(Kopfzeile)

Ein Neues Klassenzimmer

JEP 387 : Elastic Metaspace

Die JVM braucht Speicher zum Leben und davon leider manchmal viel. Einer der größten off-heap Verbraucher kann der Metaspace sein. JDK 16 bringt mit JEP 387 "Elastic Metaspace" eine komplette Neuimplementierung dieses Subsystems, die es schlanker und sparsamer macht.

von Thomas Stüfe

JDK 16 ist neu auf der Bildfläche erschienen und bringt eine ganze Reihe interessanter Neuerungen. Es gibt neue Sprachfeatures wie die Vector API und die frisch aus dem Inkubator geschlüpften Records, neue Plattform-Ports für Windows AArch64 und Alpine Linux, und Verbesserungen an der JVM selbst. Ein Vertreter der letzteren Gattung ist *JEP 387: Elastic Metaspace* [1].

Interessant an diesem JEP ist unter anderem, dass er in die sehr kleine Gruppe der JEPs gehört, die nicht aus Oracles Entwicklungsabteilung stammen. Vielmehr wurde dieser JEP von SAP entwickelt und gesponsort. Er zählt vom Codeumfang her zu einem der umfangreichsten Patches, die von außen kamen.

Aber was verbirgt sich hinter diesem JEP?

Den Metaspace zähmen

Die JVM kann sehr speicherhungrig sein. Der höchste Verbrauch liegt meist beim Java Heap, daher fokussieren sich viele JVM Entwickler auf die Optimierung des GCs. Projekte wie Red Hats *Shenandoah* und Oracles *ZGC* ziehen viel Aufmerksamkeit und Lob auf sich, und das vollkommen zu Recht.

Der Java Heap ist aber nur ein Teil der Story. Viele Benutzer sind erstaunt, wenn der Footprint ihrer JVM Prozesse deren eingestellte Heap-Grösse deutlich überschreitet. Das ist aber nicht verwunderlich, denn die JVM benötigt neben dem Java Heap auch anderen Speicher. Thread-Stacks, GC-Kontrollstrukturen, CDS-Archiv, Textsegmente, JIT-Arenen und dessen Kompilate sind nur ein paar Beispiele für JVM-interne Daten, die ausserhalb des Java Heaps leben. Die Gesamtheit dieser Speicherbereiche wird oft als "off-heap" oder, etwas unpräziser, als "nativer" Speicher bezeichnet. Die Summe dieses nativen Speichers kann signifikant sein und die Grösse des Java Heaps überschreiten.

Einer der größten Verbraucher nativen Speichers ist häufig der Metaspace. Ist er begrenzt und läuft voll, kommt es zum *OutOfMemoryError*. Leider sind dann die Handlungsmöglichkeiten begrenzt. Endanwender der JVM können lediglich die Limits hochsetzen (*MaxMetaspaceSize oder CompressedClassSpaceSize*). Java-Entwickler können ihre Programme umschreiben, zum Beispiel weniger Klassen laden oder Classloader früher freigeben. Aber damit sind die Möglichkeiten

eigentlich schon erschöpft. Der Metaspace verträgt bisher schlicht bestimmte ungewöhnliche, aber valide Allokationsmuster nicht besonders gut.

Genau dort hakt JEP 387 ein. Der neue Metaspace verbraucht weniger Speicher und gibt ihn bereitwilliger wieder her, er ist also sparsamer und elastischer.

Klassen-Metadaten...

Eine Java-Klasse besteht aus mehr als nur dem *java.lang.Class* Objekt. Lädt die JVM eine Klasse, baut sie im Speicher einen Baum aus Verwaltungsstrukturen auf, der sie und ihre Methoden beschreibt. Die Daten in diesem Baum entsprechen größtenteils dem geparsten und aufbereiteten Laufzeit-Abbild der Klassendatei [2], wie zum Beispiel Klass-Struktur samt vtable und itable, dem Konstantenpool, den Methodenstrukturen, Annotationen, Bytecode, und anderen mehr. Dazu zählen aber auch Daten, die nicht aus der Klassendatei stammen, sondern zur Laufzeit generiert werden, wie zum Beispiel JIT-spezifische Zähler.

… und ihr Lebenszyklus

Eine Klasse wird über einen ClassLoader geladen. Dieser erzeugt das j.l.Class Objekt im Heap und speichert die gelesenen Metadaten. Im Laufe seines Lebens lädt der ClassLoader Klassen und die Größe der angesammelten Metadaten steigt.

Das Entladen der Klassen erfolgt gesammelt dann, wenn ihr ClassLoader stirbt. Das schreibt die Java-Spezifikation so vor:

> ***„A class or interface may be unloaded if and only if its defining class loader may be reclaimed by the garbage collector“[xx]***

Dieser Satz hat interessante Konsequenzen. Jede Klasse hält eine Referenz auf den ClassLoader, der sie geladen hat. Eine Klasse wiederum wird von allen Instanzen dieser Klasse am Leben gehalten. Um also einen ClassLoader abräumen zu können, darf es keine Instanzen seiner Klassen mehr geben, die Klassen selbst müssen unerreichbar sein, und es darf keine Referenz mehr auf den ClassLoader zeigen. Erst dann wird der ClassLoader vom GC erfasst und alle seine Klassen werden entladen. In diesem Moment werden auch die angesammelten Metadaten abgeräumt. Wir haben es also mit einem „Bulk-Free“ Szenario zu tun.

Die Permanent Generation

Metadaten müssen irgenwo leben. Heute leben sie im Metaspace, aber das war nicht immer so. Vor JDK 8 lagen sie im Java Heap, in einer speziellen Generation, der sogenannten Permanent Generation. Dies verursachte Probleme.

Als Teil des Java Heaps war die Permanent Generation durch diesen begrenzt und ihre maximale Größe musste beim Start der JVM festgelegt werden. Klassen und Classloader leben in der Regel länger als normale Objekte, daher hatten last-ditch-GCs geringere Chancen, Speicher in der

Permanent Generation freizuräumen. Eine zu kleine Permanent Generation war in der Regel fatal. Im Zweifel dimensionierte man darum die Permanent Generation unnötig großzügig, und das auf Kosten der anderen Generationen.

Ein weiteres Problem der Permanent Generation war der Aufwand, den man treiben musste, um Metadaten freizugeben. Die GC behandelte sie wie normale Java-Objekte, also als etwas, das jederzeit unvorhersehbar sterben konnte. Aber Metadaten sterben einen definierten, sehr vorhersehbaren Tod: sie werden zusammen mit ihrem ClassLoader abgeräumt. Die Flexibilität eines GCs war an dieser Stelle unnötig und die CPU-Kosten verschwendet [4].

Die Permanent Generation machte auch den JVM-Entwicklern das Leben schwer. Da die Metadaten im Heap lebten, waren sie nicht address-stabil. Sie werden in der JVM aber an vielen Stellen gelesen oder modifiziert, und an allen diesen Stellen musste man Objekt-Referenzen in echte Adressen auflösen. Dies erschwerte auch das Debuggen der JVM mit nativen Debuggern.

Es war also Zeit für etwas Besseres.

Frischer Nordwind

1998 gründeten Stockholmer Studenten die Firma *Appeal Virtual Machines*, um eine alternative Java VM zu entwickeln, die *JRockit VM*. Appeal Virtual Machines wurde 2002 von BEA Systems geschluckt, BEA wiederum 2010 von Oracle. Damit war Oracle Eigentümer der JRockit VM. Einige der JRockit Entwickler sind auch noch heute bei Oracle tätig.

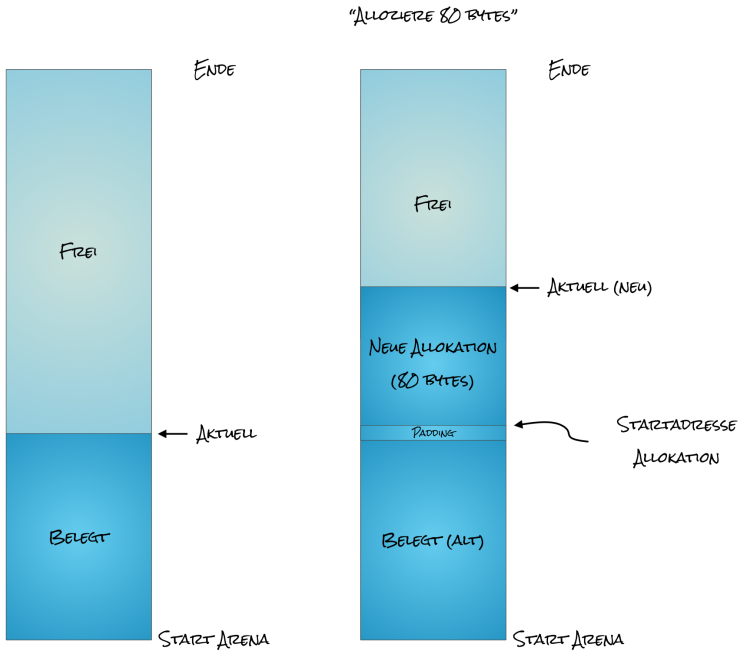
2010 übernahm Oracle Sun Microsystems. Mit diesen Übernahmen hatte Oracle nun zwei hochkarätige JVM Teams unter einem Dach und war Eigentümer zweier JVM-Implementierungen, der JRockit VM und der Sun JVM. Oracle führte die JRockitVM nicht weiter. Stattdessen arbeiteten die Teams zusammen an der ehemaligen Sun VM - dem heutigen OpenJDK. Dadurch finden sich noch heute "JRockit-Gene" an vielen Stellen im OpenJDK wieder. Die JRockitVM kannte keine Permanent Generation, und dies sah man als Vorteil. Es formte sich eine Gruppe ehemaliger Sun- und JRockit-Entwickler, die unter JEP 122 „Remove the Permanent Generation“ [5] eine neue Lösung entwickelten.

Die Geburt des Metaspace

JEP 122 wurde 2014 als Teil von JDK 8 ausgeliefert. Die Permanent Generation war abgeschafft, alle Metadaten in nativen Speicher umgezogen - der Metaspace war geboren.

Metaspace nennt man sowohl den Speicherbereich als auch den darauf aufsetzenden Allokator. Dieser Allokator ist auf das schnelle und speichersparende Verwalten von Metadaten spezialisiert, die im Bulk freigegeben werden. Das unterscheidet ihn von einem General-Purpose-Allokator wie dem C-Heap (malloc, free), der Speicherblöcke verwaltet, die in beliebiger Reihenfolge freigegeben werden können. Diese Fähigkeit bezahlen solche Allokatoren mit CPU-Zyklen und zusätzlichem Speicherbedarf. Da aber alle Metadaten eines ClassLoaders bei dessen Ableben sterben, ist ein Allokator, der ein random-free-Szenario beherrscht, unnötig.

Viel besser eignet sich dafür Arena-basierter Allokator [6]. In seiner einfachsten Form ist dies ein Speicherblock, von dem aufsteigend allokiert wird. Ein Zeiger zeigt auf den Anfang des freien Bereiches. Die nächste angeforderte Allokation wird an diese Stelle gelegt und der Zeiger erhöht. Er wird „hochgeschubst“, weshalb diese Technik auch als "Pointer-bump-allocation" bezeichnet wird. Sie ist primitiv, aber sehr effizient, und verschwendet sehr wenig Speicher.



arena.png
Abb. 1: Arena-Allokator

Man bezahlt die Effizienz einer Arena damit, dass man Blöcke nicht in beliebiger Reihenfolge freigeben kann. Die Arena kann nur entweder komplett freigegeben oder teilweise zurückgerollt werden, letzteres nur in umgekehrter Allokationsreihenfolge.

Arenas sind eine allgegenwärtige Technik, der man an vielen Stellen und unter vielen Namen begegnet. Beispielsweise sind Thread-Stacks an einen Thread gebundene Arenen. Ähnlich funktionieren auch TLABs in der JVM, oder die in Objective-C verbreiteten AutoReleasePools.

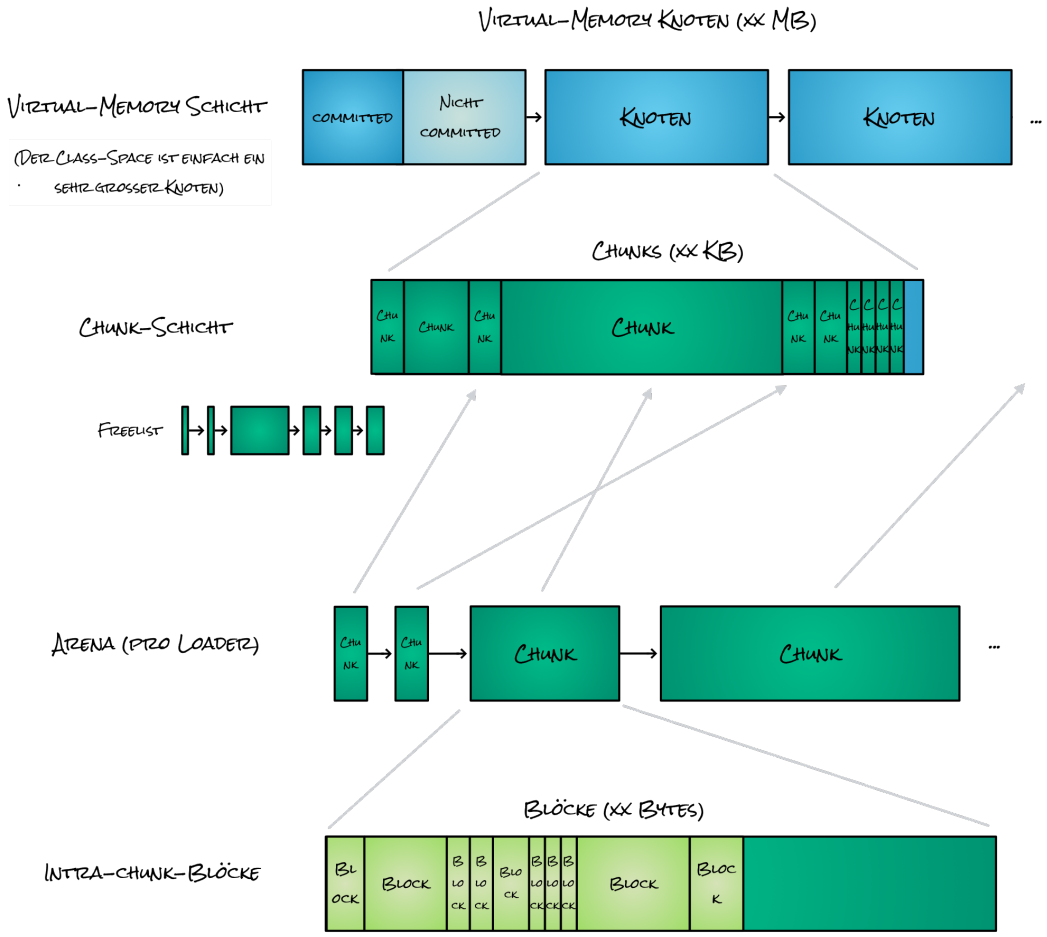
Auch der Metaspace ist ein auf Klassen-Metadaten zugeschnittener Arena-Allokator. Eine Arena gehört in diesem Fall nicht einem Thread, sondern einem ClassLoader. Sie wird angelegt, wenn der Classloader zum ersten Mal eine Klasse lädt, und füllt sich nach und nach mit Metadaten. Wenn der Classloader stirbt, wird die Arena komplett freigegeben.

Die Architektur

Technisch ist eine Metaspace-Arena kein zusammenhängender Block, sondern eine Kette von Blöcken, sog. Chunks. Der ClassLoader allokiert Platz für Metadaten vom ersten Chunk der Liste.

Ist dieser Chunk aufgebraucht, holt er sich einen neuen Chunk aus einer zentralen Chunk-Verwaltung, hängt ihn an den Kopf der Liste und allokiert aus diesem Chunk. Wird der ClassLoader entladen, werden alle Chunks in der Liste zusammen freigegeben.

Der Metaspace hält eine globale Liste großer Speicherbereiche, die vom Betriebssystem mittels mmap() reserviert und bei Bedarf nach und nach committed werden. In diese Speicherbereiche werden neue Chunks gelegt und an die ClassLoader verteilt. Freigegebene Chunks werden in eine globale Freelist eingehängt und können von anderen ClassLoadern wiederbenutzt werden.



metaspace-architecture.png
Abb. 2: Die Architektur des Metaspace

Tatsächlich ist die Metaspace-Architektur komplexer, da die Anforderungen vielseitiger sind. Zum Beispiel kann es vorkommen, daß Metadaten *doch* vor dem Ableben ihres ClassLoaders freigegeben werden müssen. Ein Beispiel dafür ist Klassen-Redefinition, bei der der Bytecode einer Klasse durch neuen Bytecode ersetzt und der Speicher für den alten Bytecode nicht mehr gebraucht wird. Obwohl der Metaspace eigentlich ein Arena-Allokator ist, geht er mit diesen „random-free“ Fällen trotzdem effizient um.

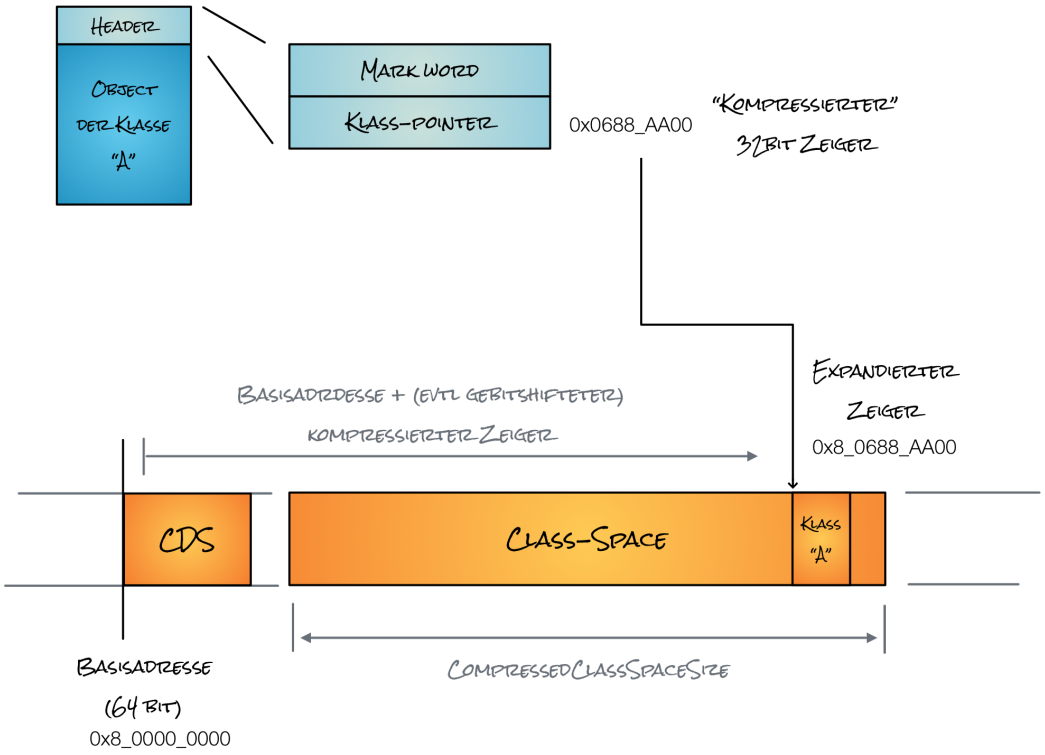
Der Class-Space

Auf 64-bit-Plattformen besteht der Metaspace normalerweise aus zwei Teilen: dem "normalen" Metaspace und dem Class-Space. Letzterer schuldet seine Existenz einer Performance- und Speicheroptimierung der JVM.

Jedes Java-Objekt hat einen Objektheader. Der Header besteht aus dem sog. Markword und einem Zeiger auf eine (tatsächlich mit „K“ geschriebene) *Klass*-Struktur. Diese Struktur liegt im Metaspace und ist die Wurzel eines Strukturbaumes, der die Klasse und ihre Methoden beschreibt.

Auf 64-bit-Plattformen ist der Zeiger auf diese Struktur ein 64bit-Wert. Das ist viel, denn diese acht Byte fließen über den Header in die Größe eines jeden Java-Objektes ein. Um Speicher zu sparen, reduziert man daher die Größe dieses Zeigers: wird dafür gesorgt, daß alle *Klass*-Strukturen innerhalb eines 4 GB großen Addressbereiches liegen, können Zeiger auf diese Strukturen als 32-bit-Indizes in diesen Bereich gespeichert werden. Die echte Adresse errechnet sich durch die Addition des 32-bit-Indexes auf einen bekannten, konstanten Basis-Zeiger – idealerweise eine einzige CPU-Instruktion. Dadurch schrumpt Grösse des Objektheaders um vier Byte, was ein paar Prozent Heap-Speicher spart – eine lohnende Optimierung.

Die Berechnung des CompressedClassPointes ist in der Realität komplexer; hier ist nur wichtig, daß alle *Klass*-Strukturen in einem zusammenhängendem Adressbereich leben müssen. Dieser Bereich, der sog. „Class-Space“, wird beim Start der JVM reserviert.



classspace.png
Abb. 3: Class-Space und die Berechnung des Classpointer

Metaspace und GC

Das Verlegen der Metadaten in nativen Speicher schuf ein neues Problem. Das Abräumen von Metadaten ist an einen GC gebunden, da ja nur ein GC einen ClassLoader entlädt. Solange die Metadaten in der Permanent Generation lebten, war ihre Grösse dem GC bekannt. Wie andere Objekte im Heap auch erzeugten sie Speicherdruck, der beim Überschreiten einer gewissen Grenze eine GC auslöste. Wenn diese GC erfolgreich tote ClassLoader-Objekte abräumte, wurden die dazugehörigen Metadaten gelöscht und der Speicherdruck im Heap verringerte sich.

Da die Metadaten nun außerhalb des Heaps im Metaspace lebten, fehlte dieser Druck – der Regelkreis war unterbrochen. Die Metadaten waren nun für die GC unsichtbar und verhielten sich somit wie andere opaque, an ein Java-Objekt gebundene Resourcen. Es mußte also auf anderem Weg eine GC gestartet werden, um tote ClassLoader-Objekte abzuräumen. Dazu diente ein Schwellwert für die Größe des Metaspace, bei dessen Erreichen ein GC gestartet und der Schwellwert neu

berechnet wird. Im Normalfall, wenn durch den GC keine Klassen entladen werden, korrigiert dieser Mechanismus einfach nur den Schwellwert nach oben.

Der Anfangswert dieses Schwellwertes wird mit dem Schalter *-XX:MetaspaceSize* gesetzt. Dessen Benennung ist übrigens irreführend: nicht die Grösse des Metaspaces wird beeinflußt, sondern der initiale Schwellwert, der den ersten Metaspace-motivierten GC anstößt.

Metaspace-Sorgen

Verglichen mit der Permanent Generation war der Metaspace ein großer Fortschritt. Allerdings fielen bald neue Probleme auf. Die JVM konnte in pathologische Situationen laufen, in denen der Metaspace stark anwuchs. Dafür gab es mehrere Gründe.

Zum einen war die Chunk-Verwaltung des Metaspace sehr einfach gehalten. Chunks, einmal erzeugt, waren in ihrer Größe nicht änderbar. Unter ungünstigen Bedingungen konnte sich die globale Freelist mit freigegebenen Chunks einer Größe füllen, die von später lebenden Loadern nicht gebraucht wurden. Diese Art der Fragmentierung konnte trotz genügend Speicherplatz zum Metaspace-OOM führen. SAP sponsorte 2018 für JDK 11 einen umfangreichen Patch [7], der dieses Problem zumindest abmilderte.

Ein zweites Problem war fehlende Elastizität. Der Metaspace erholte sich von Verbrauchsspitzen nur schlecht und gab Speicher nur zögerlich wieder her. Die JVM machte zwar halbherzige Versuche, Speicher nach einer GC an das Betriebssystem zurückzugeben. Aber schon geringe Fragmentierung hebelte diesen Mechanismus aus, und im Class-Space funktionierte er erst gar nicht.

Des weiteren war der teils beachtliche Overhead pro ClassLoader problematisch, der vor allem ClassLoader traf, die nur wenige Klassen luden. Traten solche ClassLoader in Massen auf, zeigte sich ein oft drastischer Speicherverbrauch. Solche Szenarien betrafen vor allem Applikationen und Frameworks, die feingranulare ClassLoader als eine Form von Modularisierung benutzen. Dynamische, auf Basis von Java implementierte Sprachen wie jRuby litten darunter besonders stark. Aber auch einfache ClassLoader-Lecks oder ungünstige GC-Einstellungen wurden dadurch unnötig teuer.

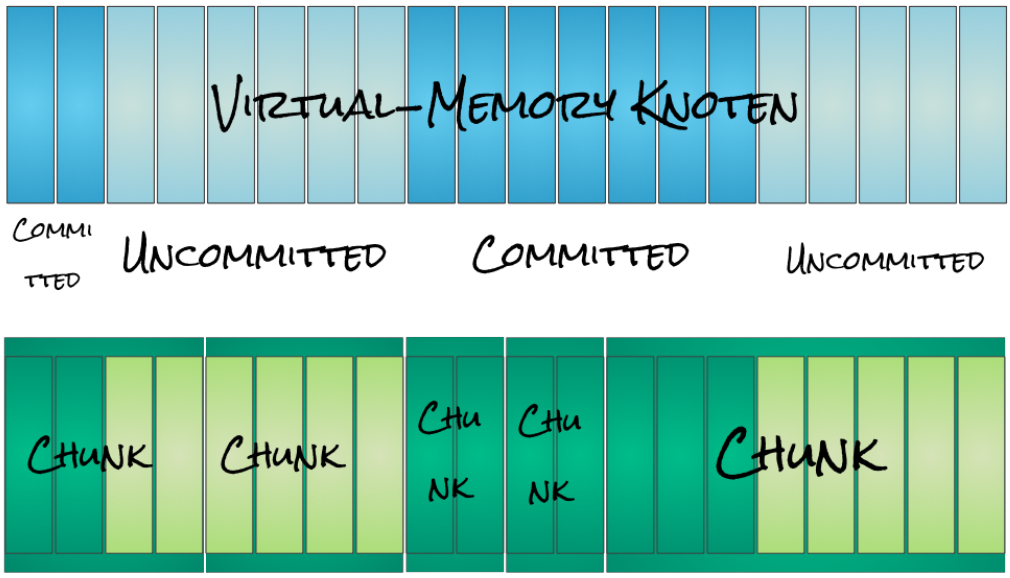
In JDK 16 erscheint nun mit JEP 387 eine Neuimplementierung des Metaspace, der diese und andere Probleme löst. Ihm liegen zwei Ideen zugrunde: *Commit-Granules* und *Buddy-Allocation*.

Commit-Granules

Der alte Metaspace konnte nur sehr grobkörnig committed und kaum uncommitted werden. Einmal erzeugte Chunks waren komplett committed, auch wenn sie unbenutzt in der Freelist (oft vergeblich) darauf warteten, von einem Loader wiederbenutzt zu werden. Dies war die Hauptursache der erwähnten Inelastizität: nach dem Entladen von Klassen verweilte der Speicher in der Metaspace-Freelist, statt an das Betriebssystem zurückzugehen.

Der neue Metaspace ist in der Lage, beliebige Speicherbereiche feinkörnig zu uncommitten – und dies funktioniert auch für den Class-Space. Chunks können nun ganz oder teilweise uncommitted sein. In der Freelist verweilende Chunks werden komplett uncomitted und verbrauchen somit keinen Speicher. An einen ClassLoader gegebende Chunks werden portionsweise und nur bei Bedarf committed – ähnlich einem Thread-Stack. Damit zögert man die Allokation von Speicher, der vielleicht nie benötigt wird, heraus.

All dies wird erreicht, indem der dem Metaspace unterliegende Speicher in eine Serie gleichgroßer Commit Granules aufgeteilt wird – die grundlegende Einheit, in der Speicher uncommitted werden kann. Die Größe dieser Granule bestimmt die Commit-Körnung und ist einstellbar. Der Metaspace hält den Commit-Zustand dieser Granules in einer zentralen Bitmap.



commitgranules.png
Abb. 4: Commit-Granules

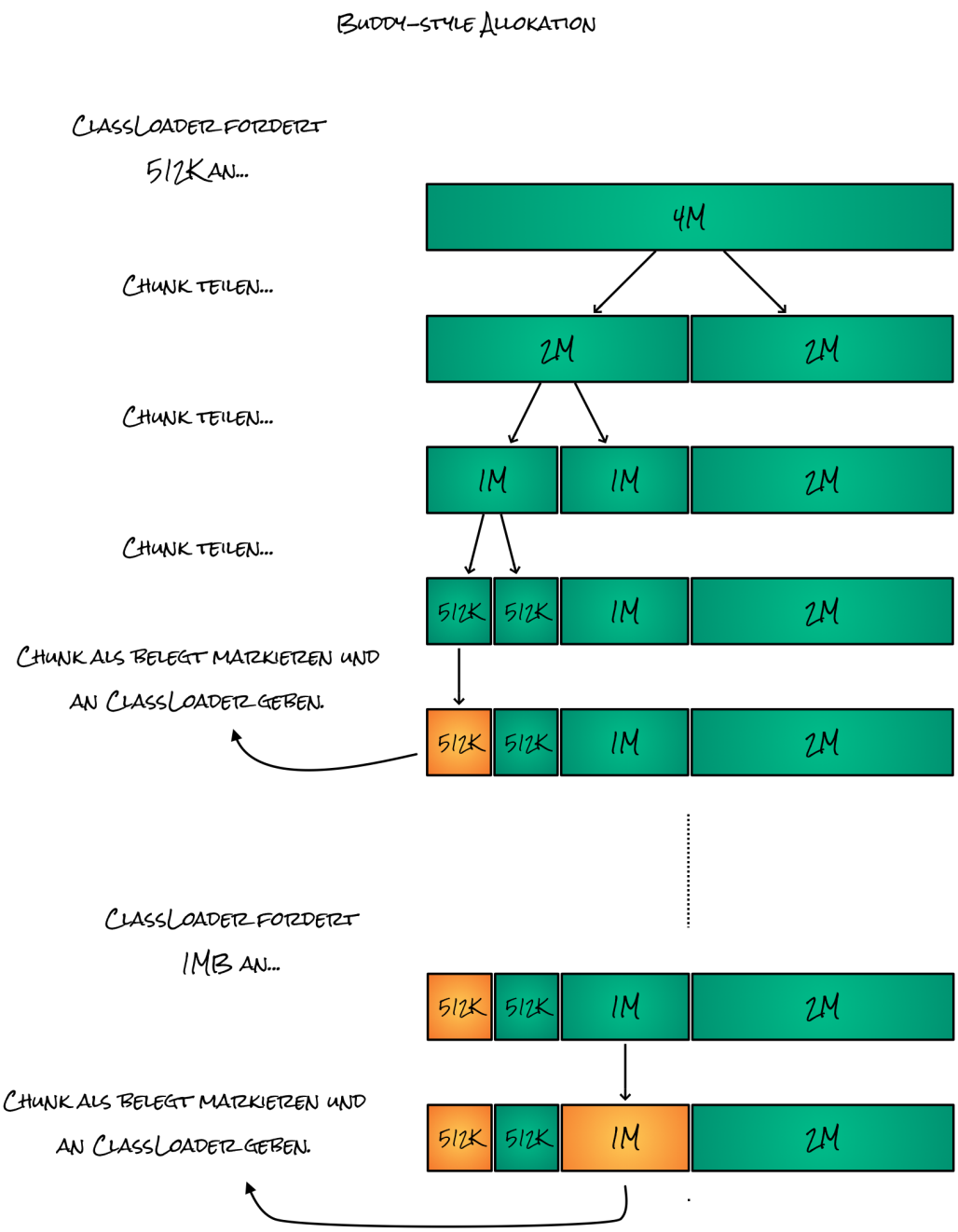
Buddy-Allocation

Der alte Metaspace legte neue Chunks einfach nacheinander ab. Es gab nur drei Chunk-Größen, etwas willkürlich gestaffelt nach 1K, 4K und 64K. Diese Chunk-Geometrie war sehr unflexibel, führte schnell zu Fragmentierung und war ein Grund für den hohen Speicherverbrauch kleiner ClassLoader.

Der neue Metaspace bedient sich zur Verwaltung der Chunks eines effizienteren Algorithmus, der sog. Buddy-Allokation [8]. Dieser Algorithmus zeichnet sich durch hohe Performance und geringe Fragmentierung aus. Benutzt wird er gerne bei der Implementation von C-Heap-Allokatoren, oder in Betriebssystemen – zum Beispiel ist die Verwaltung physischer Speicherseiten im Linux-Kernel als binäre Buddy-Allokation implementiert.

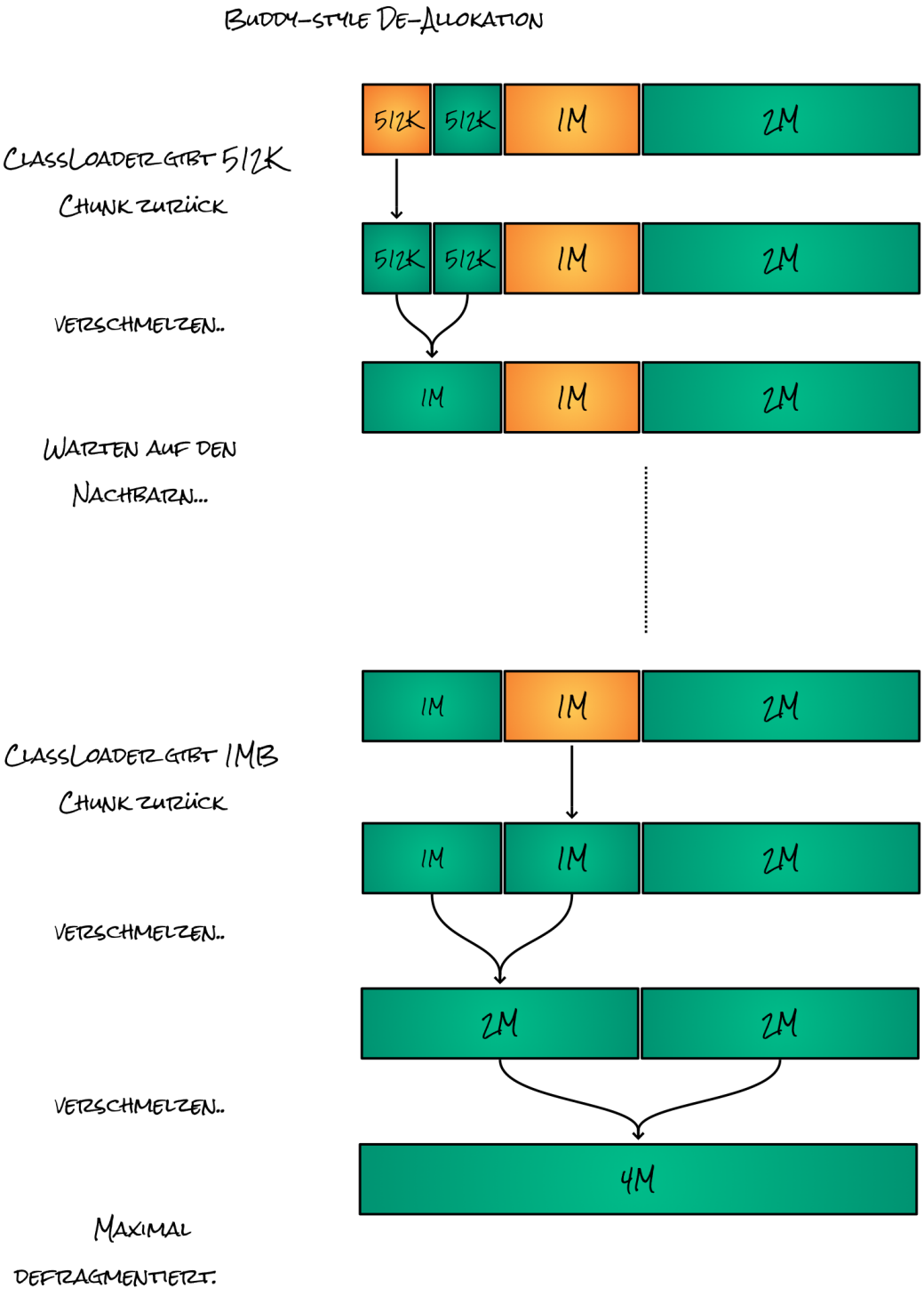
In einem Buddy-Style Allokator werden Blöcke nur in Zweier-Potenz-Größen verwaltet. Dies macht ihn ungeeignet für die Verwendung als „Endanwender-Allokator“, also als ein Allokator, dessen Aufgabe es ist, Speicherblöcke beliebiger Größe zu verwalten. Diese Einschränkung ist aber im Metaspace bedeutungslos, denn die verwalteten Chunks sind nicht das Endergebnis der Allokation, sondern die grobkörnigere Grundlage o.g. Metaspace-Arenen.

Binäre Buddy-Allokation im Metaspace funktioniert nach einem einfachen Prinzip. Es wird ein Vorrat an freien Chunks vorgehalten. Fordert ein ClassLoader einen neuen Chunk an und es gibt keinen Chunk in der passenden Größe, wird der nächstgrößere freie Chunk so lange geteilt, bis ein Chunk der richtigen Größe entsteht. Dieser wird als belegt markiert und dem ClassLoader gegeben; die resultierenden Splitter-Chunks bleiben im Vorrat freier Chunks.



buddy-allocation.png
Abb. 5: Buddy-Allokation

Gibt der ClassLoader beim Ableben seine Chunks frei, findet der umgekehrte Prozess statt: der zurückgegebene Chunk wird als frei markiert und, sofern sein Nachbar-Chunk („Buddy“) auch frei ist, mit diesem verschmolzen. Dies wird wiederholt, bis entweder die maximale Chunkgröße erreicht ist oder ein belegter Nachbar das Verschmelzen verhindert.

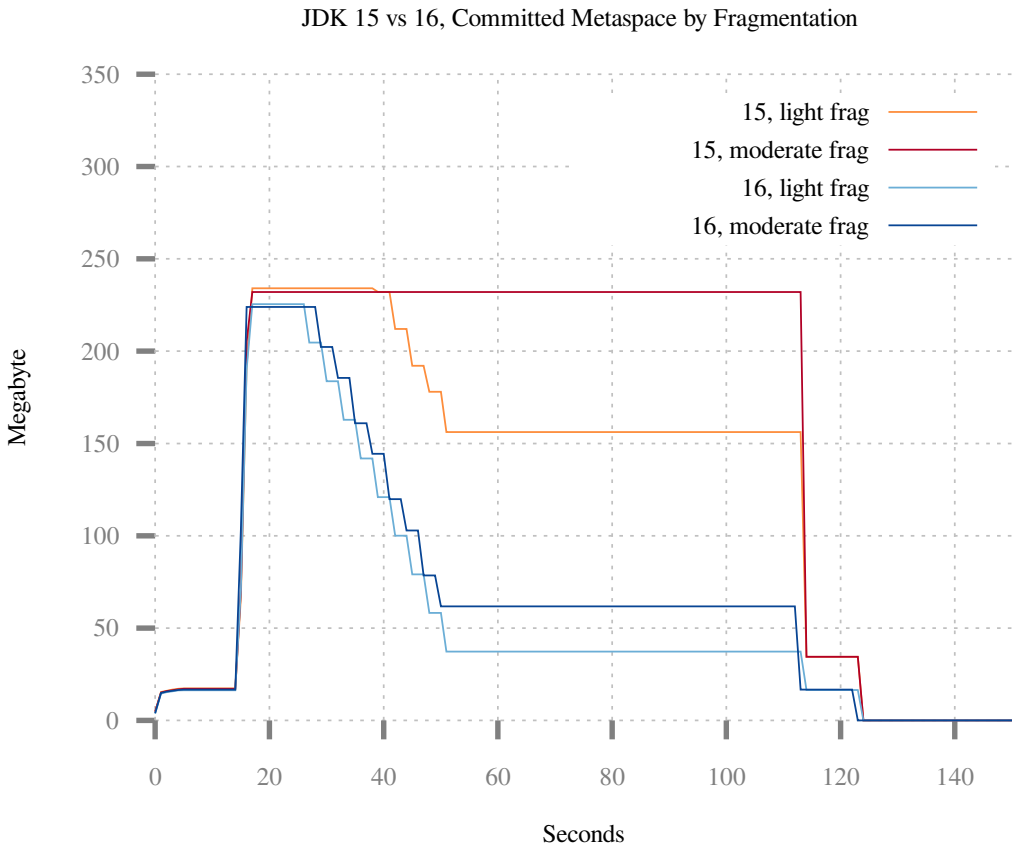


buddy-deallocation.png

Abb. 6: Buddy-Deallokation

Elastizität

Die Fähigkeit, freie Chunks ganz oder in Teilen zu decomitten, gibt dem neuen Metaspace die namensgebende Elastizität. Abb. 7 demonstriert diesen Effekt. Ein Testprogramm [9] erzeugt temporären Metaspace-Druck, indem es eine hohe Anzahl von Klassen lädt und nach kurzer Zeit das fast komplett wieder entlädt. Abhängig von der Fragmentierung des Metaspace gibt JDK 15 wenig bis gar keinen Speicher wieder frei. Der neue Metaspace in JDK 16 wiederum erholt sich sehr gut von der Verbrauchsspitze, reagiert also viel elastischer.

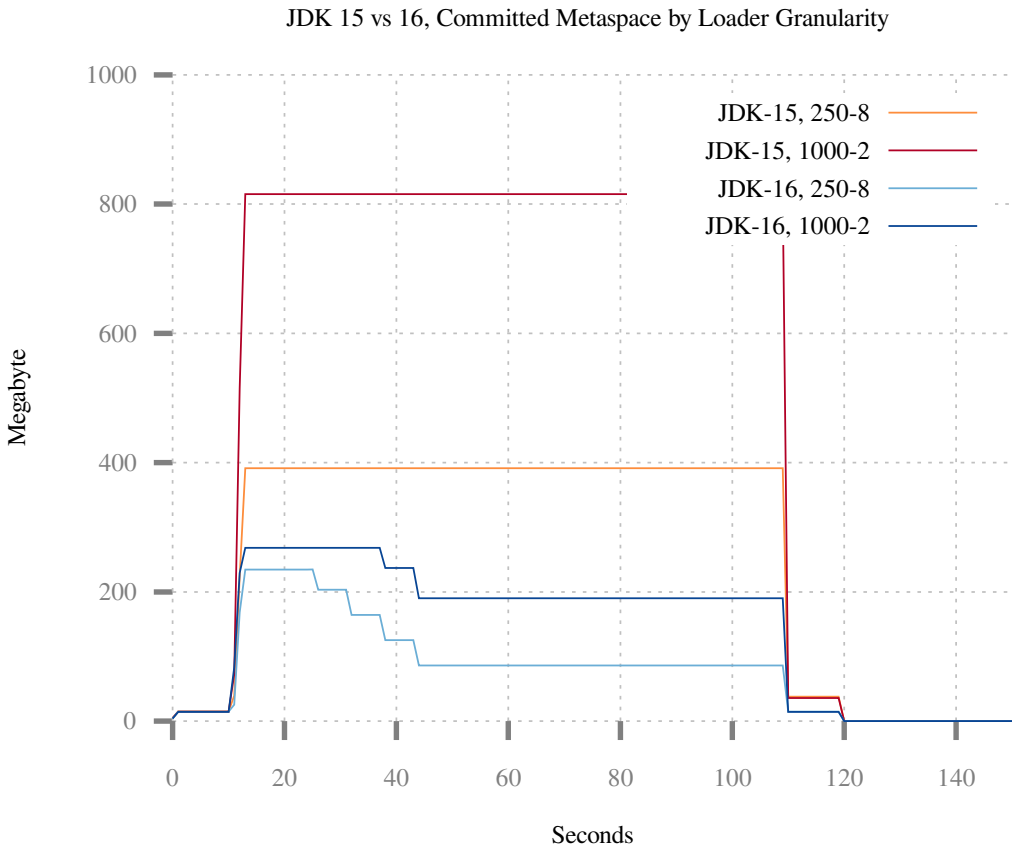


jdk15-vs-16-committed-metaspace-by-fragmentation.svg
Abb. 7: Speicherverbrauch nach einer Lastspitze abh. von Fragmentierung, JDK 15 vs JDK 16

Diese Elastizität ist sinnvoll: das Laden von Klassen geschieht zu selten, als dass es sich lohnen würde, dafür im Metaspace Speicher vorzuhalten. Herrscht realer Speicherdruck, würde er sehr wahrscheinlich an anderer Stelle fehlen. Vorteilhafter ist es, den Speicher an den Kernel zurückzugeben und diesen entscheiden zu lassen, an welche Verbraucher er neu verteilt wird - was im übrigen andere Verbraucher innerhalb der JVM mit einschließen kann.

Reduzierter Overhead

Die verbesserte Chunk-Geometrie, die durch die Buddy-Allokation Einzug hielt, hält Fragmentierung auf der Chunk-Ebene gering und reduziert die Kosten für kleine ClassLoader. Abb. 8 zeigt den positiven Effekt. Um den Overhead per ClassLoader zu demonstrieren, generiert ein Testprogramm [9] eine riesige Menge an ClassLoadern, die jeweils nur eine geringe Anzahl von Klassen laden. Je feiner die ClassLoader-Granularität – also umso mehr ClassLoader umso weniger Klassen laden – desdo deutlicher schlägt der Overhead per Loader in JDK 15 zu Buche. JDK 16 dagegen zeigt deutlich gezähmteren Speicherhunger.



jdk15-vs-16-committed-metaspace-by-loader-granularity.svg

Abb. 8: Speicherverbrauch abh. Von Loader-Granularität, JDK 15 vs JDK 16

Geringerer Anfangsverbrauch

Beim Starten der JVM teilt diese dem Bootclassloader einige MB an Metaspace-Speicher zu. Dies geschieht unter der Annahme, daß der Bootclassloader sehr viele Klassen laden und diesen Speicher brauchen wird. Diese Annahme ist aber nicht immer zutreffend, denn nicht jedes Java-Programm lädt eine große Menge an JDK-Klassen, und zudem liegen viele dieser Klassen mittlerweile im CDS-Archiv und nicht im Metaspace.

Im neuen Metaspace werden Chunks nur bei Bedarf comitted. Der Bootclassloader verbraucht darum nur Speicher, den er auch wirklich braucht. Dies führt zu deutlich geringerem Verbrauch beim Start der VM: ein einfaches „Hello-World“ Programm verbrauchte mit JDK 15 noch 4,75Mb comitteten Speichers, mit JDK 16 sind es nur noch 448Kb.

Höhere Codequalität

Wichtig vor allem für uns JDK-Maintainer ist die höhere Code-Qualität im Metaspace, die mit JEP 387 einhergeht. Trotz höherem Funktionsumfang wurde viel unnötige Komplexität reduziert. Dies vereinfacht die Wartung und macht das Einbauen weiterer Verbesserungen in der Zukunft einfacher und billiger. Code-Hygiene wie diese ist lebensnotwendig für ein langlebiges und großes Projekt wie das OpenJDK.

Open Source funktioniert

Einen Patch einer solchen Größenordnung upstream ins OpenJDK zu bringen, ist selbst unter einfachen Bedingungen eine Herausforderung. Der Metaspace liegt jedoch zusätzlich an zentraler Stelle in der JVM und damit mitten im Territorium von Oracles Runtime-Gruppe. Für den Erfolg des Projekts war deshalb eine gute Kooperation notwendig – und dies über Firmengrenzen hinweg. Diese Kooperation erwies sich als erfreulich produktiv. Freier Informationsfluss und Unterstützung durch die Oracle-Entwickler besonders in der Review-Phase trugen wesentlich zum Erfolg des Projekts bei.

Thomas Stüfe *ist als JVM-Entwickler für die SAP tätig. Er ist seit 2014 im OpenJDK Projekt als Committer und Reviewer aktiv und Autor vieler Verbesserungen in der JVM und im JDK. Er ist Autor des in diesem Artikel vorgestellten JEP 387.*

Links & Literatur

[1] JEP 387: Elastic Metaspace https://openjdk.java.net/jeps/387
[2] Java Virtual Machine Specification https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html
[3] Java Language Specification https://docs.oracle.com/javase/specs/jls/se7/html/jls-12.html#jls-12.7
[4] Thomas Schatzl, Laurent Daynès, H. Mössenböck, „Optimized memory management for class metadata in a JVM“ https://dl.acm.org/doi/abs/10.1145/2093157.2093182
[5] JEP 122: Remove the Permanent Generation https://openjdk.java.net/jeps/122
[6] Region-based memory management https://en.wikipedia.org/wiki/Region-based_memory_management
[7] JDK-8198423 „Improve metaspace chunk allocation“ https://bugs.openjdk.java.net/browse/JDK-8198423
[8] Buddy memory allocation https://en.wikipedia.org/wiki/Buddy_memory_allocation
[9] https://github.com/tstuefe/jep387/blob/master/examples/src/main/java/de/stuefe/repros/metaspace/InterleavedLoaders.java