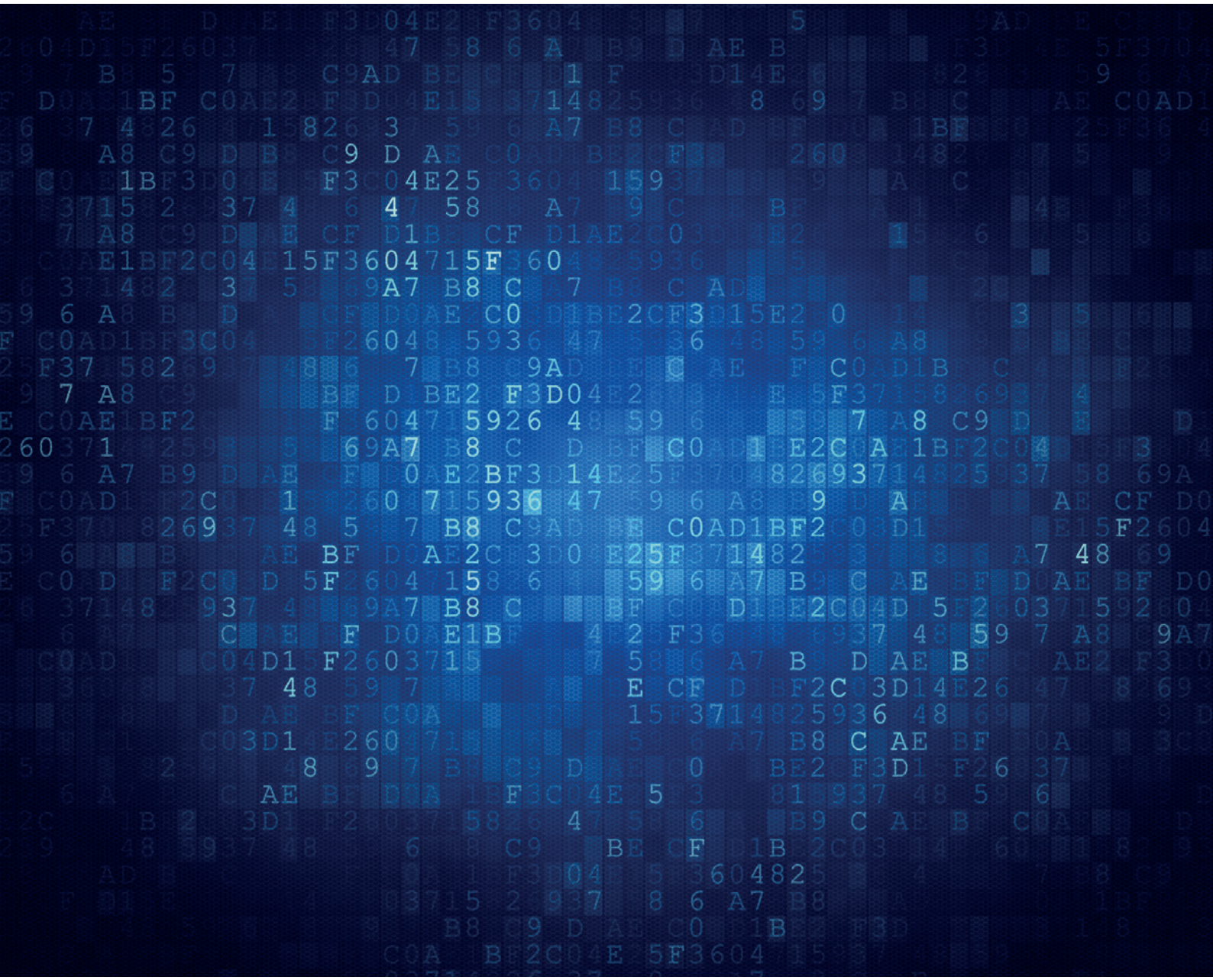

Eine JVM für die Cloud: die GraalVM

Karine Vardanyan und Stephan Rauh, OPITZ CONSULTING Deutschland

Seit Anfang 2019 sorgt Oracles neue Virtual Machine, die GraalVM für Aufsehen. Für Java-Entwickler ist sie ein Ahead-of-Time-Compiler und verspricht, polyglott und hochperformant zu sein. Doch kann sie sich wirklich gegen Javas Just-in-Time-Compiler durchsetzen? Welche Vor- und Nachteile bietet sie und wie schlägt sie sich in der Cloud? In diesem Artikel werfen wir einen detaillierten Blick auf diese universelle Virtual Machine und was man mit ihr so alles machen kann. Kann sie ihrem Namensbezug, dem heiligen Gral, gerecht werden?

Die Geschichte des Java-Compilers reicht über zwanzig Jahre zurück. Die ersten Versionen von Java waren bemerkenswert langsam. Mit der Veröffentlichung des HotSpot-Compilers im April 1999 änderte sich die Situation. Beispielsweise zeigt eine Performance-Messung *[1]*, dass Java zwischen den Versionen 1.1.7 und 1.2 um den Faktor 40 schneller wurde. Wie so oft hängen die genauen Zahlen stark vom jeweiligen Anwendungsfall ab. Fest steht, dass Java seinerzeit dramatisch schneller wurde. Es wurde erstmals möglich, Performance-intensive Anwendungen, wie zum Beispiel Text-Editoren, in Java zu schreiben. Beispielsweise hat das Software-Unternehmen Borland seine IDE JBuilder im Jahr 1999 von Delphi auf Java migriert.

Eine kurze Geschichte der Compiler

Der Compiler wurde auf eine damals sehr ungewöhnliche Weise implementiert. Um den Unterschied zu erklären, machen wir einen kurzen Ausflug in die Geschichte.

Traditionelle Compiler übersetzen Quelltext in Maschinencode. Erst dadurch kann der Computer das Programm ausführen. Solch ein Ahead-of-Time-Compiler (kurz: AOT-Compiler) braucht üblicherweise sehr lang zum Kompilieren, dafür läuft das Programm anschließend sehr schnell. Programmiersprachen wie C und C++ verwenden diesen Ansatz. Das macht sie prädestiniert für die Entwicklung von Betriebssystemen, Computerspielen und „embedded Systems“: Wenn ein Autofahrer bremst, hilft ihm mit großer Wahrscheinlichkeit ein C-Programm dabei.

Eine andere Strategie ist es, ein Programm zu schreiben, das den Quelltext des Entwicklers direkt lesen und ausführen kann. Lange Zeit wurde JavaScript so implementiert. Der Vorteil eines solchen Interpreters ist, dass der zeitfressende Kompilierschritt entfällt. Dafür läuft das Programm sehr viel langsamer. In den 90er Jahren galt als Faustregel, dass ein (interpretiertes) BASIC-Programm circa hundert- bis tausendmal langsamer ist, als wenn es mit einem guten C-Compiler geschrieben worden wäre.

Die frühen Java-Versionen verwendeten eine dritte Strategie. Sie übersetzten das Programm in einen „Bytecode“. Das ist eine Programmiersprache, die von einem Interpreter ausgeführt wird, aber sehr viel besser für die Arbeitsweise einer CPU optimiert ist als der ursprüngliche Quelltext. Der Bytecode-Interpreter ist erheblich schneller als ein normaler Interpreter und hat noch weitere Vorteile. SUN hatte Bytecode-Interpreter für viele verschiedene CPUs und Betriebssysteme geschrieben. Java-Programme laufen heutzutage auf einer Vielzahl von Plattformen – vom Großrechner über PC bis zum Android-Handy. Sie werden für eine „virtuelle Maschine“ geschrieben; diese virtuelle Maschine wird für die jeweilige Zielhardware angepasst und erlaubt es, ein und dasselbe Programm quasi überall auszuführen. Detailliertere Informationen über Bytecode sind im Blog des Autors Rauh *[3]* zu finden.

Der Just-in-Time-Compiler

Die virtuelle Maschine und das „write once, run anywhere“-Versprechen waren das Killer-Argument für den Einsatz von Java. SUN konnte den Bytecode also nicht durch einen traditionellen Compiler ersetzen. Stattdessen wurde die virtuelle Maschine um den „Just-in-Time“-Compiler ergänzt. Die Idee ist dabei, dass das Programm erst während der Programmausführung kompiliert wird. Das kann man sich wie einen Cache vorstellen: Am Ende des Tages führt der Interpreter ebenfalls Maschinencode aus. Diesen Maschinencode kann man cachen. Damit entfällt der zeitaufwendige Übersetzungsschritt. Wenn ein Algorithmus zum ersten Mal ausgeführt wird, ist er wie gewohnt langsam, bei der nächsten Wiederholung nimmt er jedoch Fahrt auf.

Dieser Effekt ist sehr eindrucksvoll in der Performancemessung sichtbar, die einer der Autoren (Stephan Rauh) vor einigen Jahren auf seinem Blog *[4]* durchgeführt hat *(siehe Abbildung 1)*. Die Grafik zeigt auf der Y-Achse die Zeit, die der Testalgorithmus braucht. Auf der X-Achse ist die Anzahl der Wiederholungen dargestellt. Bei jeder Wiederholung wird der Algorithmus etwas schneller. Erst ab zehntausend Wiederholungen ändert sich nicht mehr viel.

Zwanzig Jahre Optimierungen

Abbildung 1 zeigt eines ganz deutlich: Es ist nicht bei dem einfachen Cache geblieben. Der Compiler optimiert den Maschinencode permanent. Er konzentriert sich dabei auf die Stellen, die oft aufgerufen werden, die quasi „heißgelaufen“ sind. Er wurde konsequenterweise „HotSpot-Compiler“ getauft. Die Tipps und Tricks, die der Hot-Spot-Compiler anwendet, sind äußerst umfangreich. Zu diesem Thema empfehlen die Autoren die Vorträge von Charles Nutter *[5]*, die unter anderem Loop Unrolling, Escape Analysis und viele weitere Ideen umfassen.

Viele Getter und Setter in Programmen überleben nicht lange. Sie sind die ersten Opfer des „Inlining“. Der HotSpot-Compiler merkt, dass die Getter und Setter nur triviale Logik enthalten, und ersetzt sie kurzerhand durch den direkten Zugriff auf das Attribut. Allein durch diesen kleinen Trick wird das Programm erheblich effizienter.

Den Rest können wir uns denken: Im Laufe der letzten zwanzig Jahre wurde ununterbrochen am HotSpot-Compiler entwickelt. Java-Programme laufen heute erheblich schneller als früher. Auch das sehen wir an der Grafik, die die Entwicklung zwischen Java 1.2 und Java 1.7 zeigt. In der Zeit danach ist die Entwicklung keineswegs stehengeblieben. Laut Open Hub hat der Compiler von Java 7 ganze 734.000 Zeilen, der Compiler von Java 9 bereits 1,34 Millionen. So beeindruckend der Erfolg der HotSpot-JVM ist, sie ist vermutlich nicht der richtige Ort, um radikal neue Ideen auszuprobieren. Das Risiko ist einfach zu groß, und es dürfte auch gar nicht so einfach sein. Cliff Click – einer der Entwickler des HotSpot-Compilers C2 – sagt [11, Folie 40], dass er heutzutage keinen Compiler mehr in C oder C++ schreiben würde. C++-Programme neigen dazu, komplex und schwer wartbar zu werden. Laut Chris Seaton *[9]* ist der C2-Compiler – wie nach 20 Jahren nicht anders zu erwarten – schwer zu warten und zu erweitern.

Polyglot Programming

Dabei gibt es durchaus wichtige Entwicklungen, die regelrecht nach einem neuen Compiler rufen. Vor einigen Jahren wurde beispielsweise die Idee des „isomorphen JavaScript“ populär. Fast alle Benutzeroberflächen werden heutzutage in JavaScript implementiert. Auf dem Server hat sich dagegen Java etabliert. Für Entwickler von Eingabeformularen ergibt sich damit ein Problem. Sie müssen Validierungen und Plausibilitäten doppelt implementieren. Einmal auf dem Client – um dem Anwender sofortiges Feedback zu geben –

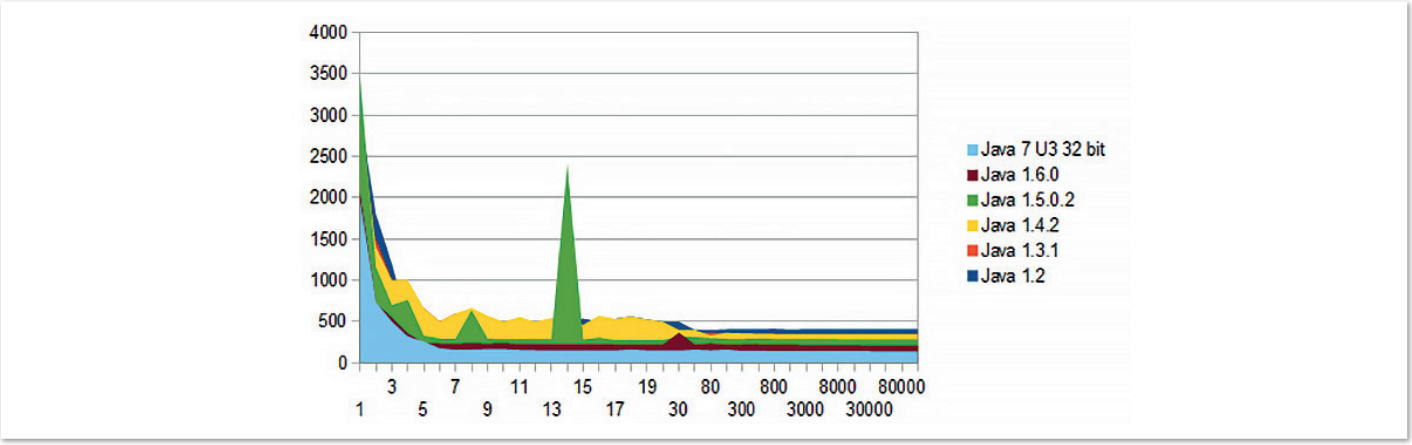

Abbildung 1: Performancegewinne durch den JIT-Compiler im Laufe der Zeit (© Stephan Rauh)

und zum anderen aus Sicherheitsgründen auf dem Server. Es wäre also schön, auch auf dem Server JavaScript ausführen zu können. Dann würde eine Implementierung reichen.

Mit den Projekten Nashorn und Rhino gibt es diese Möglichkeit in aktuellen Java-Projekten. JavaScript fühlt sich in diesen Umgebungen aber immer noch wie ein Fremdkörper an und die Performance reicht nicht an die im Browser heran.

Polyglot Compiler

Hier kommt die GraalVM ins Spiel. Streng genommen ist der Name „GraalVM" irreführend: Aus Sicht eines Java-Entwickler handelt es sich um einen neuen Compiler. Der Rest der JVM wird unverändert weiterverwendet. Man kann mit der GraalVM jedoch auch JavaScript-Anwendungen laufen lassen. In diesem Fall fühlt sich die GraalVM so an, als wäre sie ein Ersatz für Node.js und für npm. Bei Experimenten haben die Autoren es sogar geschafft, die Angular-CLI zu installieren. Sie konnten mithilfe der CLI ein Angular-Programm bauen und kompilieren. Das lässt hoffen, dass die Warnungen bezüglich der Node.js-Kompatibilität auf der GraalVM-Homepage *[8]* übertrieben sind.

Als wäre das noch nicht spannend genug, bietet die GraalVM auch Support für Python, R, Ruby, Scala und den LLVM-Bitcode. Das wiederum bedeutet, dass auch Sprachen wie C/C++ und Rust auf der GraalVM laufen, wenn auch über den Umweg eines LLVM-Compilers.

Language-Agnostic Compiler

Diese Programmiersprachen kommen aus denkbar unterschiedlichen Traditionen. Einige dieser Sprachen sind schon aus der JVM-Welt bekannt. Scala und JRuby generieren Bytecode und können damit auf der JVM laufen und – innerhalb gewisser Grenzen – sogar mit Programmteilen, die in anderen Sprachen geschrieben wurden, zusammenarbeiten. Ganz reibungslos war das allerdings nie: Die JVM ist und bleibt für Java optimiert und mit Ausnahme des Bytecodes invokedynamic wurde die JVM nie für andere Programmiersprachen erweitert. Das zwingt die Sprachentwickler zu Klimmzügen, die sich typischerweise in der Performance und/oder dem Speicherverbrauch niederschlagen.

Die GraalVM geht einen anderen Weg. Auch hier gibt es wieder einen zweistufigen Prozess: Der Quelltext wird in einen Graphen übersetzt und aus diesem wird der Maschinencode generiert. Den Graphen kennen einige Entwickler bereits aus anderen Kontexten unter dem Namen „Abstract Syntax Tree", kurz „AST". Wer das zum ersten Mal sieht, mag vielleicht ein wenig verblüfft sein: Jeder Programmcode kann als Graph dargestellt werden. Tatsächlich ist der AST das Kernstück praktisch jeden Compilers. Auf die Einzelheiten einzugehen, würde den Rahmen dieses Artikels sprengen; für einen Einstieg empfehlen die Autoren einen Blick in die englischsprachige Wikipedia-Seite *[13]* und in die ersten paar Dutzend Seiten des berühmten „Drachenbuches" *[14]*.

Die Besonderheit des Graal-Graphen ist, dass er nicht auf eine einzige Programmiersprache beschränkt ist. Bei Graal verwenden alle Programmiersprachen denselben AST. Sie haben damit ein einheitliches Framework, um neue Sprachen zu entwickeln. Für Sprachentwickler ist das ein sehr attraktives Feature.

Es ist auch relativ einfach, eigene Optimierungen zu implementieren. Der Compiler ist in Java geschrieben und durch den Graphen ist es ziemlich einfach, Optimierungen zu entwerfen. Chris Seatons Vortrag *[9]* zeigt die Idee anhand einiger Beispiele.

```
@CoreMethod(names = "+", required = 1)
public abstract static class AddNode
   extends BignumCoreMethodNode {

   @Specialization
    protected long addWithOverflow(int a, int b) {
        return (long) a + (long) b;
    }
  }
```

Listing 1

Truffle

Will man selbst eine Sprache entwickeln, ist das Framework Truffle sogar noch einfacher. In diesem Fall hinterlegt man einfach für jeden Knoten der AST eine in Java geschriebene Implementation. Die GraalVM kompiliert daraus nativen Maschinencode. Durch Optimierungen wie die Partial Evaluation ist die Performance sehr gut. Laut Chris Seaton war beispielsweise TruffleRuby durch diesen Ansatz 2017 die schnellste Implementation von Ruby, nach seinen Worten „oft zehn Mal schneller als andere Implementationen" *[9]*.

Auf GitHub ist ein Quelltext zu finden, der eindrucksvoll zeigt, wie einfach es ist, mit Truffle einen Compiler zu schreiben *[10]*. Auf das Wesentliche reduziert, sieht die Implementation der Addition zweier Zahlen wie in *Listing 1* aus. Auf den ersten Blick ist das nicht beeindruckend. Klar, so sieht eine Addition in Java aus, ja und? Der Clou an der Sache ist, dass es eben gerade keine Addition in Java ist. Es zeigt, wie man beim Entwerfen einer Programmiersprache die Addition implementiert. Es könnte PASCAL sein oder Ruby oder LUA oder eine Programmiersprache, die man entwirft, um bestimmte Ideen auszuprobieren.

Man braucht also nicht mehr Assembler zu lernen, um eine Sprache zu entwickeln oder bei der Weiterentwicklung einer Programmiersprache zu unterstützen. Es reicht, eine weitverbreitete Hochsprache – Java – zu kennen.

Warum ist der Graal-Compiler besser?

Nach Einschätzung der Autoren ist das der große Vorteil des Graal-Compilers. Er macht es Entwicklern leicht, sich bei der Sprachentwicklung einzubringen. Damit ist der Pool an Leuten, die gute Ideen einbringen können, größer als beim HotSpot-Projekt.

Das gilt auf allen Ebenen. Der Java-Compiler von GraalVM erzeugt durchaus nativen Assemblercode, und das kann man auch für andere Programmiersprache tun. Der Entwickler hat die Wahl. Der größte Teil des Compilers wird in allen Fällen mit Java geschrieben, mit allen Vorteilen, die das mit sich bringt: einem leistungsfähigen Debugger, ausgereiften IDEs und vielem mehr.

Als die Autoren sich im Frühjahr 2018 mit der GraalVM zu beschäftigen begannen *[12]*, war es noch so, dass die Performance der GraalVM tendenziell schlechter als die des HotSpot-Compilers war. Inzwischen scheint die GraalVM aufgeholt und zum Überholen angesetzt zu haben.

Herausforderung Cloud

Zurück zum Anfang. Bisher haben die Autoren beschrieben, warum sie die GraalVM faszinierend finden, und eine Menge über Interpreter, Compiler und Sprachimplementierung berichtet. Unter anderem auch, dass Java-Programme nicht sofort, sondern erst nach einiger Zeit sehr schnell sind.

Genau das ist in Cloud-Umgebungen allerdings unerwünscht. Die Idee der Cloud ist, nur dann für einen Service zu bezahlen, wenn dieser auch benutzt wird. Die übrige Zeit kann der Service vom Cloud-Provider abgeschaltet werden. Der Speicherplatz und die CPU können sinnvoller verwendet werden, als auf einen Service-Aufruf zu warten, der vielleicht nie kommt. Besonders konsequent ist dies bei Amazon Lambda umgesetzt. Im Prinzip besteht der Service hier nur aus einer – potenziell kleinen – Funktion. Wenn der Service in Java implementiert wird, passiert jedoch etwas ganz anderes. Jedes Mal, wenn die Funktion aufgerufen wird, wird eine JVM gestartet. Das bedeutet, dass rund 2.000 Klassen geladen werden. Die meisten dieser Klassen benötigt der Entwickler gar nicht. Sie werden aber zum Starten der JVM benötigt. Erst nach ein bis zwei Sekunden ist die JVM verfügbar und kann mit dem Ausführen des Service beginnen.

Mit einigen Tricks kann man erreichen, dass diese JVM nicht wieder abgeschaltet wird. Beim nächsten Mal entfällt dadurch die Aufwärmzeit. Allerdings widerspricht das der Idee der Cloud. Anstatt Ressourcen möglichst effizient zu nutzen, wird mit diesem Ansatz das Recycling von CPU und Speicher verhindert.

Ahead-of-Time-Compiler

Die GraalVM bringt einen Ahead-of-Time-Compiler mit (die „SubstrateVM"). Der Entwickler kann damit wie in den guten, alten C++-Zeiten ein Programm in nativen Maschinencode übersetzen. Unter Windows ist das dann ein .EXE-File. Diese Datei läuft auf jedem Rechner, auch wenn dort kein Java installiert ist. Lediglich die Plattformunabhängigkeit geht verloren. Wenn man in die Cloud gehen möchte, muss man darauf achten, für das richtige Betriebssystem zu kompilieren – in der Regel also Linux.

Der springende Punkt ist, dass das Executable sehr schnell startet. Bei den Messungen der Autoren lag die Startzeit im Millisekundenbereich, eigentlich sogar noch unterhalb ihrer Messgenauigkeit.

Nachteile und Einschränkungen

Es gibt ein paar Einschränkungen. Die offensichtlichste ist, dass das Programm nicht mehr zur Laufzeit optimiert werden kann. Das Executable wird immer etwas langsamer sein als das gleiche Programm mit dem JIT-Compiler – zumindest, wenn die Zeit reicht, um den HotSpot-Compiler warmlaufen zu lassen.

Der andere Nachteil ist, dass das Reflection-API nicht funktionieren kann. Die Idee des Reflection-API ist es, zur Laufzeit herauszufinden, welche Klassen im Speicher sind. Der springende Punkt ist „zur Laufzeit". Der Ahead-of-Time-Compiler läuft ins Leere. Das würde in der Java-Community auf wenig Gegenliebe stoßen, und das weiß auch das GraalVM-Team. Im Prinzip ist praktisch jedes moderne Java-Framework davon betroffen: Hibernate, Spring und viele weitere. Darauf möchte niemand verzichten.

Die SubstrateVM löst das Problem durch einen Kompromiss. Die Idee dahinter beruht auf der Beobachtung, dass so gut wie niemand Änderungen an einer laufenden Softwareinstallation vornimmt. Wenn ein Programm erst einmal an den Kunden ausgeliefert wurde, bleibt das Programm so. Veränderungen gibt es erst bei der nächsten Lieferung.

Die Flexibilität des Reflection-API ist also überflüssig. Es ist leicht möglich, bereits zur Kompilierzeit zu erkennen, welche Klassen geladen werden. Im Prinzip kann man jeden Reflection-API-Aufruf auflösen und direkt die angefragte Klasse oder Methode liefern. Das funktioniert auch in der Praxis, wenn auch mit einer beeindruckenden Liste von Einschränkungen *[15]*. Wenn

man nach „SubstrateVM" und „Hibernate" oder „Spring Boot" googelt, finden man in erster Linie Artikel, die sagen, dass das nicht funktioniert. Soweit die Autoren das erkennen können, wird intensiv daran gearbeitet.

In der Zwischenzeit sind andere Frameworks populär geworden, die von vornherein für die SubstrateVM entwickelt wurden. Insbesondere ist hier Quarkus zu nennen.

State of the Art

Die aktuellen Versionen von GraalVM werden als produktionsreif bezeichnet. Im Großen und Ganzen deckt sich das mit den Erfahrungen der Autoren. Nichtsdestotrotz empfehlen sie, einen Umstieg vorsichtig zu planen und erst einmal gründlich zu testen. In einigen Bereichen sind ihnen auch noch offene Baustellen aufgefallen. Beispielsweise fehlt die Node.js-Umgebung in der Windows-Version von GraalVM. Außerdem sind Entwickler mit der GraalVM 19.2.0 noch auf Java 8 festgelegt. Für Java 11 gibt es aktuell (Stand: September 2019) noch keine GraalVM.

Interessiert man sich für den Ahead-of-Time-Compiler, sollte man sich mit Frameworks wie zum Beispiel Quarkus vertraut machen. Das bedeutet möglicherweise den Abschied von liebgewonnenen Frameworks wie Spring Boot oder Hibernate. Der Vorteil ist, dass Quarkus speziell für die GraalVM entwickelt wurde.

Fazit

Nach zwanzig Jahren wagt Oracle mit der GraalVM einen radikalen Neustart. Die Community kann gespannt sein, ob die GraalVM irgendwann Teil der Standard-Java-Installation oder ob es auch weiterhin zwei verschiedene Installationen geben wird. Der aktuelle Stand der Entwicklung stimmt die Autoren allerdings optimistisch!

Quellen

[1] *http://www.commercialventvac.com/java_performance.html*
Jeff Silverman (2008)

[2] *https://en.wikipedia.org/wiki/JBuilder*
Wikipedia (2019)

[3] *https://www.beyondjava.net/java-programmers-guide-java-byte-code*
Stephan Rauh (2019)

[4] *https://www.beyondjava.net/java-performance-through-the-ages*
Stephan Rauh (2013)

[5] *https://www.youtube.com/watch?v=MFvDLg-qKuU*
Charles Nutter (2017), „From Java to Assembly: Down the Rabbit Hole"

[6] *https://www.openhub.net/p/jdk7u-hotspot*
OpenHub (2019), Projektkennzahlen jdk7u-hotspot

[7] *https://www.openhub.net/p/jdk9-hotspot*
OpenHub (2019), Projektkennzahlen jdk9-hotspot

[8] *https://www.graalvm.org/docs/reference-manual/languages/js/#graalvm-javascript-compatibility*
GraalVM.org (2019), GraalVM JavaScript Compatibility

[9] *https://chrisseaton.com/truffleruby/jokerconf17/*
Chris Seaton (2017), Understanding How Graal Works - a Java JIT Compiler Written in Java

[10] *https://github.com/oracle/truffleruby/blob/master/src/main/java/org/truffleruby/core/numeric/IntegerNodes.java*
GitHub (2019), Implementation of the integer operations in TruffleRuby

[11] *https://ia601208.us.archive.org/16/items/vmss16/click.pdf*
Cliff Click (2016), Bit of Advice for the VM Writer

[12] *https://www.beyondjava.net/graal-towards-the-holy-grail-of-polyglot-programming*
Stephan Rauh (2018), Graal - Towards the Holy Grail of Polyglot Programming

[13] *https://en.wikipedia.org/wiki/Abstract_syntax_tree*
Wikipedia (2019), Abstract Syntax Tree

[14] Compiler: Prinzipien, Techniken und Werkzeuge; Aho, Sethi und Ullman (2008 – 3. Auflage)

[15] *https://github.com/oracle/graal/blob/master/substratevm/LIMITATIONS.md*
GraalVM-Projekt (2019): Native Image Java Limitations

Karine Vardanyan
OPITZ CONSULTING Deutschland
Karine.Vardanyan@opitz-consulting.com

Parallel zu ihrem Masterstudium an der TU Darmstadt arbeitet Karine Vardanyan seit sieben Monaten als Werkstudentin im Bereich Software Engineering bei der OPITZ CONSULTING Deutschland GmbH. Sowohl an der Universität als auch bei der Arbeit befasst sie sich mit spannenden Themen und Software-Entwicklungstechnologien wie künstliche Intelligenz, Java und vielem mehr.

Stephan Rauh
OPITZ CONSULTING Deutschland
Stephan.Rauh@opitz-consulting.com

Stephan Rauh arbeitet als Solution Architect bei der OPITZ CONSULTING Deutschland GmbH und befasst sich seit Jahren mit Angular, JSF und generell mit UI-Technologien. Er ist durch seinen Blog http://www.beyondjava.net und sein Open-Source-Framework BootsFaces bekannt geworden. Stephans weitere Steckenpferde sind das „Tech-Scouting", Programmiersprachen und Compiler.