Ostfalia - Hochschule für angewandte Wissenschaften
Fakultät Informatik
Studiengang Informatik

Seminar

GraalVM - NativeImage

eingereicht bei Prof. Dr. B. Müller

von Erik Simonsen 70455429

Wolfenbüttel, den 24. Januar 2020

Zusammenfassung

In den letzten Jahren ging der Trend in der Softwareentwicklung von Monolithen, die auf hohen Durchsatz ausgelegt waren, zu Microservices und Functions as a Service(FaaS). Diese Anwendungsmodelle sind auf auf kleine, in sich geschlossene Anwendungen ausgelegt. Statt einen möglichst hohen Durchsatz während der Laufzeit zu erzielen, werden die Services nur dann gestartet, wenn sie angefragt werden und ansonsten heruntergefahren. Dadurch entstehen nur Kosten, wenn ein Service tatsächlich genutzt wird und nicht wenn dieser im Idle-Zustand ist. Die Kosten die bei einer Anfrage an den Service von den Cloudanbietern erhoben werden, hängen hauptsächlich von dessen Startzeit, Laufzeit und Speicherverbrauch ab.

Die Nutzung von Java spielte in diesen Bereichen bislang eher eine kleine Rolle, aufgrund der langsamen Startzeiten und Aufwärmzeiten, sowie des hohen Speicherverbrauchs der JVM. Ein Faktor dafür ist unter anderem die Funktionalität die die JVM zur Laufzeit bietet und es von anderen Plattformen abhebt, beispielsweise *dynamic class loading*. Für Anwendungen, die solche Java-Features nicht nutzen, und zum Deployment der sog. Closed-World Annahme entsprechen, bietet die Native Image-Funktionalität von GraalVM schnelle Startzeiten und geringen Speicherverbrauch, und macht sie dadurch geeignet für genannte Anwendungsmodelle. Dabei wird der Anwendungscode, dazugehörige Bibliotheken, die JVM und spezifische Virtual Machine-Komponenten ahead-of-time in eine nativ ausführbare, in sich geschlossene Datei (*native image* bzw. *executable*) kompiliert. Während des Build-Prozesses werden erhebliche Code-Optimierungen gemacht und der Heap des *native image* wird bereits vorgefüllt. Dadurch wird eine erhebliche Verbesserung der Startzeit und des Speicherverbrauches gegenüber der Java Hotspot VM erzielt.

Ehrenwörtliche Erklärung

Hiermit erkläre ich ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig und ohne unerlaubte fremde Hilfe angefertigt habe, andere als die angegebenen Quellen nicht benutzt und die benutzten Quellen wörtlich oder die inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Wolfenbüttel, den 24. Januar 2020

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Softwareentwicklung hat sich in den letzten Jahren stark verändert, moderne Anwendungen setzen immer öfter auf Microservices und verschiedene Modelle des Cloud-Computings, wie Serverless Computing. Statt große Anwendungsserver zu nutzen, werden dabei unabhängige Services deployed (Function as a Service, FaaS). Sobald die Workload steigt, also die Anzahl der Anfragen an den Service steigen, werden von der jeweiligen Cloudplattform mehrere *Worker* erstellt, welche die Anfragen bearbeiten. Bei der ersten Anfrage die ein neuer *Worker* bearbeitet, muss zuerst die Laufzeitumgebung der jeweiligen Programmiersprache und die Konfiguration des Services initialisiert werden. Da solche Kaltstarts bei virtuellen Machinen häufig sehr langsam sind[1] und Cloudanbieter Kosten abhängig von der Start-, und Laufzeit sowie des Speicherverbrauches eines Services erheben, bietet sich die Nutzung von Java VMs kaum an.

1.2 GraalVM

GraalVM ist ein Softwareprodukt von Oracle. Es wird als Ökosystem bezeichnet und die zentrale Komponente ist der *GraalVM Compiler*, ein in Java geschriebener JIT(Just-in-time)-Compiler für Java, der laut dem Hersteller durch neue Codeanalyse- und Optimierungsmethoden erhebliche Performancevorteile bietet. Darüber hinaus verfügt GraalVM über die Möglichkeit der polyglotten Programmierung mithilfe des Truffle-Frameworks und der Ausführung von nativem Code auf der JVM, durch einen LLVM Bitcode Interpreter. Zudem bietet GraalVM die *Native Image* Funktionalität, die im weiteren Verlauf dieser Arbeit thematisiert wird. *Native Image* ermöglicht das ahead-of-time kompilieren von Java-Anwendungen in nativ ausführbare Dateien (engl. executable), in dieser Arbeit als *native image* bezeichnet. Die Grundlage von GraalVM bietet das OpenJDK[Graa].

Darin enthalten ist, neben Komponenten & Werkzeugen für die Entwicklung, auch die Java-Laufzeitumgebung (Java Runtime Environment, JRE), welche die HotSpot JVM beinhaltet. Die Hotspot JVM enthält zwei JIT-Compiler: C1, einen schnellen und nur leicht optimierenden Compiler, ursprünglich gedacht für Desktopanwendungen, und C2, einen aggressiv optimierenden Compiler für Serveranwendungen, bei denen es auf höchsten Durchsatz ankommt[Ora]. Über das JMVCI (Java Virtual Machine Compiler Interface, [Ope]) wird der GraalVM-Compiler in das OpenJDK integriert und ersetzt den Compiler C2 der HotSpot JVM. Entgegen des Namens ist GraalVM also genau genommen keine eigene virtuelle Maschine, sondern nutzt die HotSpot JVM mit einem anderen JIT-Compiler und bietet einiges an Zusatzfunktionalität.

[1] Besonders bei Java VMs durch Ausführen von Class Loading, Profiling, dynamischer Kompilierung etc.

1.3 NativeImage

Die *NativeImage*-Funktionalität wird bei GraalVM als optionale Komponente angeboten, und muss explizit installiert werden. Sie erlaubt die AOT-Kompilierung von Java-Anwendungen in nativ ausführbare Dateien. Das *native image* läuft nicht auf der JVM und bezieht Funktionalitäten, wie Speicherverwaltung, Thread-Scheduling und Garbage Collection von einer, auf das notwendigste reduzierten, in Java geschriebenen virtuellen Machine , genannt *SubstrateVM*[Grab]. Die *SubstrateVM* wird, zusätzlich zum Anwendungscode, Bibliotheken und dem JDK in das *native image* hineinkompiliert. Weil das executable in sich abgeschlossen ist muss die Closed-World Annahme erfüllt werden, das heißt jeglicher Bytecode muss zum Zeitpunkt der Kompilierung bzw. zur Build-Time bekannt und vorhanden sein, und zur Laufzeit dürfen keine neuen Klassen geladen oder Bytecode generiert werden. Ansonsten wäre ein Großteil der ausgeführten Optimierungen während der Build-Time nicht möglich.

Im Weiteren Verlauf der Arbeit wird der gesamte Build-Prozess erläutert, Performanceparameter zwischen einer Anwendung als *native image* und auf der Hotspot JVM verglichen, sowie die Limitierungen von GraalVM Native Image dargelegt.

2 Systemüberblick

In diesem Abschnitt wird die Systemstruktur gezeigt, sowie alle, zur Build- und Runtime relevanten Komponenten von GraalVM NativeImage erläutert.

Der Anwendungscode (Java Bytecode), die benötigten Bibliotheken, das JDK und VM-Komponenten(SubstrateVM)[2] dienen als Eingabe des sog. *image builders*. Dieser verarbeitet die Eingabe und produziert daraus eine, auf ein spezifisches Betriebssystem und Prozessorarchitektur angepasste, nativ ausführbare Datei, ein *native image*. Abbildung 1 zeigt die Komponenten und den Ablauf des Build-Vorgangs eines *native image*.

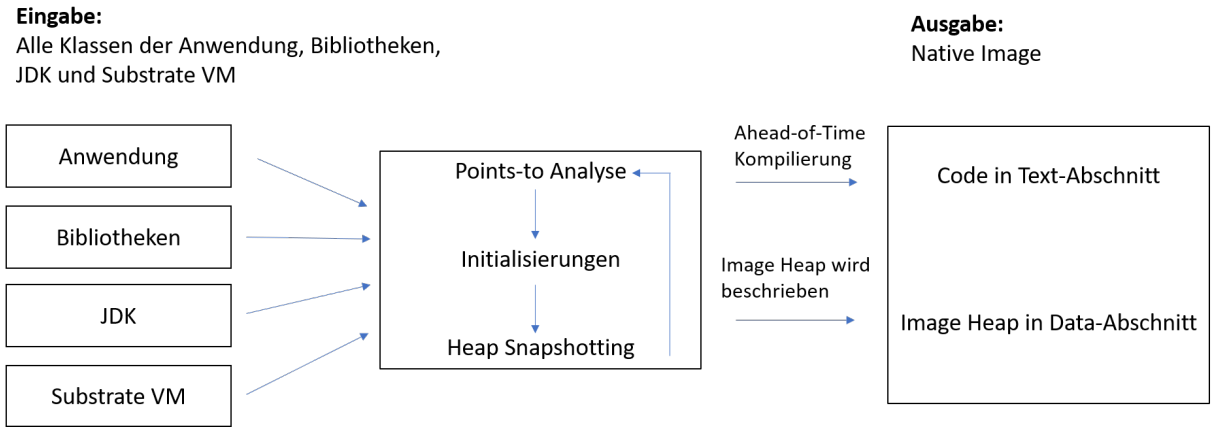

Abbildung 1: NativeImage Build-Vorgang

Auf der Eingabe werden die Points-To Analyse, das Ausführen von Initialisierungscode und das Heapsnapshotting iterativ durchgeführt bis ein definierter Punkt erreicht ist. Zudem werden die registrierten Callbacks der Anwendung, die über die Feature-API von GraalVM nutzbar sind, in diesem Zuge auch ausgeführt. Das Ergebnis dieses Prozesses ist eine Liste von, zur Laufzeit, erreichbaren Klassen, Methoden und Feldern, sowie ein Objekt-Graph mit erreichbaren Objekten. Im Letzten Schritt des Build-Vorgangs werden die erreichbaren Methoden zu Maschinencode kompiliert und der Objekt-Graph wird als *image heap* ausgeschrieben, und in die Form des zur Laufzeit tatsächlich vorhandenen Heaps transformiert. Danach wird der Maschinencode in die Text-Section, und der image heap in die Data-Section [Com95, Fig. 1-13] des native image geschrieben. Der gesamte Prozess wird *image build time* genannt[Wim+19]. Der *image builder* selber ist eine Java-Anwendung. Sowohl die Pointer-Analyse und das ahead-of-time-Compiling, als auch das Ausführen der Klasseninitializer und Callback-Funktionen, werden also von ein und derselben JVM übernommen.[3]

[2]Umfasst u.A. Speicherverwaltung, Thread-Scheduling und Garbage Collection

[3]Das heißt wiederrum, dass bei der *image build time*, die Objekte, aus denen sich der spätere *image heap* zusammensetzt, normale Objekte im Java Heap des *image builder* sind.

2.1 Points-to Analyse

Die Points-To Analysis (dt. Pointer-Analyse) ist eine Technik zur Berechnung der Menge von Objekten bzw. dessen Speicherbereichen, auf die eine Programm Variable zur Laufzeit zeigen kann[Hin01; SB15]. Da der komplette Quellcode des zu analysierenden Programmes vorliegen muss, ist die Pointer-Analyse eine statische Analysetechnik.

Die Pointer Analyse startet mit allen Eingangspunkten, beispielsweise der main()-Methode. Dann werden iterativ alle erreichbaren Methoden verarbeitet, bis ein Fixpunkt erreicht ist. Der Java-Bytecode wird durch das Frontend des Compilers[4] in eine Zwischenform überführt (engl. intermediate representation = IR), die das Programm repräsentiert und weitere Optimierungen und Transformationen ermöglicht[Sim+15]. Diese Zwischenform wird als nächstes in einen sog. *type-flow graph* konvertiert. Jeder Knoten stellt eine Operation auf einem Objekttypen dar. Die gerichteten Kanten verlaufen von der Definition des Objektes bis zur tatsächlichen Verwendung. Jeder Knoten verwaltet eine Liste mit Typen (engl. *type state*), die diesen Knoten erreichen können (siehe Abbildung 2). Die Listen werden durch die Kanten des Graphen propagiert. Sobald einer Liste Typen hinzugefügt werden, wird die Änderung an alle weiteren Verwendungen (Kindelemente eines Knotens) propagiert. Typlisten dürfen nur Elemente hinzugefügt werden, und keine Elemente gelöscht werden, damit sichergestellt ist das die Analyse den Fixpunkt erreicht und terminiert. Für jeden Typ, schaut die Pointer Analyse ob dieser instanziiert ist. Falls dem so ist, wird er durch Bytecodes markiert. Für jedes Feld, wird separat ermittelt ob die Felder während der Laufzeit nur gelesen oder auch geschrieben werden[Wim+19], und als *write* oder read markiert. Felder die als *read* markiert sind, werden bereits während der Ahead-of-time Kompilierung berechnet, statt erst zur Laufzeit (sog. *constant-folding*, [GSI14, Kapitel 4.2]).

[4]das Frontend ist eine von 3 Stufen des Kompilierungsvorganges (Front end, Middle end, Back end), auch Analysephase genannt. Dabei wird der Code analyisiert, strukturiert und auf Fehler geprüft.

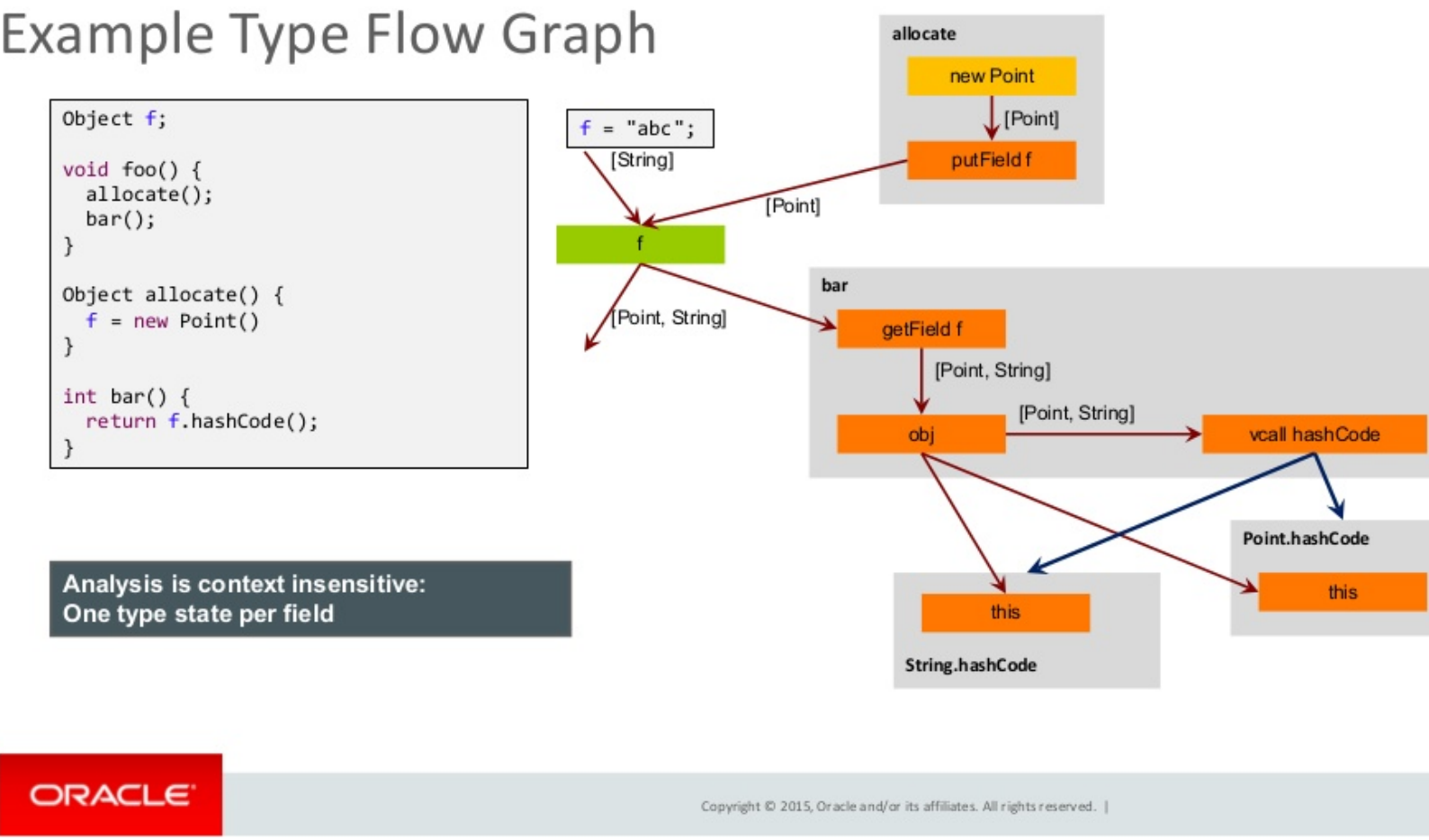

Abbildung 2: Type-flow Graph, [Wim15, Folie 71 & 72]

2.2 Initialisierungscode

Sobald bei der Pointer-Analyse keine Typen mehr zu den Typlisten der Knoten hinzugefügt werden, wird der Initialisierungscode ausgeführt. Als Initialisierungscode werden Klasseninitialisierer und Callbacks durch die Feature-API von GraalVM eingestuft. Im Klasseninitialisierer einer Klasse werden dessen statische Felder initialisiert. Standardmäßig werden alle Klassen erst zur Laufzeit initialisiert, der Entwickler kann jedoch, um die Performance zur Laufzeit zu optimieren, mit dem Befehl *–initialize-at-build-time* eine Liste von Klassen angeben die bereits zur Buildtime initialisiert werden[Wim19, Ab Version 19.0]. Durch die Feature-API kann die Anwendung Callback-Funktionen, mit benutzerdefiniertem Code, registrieren die zur Buildtime, also vor/während/nach der Analysephase ausgeführt werden [Grac]. Der Initialisierungscode kann den Status der Pointer-Analyse abfragen. Es kann also beispielsweise abgerufen werden, ob ein Typ, eine Methode oder ein Feld bereits als „erreichbar“ markiert wurde.

2.3 Heap Snapshotting

Beim Heap Snapshotting wird aus dem *type-flow graph* der Pointer Analyse ein Objekt-Graph generiert.

Die Wurzelzeiger (eng. root pointers) des Graphen sind statische Objekt Felder, die von der Pointer Analyse als *read* markiert sind. Für jedes, als *read* markierte, Objekt wird nun geprüft welche weiteren Objekte durch dessen Felder erreichbar sind. Dafür werden die Felder nachverfolgt, indem die Feldwerte durch Reflection gelesen werden. Dies ist möglich, da der *image builder* eine Java Anwendung ist. Falls die Klasse des Feldwertes von der Pointer-Analyse noch nicht als möglicher Typ für das Feld ermittelt wurde, wird die Klasse als weiterer Feldtyp des Feldes im *type-flow graph* registriert (siehe Abbildung 2).

Nach dem Heap Snapshotting wird die Pointer-Analyse wieder ausgeführt, um den neuen Typ des Feldes allen transitiven Verwendungen des Feldes zu propagieren. Dies geschieht durch den *type-flow graph*. Da im *type-flow graph* neue Typen hinzugekommen sind, werden auch bei der ernuten Ausführung der Klasseninitialisierer neue Initialisierer ausgeführt. Anschließend wird wieder das Heap Snapshotting ausgeführt. Die iterative Ausführung dieser Phasen wird beendet, sobald die Pointer-Analyse in zwei aufeinanderfolgenden Iterationen keine Änderungen findet, wenn also sowohl die Klasseninitialisierer, als auch das Heap-Snapshotting keinen neuen Code erreichbar gemacht haben. Die Objekte des Graphen werden serializiert und in die Data-Sektion des *native image* geschrieben, und bilden somit den *image heap*[Wim+19].

2.4 Ahead-of-Time Compilation

Alle Methoden die von der Points-To Analyse als *erreichbar* markiert sind, werden vom GraalVM Kompiler *Ahead-of-Time(AOT)* kompiliert und in der Text-Section des *native image* platziert. Der Compiler führt alle standardmäßigen Optimierungen, wie u.A Inline-Ersetzung, Loop-Unrolling und Constant-Folding, aus. Zudem nutzt der Compiler die Resultate der Points-To analyse um die Code-Qualität zu verbessern. Dabei werden unter Anderem alle Felder, die nicht als *write* markiert sind direkt als Konstante ausgewertet und Null-Prüfungen werden entfernt, falls der Typ als *niemals null* markiert wurde. Code der nur zur *build time* aber nicht zur *compile time* ausgeführt wird, wie Klasseninitialisierer, wird nicht kompiliert.

2.5 Isolate

Isolates stellen mehrere voneinander unabhängige VM-Instanzen im gleichen Prozess bereit. Statt eines großen Java-Heaps, kann der Heap dadurch in mehrere kleine Teile aufgeteilt werden. Beim Erstellen eines Isolates wird ein neuer separater Heap angelegt, welcher den *image heap* als Startpunkt referenziert. Dadurch sind alle, während des *image building* ausgeführten Initialisierungen sofort in jedem Isolat verfügbar.

2.6 Image Heap

Die Ausführung des *native image* erfolgt mit einem bereits, durch das Heap-Snapshotting zur Build-Time, gefüllten Java Heap, dem *image heap*. Das Starten des executables und das Instanziieren von Isolaten in einem existierenden Prozess muss schnell sein und wenig Speicheraufwand benötigen, damit für kurzfristige Aufgaben sofort mehrere VM-Instanzen gestartet werden können. Um das zu gewährleisten, wird der image heap nicht für jede Instanz neu kopiert werden, sondern stattdessen auf verschiedene Speicheradressen im selben Adressraum abgebildet. Daher sind alle Referenzen relativ zum Start des *image heaps*, sowohl von bereits im image heap vorhandenen Objekten, als auch von Objekten die erst zur Laufzeit allokiert werden. Dadurch kann der *image heap* mehrere Male im Speicher des Adressraumes abgebildet werden, und erlaubt somit schnelles Erstellen von Isolaten, da das Replizieren des *image heaps* kein Kopieren erfordert.
Zudem wird der Speicheraufwand durch das, auf den image heap angewandte, Copy-On-Write Verfahren deutlich reduziert. Dabei erhalten alle Prozesse die auf die gleiche Speicherseite zugreifen, eine flache Kopie davon, also nur Zeiger die auf die tatsächliche Speicheradressen zeigen. Solange die Prozesse die Daten lediglich auslesen, ist es nicht nötig die Daten zu kopieren oder erneut im Hauptspeicher anzulegen. Alle Prozesse greifen auf das “Original“ zu. Erst wenn ein Prozess Daten ändert (write), wird ein neuer Datenblock angelegt(copy) und die entsprechenden Referenzen angepasst [BC02]. Der *image heap* wird vom *garbage collector* als Wurzelzeiger (root pointer) behandelt, und wird daher nicht deallokiert.

3 Performance Benchmarks

GraalVM bietet die Option statt des Graal-Compilers auch den HotSpot-Compiler zu nutzen. Dadurch kann mit verhältnismäßig wenig Aufwand die Startzeit und der Speicherbedarf einer Anwendung, abhängig vom verwendeten Compiler verglichen werden.

Bei einem trivialen Java-Microservice, implementiert in den populären Microservice Frameworks Quarkus, Micronaut und Helidon[5], konnten erhebliche Unterschiede bei der Startzeit und beim Speicherbedarf festgestellt werden, je nachdem welcher Compiler genutzt wurde. Wenn der Microservice mit dem Graal-Compiler zu einem *native image* kompiliert wurde, war die Startzeit um ca. den Faktor 50 schneller (siehe Abb. 4) und der Speicherbedarf um ca. den Faktor 5 geringer (siehe Abb. 5) war, als bei der Java HotSpot VM. Auch bezüglich der entgegengenommenen Anfragen pro Sekunde wurde mithilfe eines, in Quarkus implementierten trivialen REST-Services ein Vergleich durchgeführt. Nach der Warmup-Phase (ca. 70.000 Requests) liegen sowohl das *native-image* als auch die HotSpotVM fast gleichauf mit 31.000 Requests die pro Sekunde verarbeit werden können. Anzumerken hierbei ist allerdings, dass während der Warmup-Phase das *native-image* wesentlich mehr Request pro Sekunde verarbeiten kann und schneller die Spitzenleistung erreicht (siehe Abb. 3).

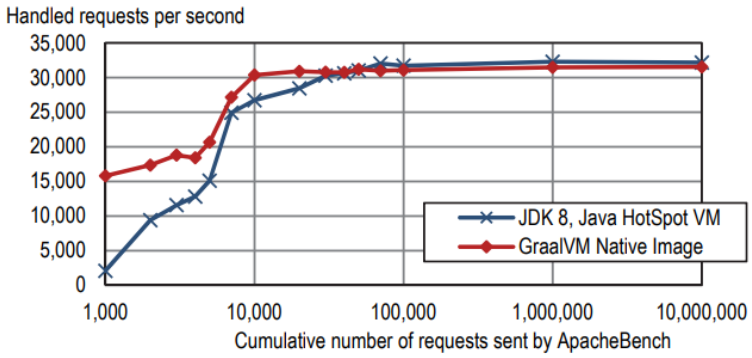

Abbildung 3: Spitzenleistung von einem REST-basierten Service in Quarkus [Wim+19, Fig.12]

[5]Diese Frameworks nutzen GraalVM standardmäßig als Anwendungsplattform

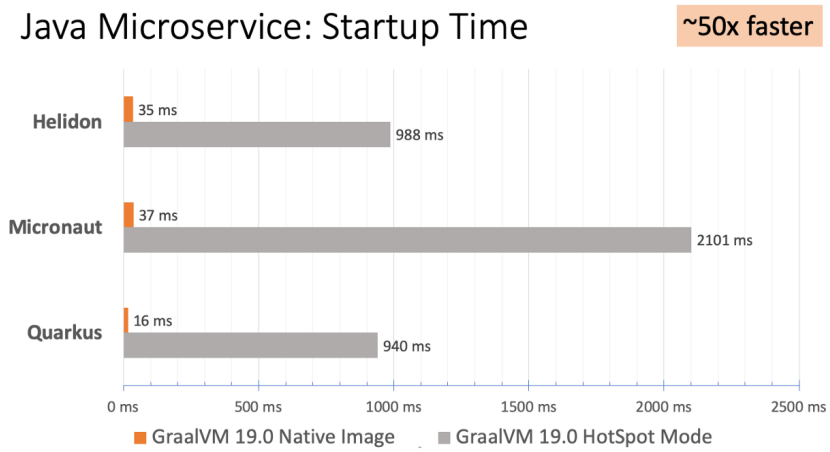

Abbildung 4: Microservices Frameworks Startzeiten mit Native Image [Grae]

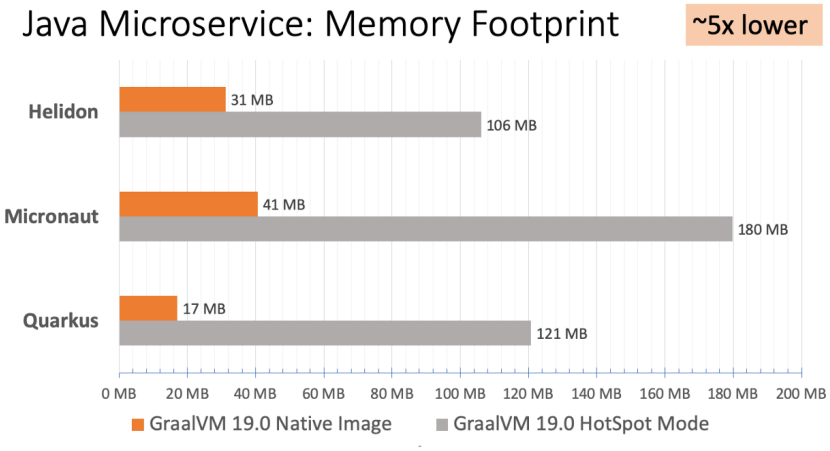

Abbildung 5: Microservices Frameworks Speicherbedarf mit Native Image [Grae]

4 Limitierungen

Die wesentliche Limitierung von *native image* ist die *Closed-World*-Annahme. Das bedeutet, dass der gesamte Anwendungscode zur *image build time* zur Verfügung stehen muss. Java-Features die Nutzen von dynamischer Selbstbeobachtung (engl. dynamic introspection) machen (Reflection & Dynamic Proxy) werden durch Konfigurationsdateien unterstützt. Im Falle von Reflection werden in der Datei die individuellen Klassen, Methoden und Felder angegeben die durch Reflection zugänglich sind. Bei Dynamic Proxy wird die Liste der Interfaces angegeben die dynamische Proxies definieren. Durch die Nutzung von Konfigurationsdateien werden die dynamischen Inhalte bereits zur *image build time*, also ahead-of-time, bekannt und können aufgelöst werden. Um dem Entwickler Arbeit abzunehmen kann die Anwendung im Vorraus auf der Hotspot JVM simuliert werden. Der sog. *Tracing Agent* verbindet sich dabei mit der JVM, fängt alle Aufrufe an Klassen, Methoden, Felder und Ressourcen ab und generiert dadurch die jeweiligen Konfigurationsdateien[Grad].

Features wie Dynamic Class Loading und InvokeDynamic, die zur Laufzeit dynamisch neuen Bytecode generieren oder Methodenaufrufe einfügen und ändern können, werden nicht unterstützt. Auch der Security Manager wird nicht unterstützt, da es kein dynamisches Klassenloading gibt und dementsprechend nur vertrauenswürdiger Code ausgeführt wird [Ora19]. Für eine vollständige Auflistung der Features und deren Unterstützung siehe Abbildung 6.

What	Support Status
Dynamic Class Loading / Unloading	Not supported
Reflection	Supported (Requires Configuration)
Dynamic Proxy	Supported (Requires Configuration)
Java Native Interface (JNI)	Mostly supported
Unsafe Memory Access	Mostly supported
Class Initializers	Supported
InvokeDynamic Bytecode and Method Handles	Not supported
Lambda Expressions	Supported
Synchronized, wait, and notify	Supported
Finalizers	Not supported
Serialization	Not supported
References	Mostly supported
Threads	Supported
Identity Hash Code	Supported
Security Manager	Not supported
JVMTI, JMX, other native VM interfaces	Not supported
JCA Security Services	Supported

Abbildung 6: NativeImage Limitierungen [Ora19]

5 Fazit

Ziel dieser Arbeit war es einen Überblick über die NativeImage-Funktionalität von GraalVM zu geben, und deren Systemkomponenten und Funktionsweise zu erläutern. Durch *native image* wird ein leichtgewichtige, in sich geschlossene ausführbare Datei erzeugt. Als Eingabe dient der Anwendungscode, dazugehörige Bibliotheken, das JDK und die notwendigen VM-Komponenten (SubstrateVM). Auf dieser Eingabe wird iterativ die Points-To Analyse, das Ausführen von Initialisierungscode und das Heap-Snapshotting ausgeführt. Das Resultat wird anschließend ahead-of-time kompiliert und in den Text-Abschnitt des executables geschrieben. Zudem wird ein Java-Heap, der *image heap*, erstellt, gefüllt und anschließend in den Data-Abschnitt des executables eingefügt.

Durch die Initialisierung der Anwendung zur Build-Time und das Nutzen des *image heaps* wird eine schnelle Startzeit, eine geringe Speichernutzung und eine mit der HotSpot JVM vergleichbare Spitzenleistung ermöglicht. Zur geringeren Speichernutzung trägt auch der Umstand bei, dass es nicht mehr notwendig ist die JVM während der Laufzeit laufen zu lassen, da keine Codeoptimierungen zur Laufzeit durch den JIT-Compiler und Profiler stattfinden. Dies trägt zudem zu einem konstanteren und besser vorhersagbarem Performanceverhalten der Anwendung bei. Beim Vergleich populärer Java-Microservice Frameworks konnte im Durchschnitt eine um den Faktor 5 verbesserte Speichernutzung und um den Faktor 50 verbesserte Startzeit gemessen werden, wenn statt der Hotspot JVM ein *native image* verwendet wurde.
Dadurch eignet sich die GraalVM Native Image für die Nutzung von Java in Microservices und Serverless Cloud Functions, da hier bei jedem Start eines Services oder einer Funktion Kosten entsprechend der Start-, Laufzeit und der Speichernutzung erhoben werden, und diese Faktoren somit, im Gegensatz zu maximalem Durchsatz, die höchste Priorität haben. Sobald allerdings der Heap Dimensionen von mehreren Gbyte - Tbyte annimmt und/oder Klassen bzw. andere Teile des Codes erst zur Laufzeit bekannt werden, und somit die Closed-World Annahme verletzen, bietet sich die Nutzung HotSpot JVM eher an.

Literaturverzeichnis

[BC02] D.P. Bovet und M. Cesati. *Understanding the Linux Kernel*. O'Reilly Series. O'Reilly, 2002. ISBN: 9780596002138. URL: https://books.google.de/books?id=9yIEji1UheIC.

[Com95] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2*. 1995. URL: http://refspecs.linuxfoundation.org/elf/elf.pdf (besucht am 23. 12. 2019).

[Graa] GraalVM. *Distribution Components List*. URL: https://www.graalvm.org/docs/ (besucht am 21. 01. 2020).

[Grab] GraalVM. *GraalVM Native Image*. URL: https://www.graalvm.org/docs/reference-manual/native-image/ (besucht am 18. 01. 2020).

[Grac] GraalVM. *Interface Feature*. URL: https://www.graalvm.org/sdk/javadoc/index.html?org/graalvm/nativeimage/hosted/Feature.html (besucht am 24. 12. 2019).

[Grad] GraalVM. *Tracing Agent*. URL: https://www.graalvm.org/docs/reference-manual/native-image/#tracing-agent (besucht am 23. 01. 2020).

[Grae] GraalVM. *Why GraalVM?* URL: https://www.graalvm.org/docs/why-graal/#for-microservices-frameworks (besucht am 22. 01. 2020).

[GSI14] O. Grumberg, H. Seidl und M. Irlbeck. *Software Systems Safety*. NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, 2014. Kap. 4.2. Constant Folding and Propagation, S. 124. ISBN: 9781614993858. URL: https://books.google.de/books?id=KuqbCAAAQBAJ.

[Hin01] Michael Hind. „Pointer Analysis: Haven't We Solved This Problem Yet?" In: *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. PASTE '01. Snowbird, Utah, USA: Association for Computing Machinery, 2001, 54–61. ISBN: 1581134134. DOI: 10.1145/379605.379665. URL: https://doi.org/10.1145/379605.379665.

[Ope] OpenJDK. *JEP 243: Java-Level JVM Compiler Interface*. URL: https://openjdk.java.net/jeps/243 (besucht am 21. 01. 2020).

[Ora] Oracle. *The Java HotSpot Performance Engine Architecture*. URL: https://www.oracle.com/technetwork/java/whitepaper-135217.html#3 (besucht am 23. 01. 2020).

[Ora19] Oracle. *Native Image Java Limitiations*. 2019. URL: https://github.com/oracle/graal/blob/master/substratevm/LIMITATIONS.md (besucht am 20. 01. 2020).

[SB15] Yannis Smaragdakis und George Balatsouras. „Pointer Analysis". In: *Found. Trends Program. Lang.* 2.1 (Apr. 2015), S. 1–69. ISSN: 2325-1107. DOI: 10.1561/2500000014. URL: https://www.researchgate.net/publication/276082849_Pointer_Analysis.

[Sim+15] Christian Simon Doug and u. a. „Snippets: Taking the High Road to a Low Level“. In: *ACM Transactions on Architecture and Code Optimization* 12 (Juni 2015), 20:1–20:25. DOI: 10.1145/2764907.

[Wim15] Christian Wimmer. *Graal Tutorial at CGO 2015 by Christian Wimmer*. Hierbei handelt es sich um eine Präsentation von Christian Wimmer (Project Lead - Native Image of GraalVM) auf der CGO 2015. Die Präsentation konnte vom Author dieser Hausarbeit nur auf slideshare.net gefunden werden. Oracle. 2015. URL: https://www.slideshare.net/ThomasWuerthinger/2015-cgo-graal (besucht am 25. 12. 2019).

[Wim19] Christian Wimmer. *Updates on Class Initialization in GraalVM Native Image Generation*. Sep. 2019. URL: https://medium.com/graalvm/updates-on-class-initialization-in-graalvm-native-image-generation-c61faca461f7 (besucht am 26. 12. 2019).

[Wim+19] Christian Wimmer u. a. „Initialize Once, Start Fast: Application Initialization at Build Time“. In: *Proc. ACM Program. Lang.* 3.OOPSLA (Okt. 2019), 184:1–184:29. ISSN: 2475-1421. DOI: 10.1145/3360610. URL: http://doi.acm.org/10.1145/3360610.