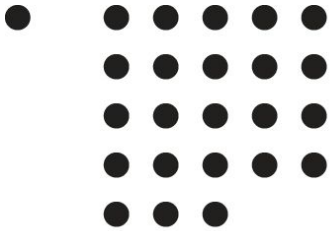


Fachhochschule Köln
Cologne University of Applied Sciences

Fakultät für Informatik
und Ingenieurwissenschaften

Java und Dalvik Bytecode - Ein Vergleich

WPF Compiler und Interpreter
WS 2014/2015

Studiengang Medieninformatik

Autoren

Bryan Riddle,
Matrikelnummer 11100449
bryan.riddle@smail.fh-koeln.de

Sydney Manalo-Schwarz,
Matrikelnummer 11099419
sydney.manalo-schwarz@smail.fh-koeln.de

betreut von

Prof. Dr. Erich Ehses

Gummersbach, Juli 2015

Abstract

Ziel dieser Ausarbeitung ist es, einen Vergleich zwischen den unterschiedlichen Bytecode-Architekturen der Java Virtual Machine (JVM) und der Dalvik Virtual Machine (DVM) anzustellen. Vor dem Hintergrund der Plattformeinschränkungen, die zu ihrer Entwicklung geführt haben, werden die architektonischen Designentscheidungen, welche der DVM und dem entsprechenden Bytecode Format zugrunde liegen, untersucht und Besonderheiten gegenüber Java herausgearbeitet. Die Autoren bedienen sich zur Illustration der Unterschiede neben Beschreibungen der verschiedenen Bytecode-Architekturen auch dem direkten Vergleich zwischen entsprechenden Codebeispielen. Die aus diesem Vergleich erwachsenen Schlüsse werden im Hinblick auf den aktuellen Stand der Technik und die zukünftig zu erwartenden Entwicklungen in einem Fazit zusammengefasst.

Inhaltsverzeichnis

I. Die Java Virtual Machine und Java Bytecode

Die Java Virtual Machine (JVM) ist die Laufzeitumgebung (auch Virtual Machine, kurz VM), die dafür verantwortlich ist, die zu Java Bytecode übersetzten Java-Programme zu interpretieren und auszuführen. In der Regel wird jedes gestartete Java-Programm in einer eigenen VM-Instanz ausgeführt. Der Nutzen der VM ist, die Programausführung von der Systemarchitektur zu abstrahieren und Java-Programme portabel zu machen. Java Programme müssen somit nur in Bytecode übesetzt werden, welcher auf jedem System lauffähig ist, sofern eine JVM dafür existiert. Wenn eine Java .class-Datei durch die JVM geladen wird, besteht sie aus einem Satz von Bytecode Befehlen für jede Methode innerhalb der Klasse. Diese können durch Interpretation oder Just-in-Time-Kompilierung ausgeführt werden. [Java Wiki]

Der Bytecode der JVM ist Stack-basiert, nach dem Last-in, First-out Prinzip (LIFO). Die einzelnen Speicherzellen des Stacks sind 32 Bit groß, womit alle Variablen mindestens 32 Bit belegen. Variablen vom Typ Long oder Double, welche 64 Bit benötigen, nehmen demnach zwei Zellen in Anspruch, welche im Stack übereinander liegen. Da die JVM keine Register zur Verfügung stellt, um Werte direkt zu adressieren, müssen alle Werte und Referenzen zuerst auf den Operandenstack geladen werden, bevor sie in Berechnungen verarbeitet werden können. Zu diesem Zweck gibt es eine Teilmenge von Bytecode-Befehlen, die dazu dienen, lokale Variablen und Konstanten auf den Stack zu laden und vom Stack zu nehmen und zu speichern. Aufrufargumente werden bei der Erzeugung eines Stack-Frames in die Menge der lokalen Variablen des Stack-Frames aufgenommen.

Der Befehlssatz der JVM ist mehrheitlich explizit typisiert. Das heißt, um eine gewisse Operation auszuführen, gibt es Varianten für jeden Datentyp auf welchen diese Operation angewendet werden kann. So wird ermöglicht, dass die JVM eine Typprüfung zur Laufzeit vornehmen kann, allerdings wird die erforderliche Befehlszahl aufgebläht um die zusätzliche Typinformation abbilden zu können. [1]

Die Bytecode Befehle der JVM sind 1 Byte groß. Sie fallen in die Kategorien:

1. Lade- und Speicher Befehle
2. Arithmetische Operationen
3. Befehle zur Typkonvertierung
4. Befehle zur Objekterschaffung und -manipulation
5. Befehle zur Manipulation des Operandenstacks
6. Ablaufkontrollbefehle
7. Befehle zum Methodenaufruf und zur Rückgabe von Werten
8. Befehle zum Werfen von Exceptions
9. Befehle zur Synchronisation (zur Verwaltung von asynchronen Aufrufen)

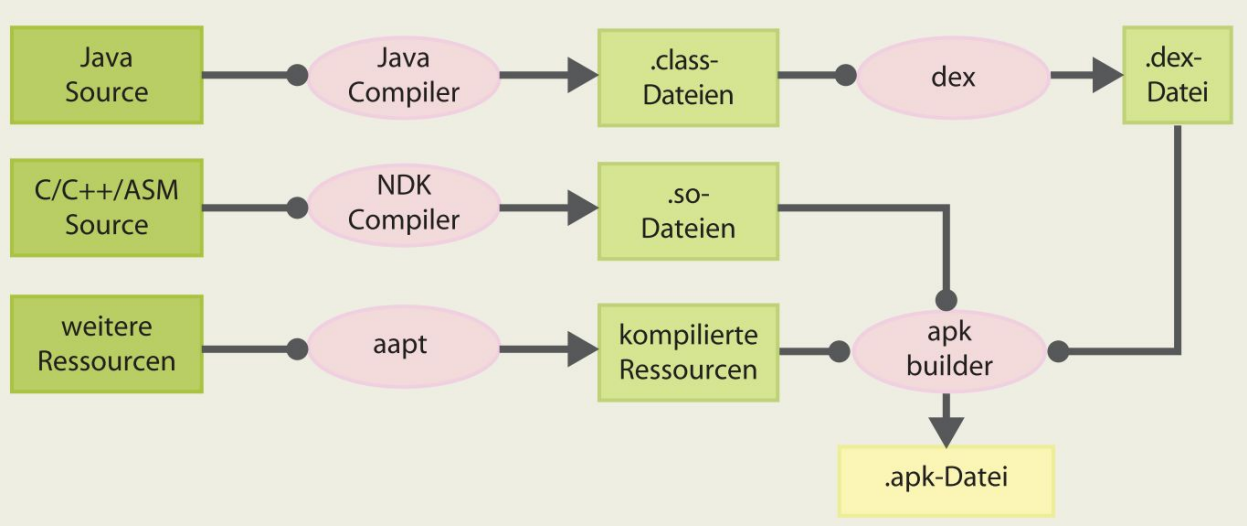
Der gesamte Befehlssatz der JVM kann unter [2] abgerufen werden

II. Android und die Dalvik Virtual Machine (DVM)

Ursprung

Die Dalvik VM wurde unter der Führung von Dan Bornstein bei Android Inc. (gegründet 2003) für das Android Betriebssystem entwickelt. Im Juli 2005 wurde Android Inc. durch Google aufgekauft und die Entwicklung von Android und Dalvik wurde bei Google fortgeführt. Ziel war es, ein quelloffenes mobiles Betriebssystem als Konkurrenten zu den damals prävalenten Betriebssystemen Windows Phone und Symbian zu schaffen.

Die erste öffentliche Vorstellung von Android erfolgte durch die Open Handset Alliance[1] im November 2007. Das erste kommerziell verfügbare Endgerät war das im Oktober 2008 erschienene HTC Dream. Um die breite Akzeptanz von Android zu begünstigen und viele Entwickler an Bord zu holen, setzte man auf Java (2001 im Tiobe-Index[2] mit 26,5% die populärste Programmiersprache [3]) als Programmiersprache zur Entwicklung von Android-Programmen. Diese werden zuerst zu Java Bytecode kompiliert und anschließend mit dem dex-Tool zu Dalvik Bytecode übersetzt. [4]



Der Android Build-Prozess. Java Quellcode wird durch den Java Compiler in .class-Dateien enthaltenen Java Bytecode übersetzt. Das dex-Tool übersetzt diesen in einer .dex-Datei enthaltenen Dalvik Bytecode, welcher mit nativem Code und kompilierten Ressourcen (Bitmaps etc.) vom apk-builder zur .apk-Datei zusammengefügt werden. [5l]

[1] Ein Industriekonsortium bestehend aus Google, Geräteherstellern, Netzbetreibern und Chipherstellern mit dem Ziel, offene Standards für mobile Geräte zu entwickeln.

[2] Ein Ranking von Programmiersprachen, das die Popularität der selbigen widerspiegelt.

Entwicklung

Um die Zielsetzung, Java als Programmiersprache zur Entwicklung von Anwendungen zu unterstützen, musste die Java API von Google reimplementiert werden. Dies wurde nötig, weil Android Inc. und später Google sich nicht mit Oracle über eine Lizensierung der Java-Bibliotheksfunktionen einigen konnten. Um keine Lizenzgebühren zahlen zu müssen, hat man sich bei Google dazu entschieden, die Java API aus Teilen des quelloffenen Apache Harmony Project und durch eigene Entwicklungen vollständig selbst zu implementieren. Der hieraus erwachsene Rechtsstreit dauert noch heute an. [6]

Bei der Entwicklung von Dalvik und Android setzte sich Android Inc. aus heutiger Sicht drakonisch wirkende Einschränkungen als Entwicklungsziele, um die Lauffähigkeit und Performanz von Android auf den mobilen Endgeräten zum damaligen Stand der Technik zu gewährleisten. Die VM sollte auf einem System lauffähig sein, welches folgende Eigenschaften besitzt:

1. Langsame CPU
2. Relativ wenig RAM
3. Betriebssystem ohne swap space
4. Betrieb über einen Akku

Aus diesen Überlegungen wurden folgende minimale Systemanforderungen als Entwicklungsziel extrapoliert:

1. CPU Taktrate von 250-500 MHz
2. Busgeschwindigkeit von 100 MHz
3. Cachegröße von 16 KB bis 32 KB
4. 64 MB Arbeitsspeicher, wovon
    a. 40 MB nach low-level startup verfügbar und
    b. 20 MB nach high-level startup verfügbar sind.

    Somit bleiben für alle Anwendungen nur 20 MB Arbeitsspeicher.

[7]

Aufbau

Um den Speicherplatzeinschränkungen gerecht zu werden, bedient man sich mehrerer Verfahren. Zum einen werden beim Kompilieren durch das dex-Tool alle Java-Klassen zu einer einzigen .dex-Datei zusammengelegt. Dies ermöglicht das Zusammenlegen aller Konstanten aus den Constant-Pools der einzelnen .class-Dateien aus der Java .jar-Datei zu einem einzigen Constant-Pool. Im gemeinsamen Constant-Pool der .dex-Datei werden die Konstanten, anders als in Java, nach Typ sortiert und somit kann auf die explizite Typisierung verzichtet werden. Es gibt also einen Pool für jeden benötigten Typen. Somit kann anhand des Pools implizit der Typ der Konstanten bestimmt werden. Konstanten, die von mehreren Klassen benötigt werden, kommen nicht mehr in jeder .class Datei vor, sondern nur noch einmal im gemeinsamen Pool der .dex-Datei.

Die zweite Strategie zur Einsparung von Speicherplatz ist, dass die Android Bibliotheksfunktionen, anders als in Java, nur einmal für alle Anwendungen im Speicher liegen. Die Android-Bibliotheksfunktionen werden durch den sogenannten “Zygote”-Prozess zur Verfügung gestellt. Damit dieser Speicherbereich sauber (unverändert) bleibt, wird dieser durch “copy-on-write” geschützt. Das bedeutet, dass ein Prozess eine eigene Kopie der Bibliotheksfunktion erhält, sollte er in ihr etwas ändern. So können alle Bibliotheksfunktionen von den Prozessen gemeinsam verwendet werden, ohne dass sie hierdurch mehrmals im Speicher liegen. [7]

Die Dalvik VM wird auf einem Linux-Kernel ausgeführt und Android-Anwendungen werden unter einem eigenen User-Account gestartet. Man bedient sich also der bewährten Linux-Rechteverwaltung, um die Anwendungen gegeneinander zu kapseln und das System zu schützen. [7]

Für die Architektur der VM hat man sich bei Dalvik für einen Register-basierten Aufbau entschieden. Dadurch werden folgende Vorteile gegenüber Java erreicht:

- Vermeidung unnötiger Befehle.
- Vermeidung unnötiger Speicherzugriffe.
- Effizientere Verarbeitung des Befehlsstroms durch höher semantische Dichte der Befehle.

Der Dalvik Bytecode hat eine um 30% geringere Befehlszahl gegenüber Java, benötigt statistisch gesehen 35% weniger Befehle für die gleichen Programme und hat einen um etwa 35% größeren Befehlsstrom, da die Befehle im Vergleich zu den 8 Bit großen Java Bytecode Befehlen mit 16 Bit doppelt so groß sind. [7, Folie 37]

In Summe ist es gelungen, den Platzbedarf von .dex-Dateien leicht unter dem Niveau von komprimierten Java .jar-Dateien zu halten, trotz des Mehrbedarfs an Speicher für die Dalvik Bytecode Befehle. [7, Folie 22]

III. Dalvik Bytecode

Das Aufrufen von Methoden und die Rückgabe von Werten ist in Dalvik wie auch in Java Stack-basiert. Die beim Methodenaufruf erzeugten Stack-Frames der Dalvik VM und die darin vorgenommen Berechnungen sind allerdings Register-basiert. Lokalen Variablen, welche zur Ausführung einer Methode benötigt werden, werden in virtuellen Registern gespeichert. Die virtuellen Register fassen 32-Bit-Werte. Um 64-Bit-Werte zu speichern werden benachbarte Register verwendet. Es wird gewöhnlich keine Ausrichtung der Register-Paare vorgenommen. Objekt-Referenzen benötigen ebenfalls nur ein virtuelles Register um abgelegt zu werden.

Bei einem Methodenaufruf werden die Aufrufargumente in die höchstwertigen Register des erzeugten Stack-Frames geschrieben und sind somit für die Verwendung durch Befehle verfügbar. Eine Speichereinheit im Befehlsstrom ist 16 Bit groß. In manchen Befehle werden Bits ignoriert oder müssen “0” sein, da sie nicht verwendet werden.

Die Anweisungen sind nicht unnötigerweise auf einen bestimmten Typ beschränkt. Zum Beispiel sind move-Befehle, welche den Wert eines 32 Bit Registers bewegen, generisch, ohne anzuzeigen ob sie einen Int- oder einen Float-Wert bewegen.

Da es in der Praxis unüblich ist, dass eine Methode mehr als 16 Register benötigt und da eine benötigte Registermenge von 8 durchaus üblich ist, sind viele Befehle darauf limitiert nur die ersten 16 Register zu adressieren. Falls möglich, erlauben Anweisungen Verweise auf die ersten 256 Register. Darüber hinaus existieren für manche Befehle Varianten, welche viel höhere Register-Adressen ermöglichen. Move-Befehle, welche innerhalb einer Reichweite von v0 - v65535 Register adressieren können, können dazu benutzt werden diese Limitierung zu umgehen. Liegt der Fall vor, dass keine Befehlsvarianten vorhanden ist, um auf gewisse Register zuzugreifen, so wird der Inhalt der Register vor der Operation in ein niedrigeres Register bewegt. Anschließend kann der Wert in sein ursprüngliches Register gelegt werden.

Es gibt mehrere “Pseuo-Anweisungen”, die verwendet werden um Daten im Befehlsstrom zu beherbergen. Solche Befehle dürfen nie während des normalen Programmablaufs ausgeführt werden. Weiterhin müssen diese Befehle sich auf gradzahligen Bytecode-Offsets befinden. Um diese Anforderung zu erfüllen, muss das dex-Tool einen extra *nop*-Befehl (No-Operation) als Füller einfügen. Abschließend, wenn auch nicht zwingend erforderlich, wird erwartet, dass solche ein Befehl am Ende der Methode ausgeben wird, da ansonsten zusätzliche Verzweigungsanweisungen notwendig werden, um die Pseudo-Befehle zu umgehen. [8]

Der gesamte Dalvik Befehlssatz kann unter [8] eingesehen werden.

IV. Gegenüberstellung

Zur Veranschaulichung der Unterschiede zwischen den Architekturen werden zunächst einige Java-Funktionen und deren Entsprechungen in Java Bytecode nach der Übersetzung mit javac und schließlich in Dalvik Bytecode nach der Übersetzung mit dem dex-Tool untersucht. Als Beispiele dienen neben der Addition einer Konstanten die Summenbildung über ein Array.

Beispiel 1: Addition einer Konstanten

Als erstes Beispiel werden die zwei Bytecodeentsprechungen einer Java-Methode, die zu ihrem übergebenen Integer-Argument einen konstanten Wert addiert und das Ergebnis zurückliefert betrachtet.

Java Quellcode:

```
public static int addConst(int val) {
    return val + 123456;
}
```

Java Bytecode:

```
public static int addConst(int);
  [max_stack=2, max_locals=1, args_size=1]
   0: iload_0            //legt lokale Variable 0 auf den Stack
   1: ldc #int 123456    //legt Wert aus Constant Pool auf den
Stack
   2: iadd               //addiere die obersten zwei Integer auf
dem
                         //Stack und push das Ergebnis auf den
                         Stack
   3: ireturn            //return zum Aufruf mit Integer auf Stack
```

Dalvik Bytecode:

```
public static int addConst(int);
  [regs=2, ins=1, outs=0]
  0: const v0, #0x1E240        //schreibt die Konstante (0x1E240)16
==
                               //(123456)10 in v0
   1: add-int/2addr v0, v1     //addiere v0 und v1 und schreibe das
                               //Ergebnis in v0
   2: return v0                //Rückgabe des Wertes in v0
```

[9]

In diesem einfachen Beispiel lässt sich zunächst der grundlegende architektonische Unterschied zwischen dem Java- und Dalvik Bytecode-Format erkennen. Da die JVM Stack-basiert ist, müssen die Operanden mit den Befehlen *iload_0* (lädt das erste Aufrufargument auf den Stack) und *ldc* (lädt eine Konstante an angegebenen Index aus dem Constant-Pool auf den Stack) auf den Stack geladen werden, bevor sie mit *iadd* addiert werden können. *iadd* nimmt die Operanden vom Stack und legt das Ergebnis der Addition zurück auf den Stack. Mit *ireturn* wird der Integer-Rückgabewert, der auf dem Stack liegt, an den Methodenaufrufer zurückgegeben.

Die Befehle der DVM hingegegen operieren maschinennah auf virtuellen Registern. Mit *const v0, #0x1E240* ($1E240_{16}$ entspricht 123456_{10}) wird die Konstante 123456 in den ersten virtuellen Register geladen. Da der Stack-Frame nur zwei Register benötigt und die Aufrufargumente per Konvention immer bereits in die höchstwertigen allokierten Registerplätze geladen werden, befindet sich das Aufrufargument in Register v1. *add-int v0* addiert die beiden Register und schreibt das Ergebnis in v0. *return v0* gibt den Wert im Register v0 an den aufrufenden Stack-Frame zurück.

Zu erkennen ist neben der Register-basierten Natur der DVM, dass in Dalvik Bytecode, soweit es geht, generische, das heißt nicht typ-spezifische, Operationen zur Verwendung kommen. So ist es mit dem Befehl *const* möglich, eine beliebige 32-Bit-Repräsentation einer Konstanten, also neben Integer auch Float-Werte, in ein angegebenes virtuelles Register zu laden. Im gleichen Sinn enthält *return* keine Typinformation über den Rückgabewert. Im Gegensatz dazu ist im Java Bytecode mit *ireturn* die Typinformation des Rückgabewerts explizit im Rückgabebefehl vorhanden.

Beispiel 2: Summenbildung über ein Array

Im nächsten Beispiel wird eine Methode betrachtet, die in einer Schleife die Summe über ein Array bildet und das Ergebnis zurückliefert. Der Schleifenteil ist in den Bytecode-Übersetzungen rot markiert und Lese- und Schreibbefehle in den Schleifen sind in den Kommentaren kenntlich gemacht.

Java Quellcode:

```
public static long sumArray(int[] arr) {
 long sum = 0;
 for (int i : arr) {
  sum += i;
 }
 return sum;
}
```

Java Bytecode:

```
 0000: lconst_0
 0001: lstore_1
 0002: aload_0
 0003: astore_3
 0004: aload_3
 0005: arraylength
 0006: istore 04
 0008: iconst_0
 0009: istore 05
 000b: iload 05          // rl ws
 000d: iload 04          // rl ws
 000f: if_icmpge 0024    // rs rs
 0012: aload_3                 // rl ws
 0013: iload 05          // rl ws
 0015: iaload            // rs rs ws
 0016: istore 0     6          // rs wl
 0018: lload_1                 // rl rl ws ws
 0019: iload 06          // rl ws
 001b: i2l               // rs ws ws
 001c: ladd              // rs rs rs rs ws ws
 001d: lstore_1          // rs rs wl wl
 001e: iinc 05, #+01     // rl wl
 0021: goto 000b
 0024: lload_1
 0025: lreturn
```

Legende:
rl = read local
rs = read stack
wl = write local
ws = write stack

Dalvik Bytecode:

```
0000: const-wide/16 v0, #long 0
0002: array-length v2, v8
0003: const/4 v3, #int 0
0004: move v7, v3
0005: move-wide v3, v0
0006: move v0, v7
0007: if-ge v0, v2, 0010            // r r
0009: aget v1, v8, v0               // r r w
000b: int-to-long v5, v1            // r w w
000c: add-long/2addr v3, v5         // r r r r w w
000d: add-int/lit8 v0, v0, #int 1   // r w
000f: goto 0007
0010: return-wide v3
```

Legende:
r = read
w = write

[7, Folien 38 - 40]

Hier werden zwei Vorteile der Dalvik Bytecodearchitektur besonders deutlich. Zunächst fällt auf, dass die Dalvik-Version der Methode eine erheblich geringere Befehlszahl verwendet. Dies ist durch die höhere semantische Dichte der Bytecode-Befehle der DVM möglich. Im Schleifenteil der Methode werden bei Java 14 Befehle benötigt, bei Dalvik lediglich sechs.

Ermöglicht wird dadurch, dass es, wie zuvor erläutert, in Dalvik keine unterschiedlichen Pools für lokale Variablen und Konstanten gibt, sondern alle Befehle direkt auf virtuellen Registern arbeiten. Da die Befehle in Dalvik 16 Bit groß sind, ist es möglich, die 4-Bit-großen Adressen der virtuellen Register in die Befehle zu integrieren. So können Operationen auf diesen Registern direkt ausgeführt werden und Operanden müssen nicht zuerst mit separaten Befehlen auf den Stack geladen werden.

Zum Beispiel wird die Addition zur Summe auf Long in Java in 5 Befehlen implementiert *(0018 - 001d)*. In Dalvik werden zwei Befehle, *int-to-long* und *add-long/2addr* benötigt. Im zweiten Befehl sind die beiden 4-Bit-Registeradressen der Operanden bereits enthalten. Der Wert im Register des ersten Operanden wird bei der Ausführung der Addition überschrieben und macht ein nachträgliches Speichern des Ergebnisses hinfällig (*lstore* im Java-Beispiel).

Eng verwandt mit diesem Umstand ist der zweite Vorteil der Dalvik-Version dieser Methode: Es werden deutlich weniger Lese- und Schreibzugriffe benötigt. In Java sind es in Summe 45 Lese- und 16 Schreibbefehle, demgegenüber bei Dalvik nur 19 Lese- und 6 Schreibbefehle stehen. Die hohe Zahl der Speicherzugriffe unter Java ist durch die Notwendigkeit begründet, vor jeder Operation die richtigen Operanden auf den Stack zu laden bevor sie verarbeitet werden können. wohingegen sie unter Dalvik direkt an die korrekten Register adressiert werden können. Wird unter Java nicht sofort mit dem Ergebnis der Operation weitergerechnet, kommt ein weiterer Befehl zum Speichern des auf dem Stack liegenden Ergebnis in einer lokalen Variable hinzu. Selbst die Konvertierung von Integer zu Long erfordert unter Java zwei Befehle *(0019 - 001b)* und 5 Speicherzugriffe. In Dalvik kann diese Aufgabe mit nur einem einzigen

Befehl und drei Speicherzugriffen durchgeführt werden (*int-to-long* enthält die beiden Register für den Operanden und das Ergebnis).

Erreicht werden diese Vorteile neben der Register-basierten Architektur von Dalvik mit den größeren Befehlen in Dalvik im Gegensatz zu Java (16-Bit gegenüber 8-Bit große Bytecode-Befehle). In diesem Fall ist die Gesamtgröße des Befehlsstroms unter Dalvik mit 18 Byte zwar trotzdem kleiner als in Java mit 25 Byte, weil besonders viele Befehle eingespart werden können, allerdings ist in der Regel damit zu rechnen, dass Dalvik Bytecode etwa 35% mehr Speicherplatz im Befehlsstrom braucht. Dieser Nachteil wird dadurch aufgewogen, dass die DVM während der Ausführung immer zwei Byte auf einmal au dem Befehlsstrom verarbeitet, während die JVM nur ein Byte pro Befehl verarbeitet. [7][8]

Beispiel 3: Initialisierung eines Arrays

Java Quellcode:

```
private static final int[] S33KR1T_1NF0RM4T10N = {
    0x4920616d,  0x20726174,  0x68657220,  0x666f6e64,  0x206f6620,
    0x6d756666, 0x696e732e
};
```

Java Bytecode:

```
0000: bipush #+07
0002: newarray int
0004: dup
0005: iconst_0
0006: ldc #+4920616d
0008: iastore
0009: dup
000a: iconst_1
000b: ldc #+20726174
000d: iastore
000e: dup
000f: iconst_2
0010: ldc #+68657220
0012: iastore
0013: dup
0014: iconst_3
0015: ldc #+666f6e64
0017: iastore
0018: dup
0019: iconst_4
001a: ldc #+206f6620
001c: iastore
001d: dup
001e: iconst_5
001f: ldc #+6d756666
0021: iastore
0022: dup
0023: bipush #+06
0025: ldc #+696e732e
0027: iastore
0028: putstatic Example2.S33KR1T_1NF0RM4T10N:[I
002b: return
```

Dalvik Bytecode:

```
0000: const/4 v0, #int 7            // #7
0001: new-array v0, v0, int[]
0003: fill-array-data v0, 000a
0006: sput-object v0,
         Example2.S33KR1T_1NF0RM4T10N:int[]
0008: return-void
0009: nop                           // Platzhalter
000a: array-data                    // für fill-array-data @ 0003
          0: 1226858861             // #4920616d
          1: 544366964              // #20726174
          2: 1751478816             // #68657220
          3: 1718578788             // #666f6e64
          4: 544171552              // #206f6620
          5: 1836410470             // #6d756666
          6: 1768846126             // #696e732e
```

[7]

In diesem Beispiel wird ein Integer-Array der Größe sieben angelegt und mit Werten gefüllt. Die Entsprechung in Java- und Dalvik Bytecode macht den Vorteil einer der Pseudo-Befehle von Dalvik deutlich. In Java wiederholt sich für jeden Array-Index der folgende Ablauf:

1. Die auf dem Stack liegende Referenz auf das Integer-Array wird dupliziert (*dup*).
2. Der Index, an dem im Array gespeichert werden soll, wird auf den Stack geladen (*iconst_0*).
3. Der Wert, der im Array gespeichert werden soll, wird auf den Stack geladen (*ldc*).
4. Die Referenz auf das Array, der Index und der zu schreibende Wert werden vom Stack gelesen und das Array wird mit dem Wert am gegebenen Index gefüllt. (*iastore*).

In Summe werden in Java also für die einfache Anwendung ein Array explizit zu initialisieren für jeden Index vier Befehle benötigt. Bei sieben Indizes sind es in diesem Beispiel also 28 Befehle um die Werte im Array zu speichern, aus Performance Sicht eine Katastrophe.

In Dalvik existiert für diesen Anwendungsfall hingegen der Befehl *fill-array-data* zum Füllen des Arrays und der pseudo-Befehl *array-data*. Letzterer dient dazu, die Daten zur Füllung eines Arrays im Befehlsstrom selbst zu beherbergen. *array-data* ist ein pseudo-Befehl, da er während dem normalen Programmablauf niemals vom Interpreter als Befehl konsumiert wird (er liegt im Beispiel hinter *return-void* und ist damit nicht erreichbar). Wohl aber dient er dem Befehl *fill-array-data*, indem dieser aus der angegebenen Stelle im Befehlsstrom (in diesem Fall *000a*) die Daten für das zu füllende Array lesen kann. Der *array-data* Befehl liegt per Konvention immer am Ende des Stack-Frames der Methode und wird durch einen Platzhalter-Befehl (*nop)* vom eigentlichen Programmablauf getrennt. Der Platzhalter-Befehl dient auch dazu, eine 2-Byte-Ausrichtung der Arraydaten zu erreichen, damit Sie schneller durch die VM interpretiert werden können. [7][8]

V. Fazit

Die Historie zeigt, dass der Dalvik Bytecode ein erfolgreiches Modell ist. Auch wenn die Dalvik VM mittlerweile durch eine Laufzeitumgebung mit vollständig nativ übersetztem Code, ART (Android Runtime), ersetzt wurde, so hat der Dalvik Bytecode und mit ihm das dex-Tool diesen Wandel überdauert.[10]

Wie bisher wird zuerst Java Code geschrieben, der in Java Bytecode übersetzt wird, bevor daraus eine .dex-Datei erzeugt werden kann, die anschließend zur .apk-Datei (Android Application Package) zusammengefügt und als Anwendung verteilt werden kann. Der Dalvik Bytecode hat sich durch seine Vorteile gegenüber Java Bytecode auf der Android-Plattform etabliert. Die maschinennahe Auslegung zur Erhöhung der Interpretationsgeschwindigkeit und die cleveren Mechanismen (Die Redundanzvermeidung durch den gemeinsamen Constant-Pool, teilen der Android Bibilotheksfunktionen u.a.) zur Einsparung von Speicherplatz während der Ausführung trugen anfangs stark zu seinem Erfolg bei. Heute ist es mehr das gesammelte Momentum und die Unterstützung durch die verschiedenen Tools und die damit einhergehenden Optimierungsmöglichkeiten die im Android Build-Prozess zur Anwendung kommen. Die Vorteile, die aus der maschinennahen Auslegung der Bytecode-Architektur erwachsen, um Dalvik Bytecode schnell interpretierbar zu machen, sind heute diejenigen, die ihn für die schnelle Kompilierung durch ART besonders geeignet machen.

Der Dalvik-Befehlssatz bleibt unserer Meinung nach für die nähere Zukunft der Stand der Technik in Android. Die in Java Bytecode inhärenten Ineffizienzen wie die Verkomplizierung der Befehlsmenge durch die explizite Typprüfung und die Aufblähung der Befehlszahl durch die Verwendung des Operandenstacks machen Java Bytecode keine Wahl für ein mobiles Betriebssystem wie Android, das mit den Einschränkungen einer mobilen Plattform hinsichtlich Speicher, Prozesserleistung und den Kosten von Ineffizienz der Ausführung für den Akkubetrieb auskommen muss. Jeder verschwendete Prozessorzyklus ist schließlich eine Einschränkung der Akkulaufzeit und die ist am Markt eines der wichtigsten Kaufkriterien.

Die Neuentwicklung eines Bytecode-Befehlssatzes für ART oder die Schaffung eines neuen Build-Prozesses, bei dem Maschinencode das direkte Kompilierungsziel ist, ist zum heutigen Zeitpunkt sinnlos. Der Nutzen eines neuen Build-Prozesses ist für die Autoren nicht ersichtlich und der verbundene Entwicklungsaufwand wäre immens. Tatsächlich ist es doch so, dass Google selbst mittlerweile mit Android Studio eine IDE (Integrated Development Environment, zu deutsch Entwicklungsumgebung) entwickelt hat, um den etablierten Build-Prozess besser zu Unterstützen. Die hohe Portabiliät, die durch architekturunabhängigen Bytecode ermöglicht wird, ist für ein auf einer so heterogenen Menge von Architekturen weit verbreitetes Betriebssystem wie Android zwingend notwendig, da der Mehraufwand an Infrastrukturleistung zur Verteilung der vorkompilierten Binaries und die Last für Gerätehersteller, Compiler zu schreiben viel zu hoch wäre.

Dalvik Bytecode wird uns noch über viele zukünftige Android-Versionen hinweg begleiten.

VI. Quellenverzeichnis

[1] o.V., “Chapter 2. The Structure of the Java Virtual Machine”, https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.11.1, abgerufen am 30. Juli 2015.

[2] o.V., *“Chapter 6. The Java Virtual Machine Instruction set”*, http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html, abgerufen am 30. Juli 2015.

[3] o.V., *“TIOBE Programming Community Index”*, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, abgerufen am 30.Juli 2015.

[4] o.V., *“Dalvik (software)”*, https://en.wikipedia.org/wiki/Dalvik_(software), abgerufen am 30.Juli 2015.

[5] Hagen Patzke, *“Endlich kompiliert”*, CT Ausgabe 13, 2015, Seiten 172-175

[6] o.V., *“Oracle America, Inc. v. Google, Inc.”*, https://en.wikipedia.org/wiki/Oracle_America,_Inc._v._Google,_Inc., abgerufen am 30.Juli 2015.

[7] Dan Bornstein, *“Dalvik VM Internals”*, cs.nyu.edu/courses/fall14/CSCI-GA.3033-010/dalvikInternals.pdf, abgerufen am 30.Juli 2015.

[8] o.V., *“Dalvik Bytecode”*, Android Open Source Project http://source.android.com/devices/tech/dalvik/dalvik-bytecode.html#design, abgerufen am 30.Juli 2015.

[9] Michael Starzinger, *“Daneel: The difference between Java and Dalvik”*, http://www.antforge.org/blog/2011/04/27/daneel-difference-between-java-and-dalvik, abgerufen am 30.Juli 2015.

[10] o.V., *“ART and Dalvik”*, http://source.android.com/devices/tech/dalvik/, abgerufen am 20. Juli 2015.