Darum laufen Android Applikationen immer besser

Im Innern des Androiden

Die Dalvik Virtual Machine ist der eigentliche Motor des Android-Systems. Durch Tuning an diesem Motor schafft es Google, dass das mobile Betriebssystem von Version zu Version performanter wird.

Carlo U. Nicola

README

Dieser Artikel zeigt die Unterschiede von Dalvik gegenüber einer Standard Java Virtual Machine und erklärt die Vorteile der Dalvik VM.

Android ist nicht nur ein Betriebssystem sondern auch eine Software-Plattform (Java) für mobile Geräte. Der eigentliche Motor dieses Systems ist die Dalvik Virtual Machine (DVM). Google hat sie von Anfang an für anspruchsvolle Java-Applikationen (Version 5.0 beziehungsweise 1.5) konzipiert, mit dem Ziel, diese Anwendungen auf minimalen Hardware-Ressourcen möglichst effizient laufen zu lassen.

Die bescheidenen Ressourcen lassen sich wie folgt zusammenfassen: eine CPU mit einer Taktfrequenz von 500 MHz; 64 Megabyte Speicher, von denen nur 20 Megabyte für Applikationen zur Verfügung stehen; ein Linux Betriebssystem ohne Swap-Bereich und das ganze Gerät nur mit Batterien betrieben. Die für die Entwicklergemeinde jedoch wichtigste Randbedingung ist, dass alle Android-Applikationen mit einem gewöhnlichen Java SDK 5.0 geschrieben werden dürfen. Diese letzte Forderung wurde auch in der Tat umgesetzt, obwohl Java-Bytecode nicht direkt auf der DVM lauffähig ist. Java Bytecode wird für die Dalvik VM speziell kompiliert und in einer Dex-Datei (`.dex`) abgelegt.

Dieser Artikel beschreibt, worin sich die die Dalvik VM von der Standard Java VM unterscheidet, welche Rolle die Registerstruktur der Dalvik VM auf die allgemeine Performance des Android-Programmierungsmodells hat und wie sich das Dex-Format auf die Effizienz der DVM auswirkt. Er geht auf folgende Punkte ein:

- die Rolle des Programms Zygote beim Starten der Dalvik VM
- die Rolle der Registerstruktur der Dalvik VM auf die allgemeine Performance des Android-Programmierungsmodells
- der Einfluss des neuen Class-File-Formats von Dalvik auf die Effizienz der VM.

Am Ende des Artikels finden Sie auch Details zum neuen Just-in-Time-Compiler von Android 2.2.

Aller Anfang

Bevor wir uns dem Starten von Android-Anwendungen zuwenden, werfen wir zuerst einen kurzen Blick auf das generelle Startprozedere beim Ablauf des Android Boot-Prozesses: Der `init.rc`-Prozess startet verschiedene Systemdienste (Daemons). Anschließend ruft er den Service Manager und Zygote auf. Zygote sorgt im weiteren Systemablauf dafür, dass jede Android-Anwendung als normaler Linux-Prozess gestartet wird. Der System-Manager stellt die Verbindung mit dem Linux-Kernel her, damit die Dienste, welche via Kernel-Treiber auf die Hardware zugreifen, später auch den Android-Anwendungen via API zur Verfügung stehen.

Abbildung ❶ zeigt diese drei Schritte schematisch und schlägt auch die Brücke zum Start der eigentlichen Android-Anwendungen. Wie erwähnt, sorgt der Prozess Zygote dafür, dass eine Android-Anwendung als normaler Linux-Prozess mittels `fork()` gestartet wird. Jeder Linux-Prozess schleppt dabei die Bionic Library (Android-Version der Standard-C-Library `libc`) mit. Gleichzeitig startet Android zu jedem Prozess eine DVM mit eigenem Heap und

Stack und stellt die notwendigen Bindings zu den Java-Core-Bibliotheken her. Dies hat verschiedene Vorteile, die sich direkt in der Struktur der DVM widerspiegeln: Zum einem ist das Problem der Sicherheit der Android-Java-Applikation (inbegriffen Byte Code Verifizierung) fast zu 100% durch die Linux-Prozesse garantiert. Zum andern kann Zygote vor dem Applikationsstart die wichtigsten Java-Pakete (Core Libraries) im speziellen Dex-Format in einen geschützten Speicherbereich laden, auf den alle Android-Applikationen über die DVM zugreifen dürfen. Die Auswirkungen dieses Designentscheids auf die Arbeit des Garbage Collectors erklärt der Abschnitt „Shared Memory und Garbage Collection“ weiter unten.

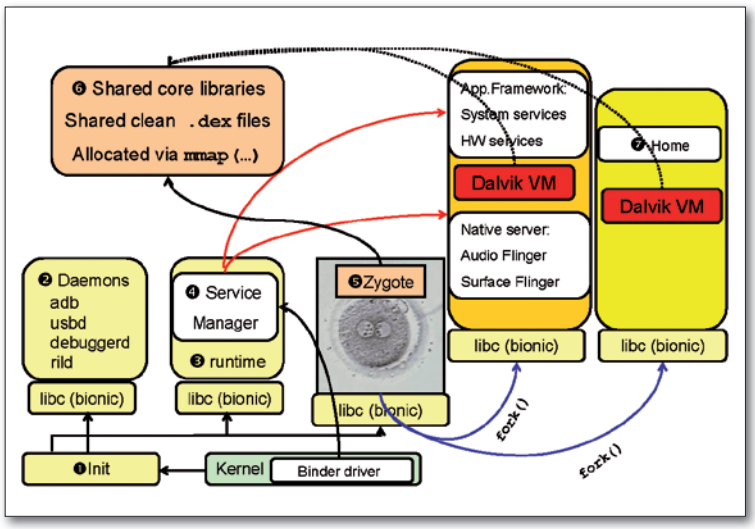
Die Dalvik Virtual Machine

Die gewöhnliche Java Virtual Machine (JVM) ist eine Stack-Maschine, deren Instruktionen mit Bytecode dargestellt sind. Der Interpreter der JVM führt pro Bytecode den Dispatch-Prozess aus, welcher aus drei Phasen besteht:

- `fetch`: der aktuelle Bytecode wird vom Stack geholt;
- `decode`: in Abhängigkeit des Bytecode-Typs auf die notwendigen Argumente auf dem Stack zugreifen;
- `execute`: die durch den Bytecode dargestellte Funktion ausführen. Diese Phase beansprucht freilich am meisten Zeit.

Listing 1 zeigt ein Beispiel für einen in C konziperten Interpreter.

In dieser Implementierung ist der Bytecode-Stack als Array dargestellt und die Dispatch- Stufe ist durch einen doppelten indirekten Zugriff via C-Zeiger kompakt und elegant realisiert. Diese einfache C-Funktion illustriert auch klar die drei Phasen des Interpreters, die oben erwähnt wurden. Nämlich wird `op_x` zuerst geholt (als Argument von `interp(...)`; dann decodiert via `DISPATCH()` und schließlich ausgeführt (Funktion Aufruf nach der entsprechenden Bytecode Marke). Weiter geht mit dem wiederholten Aufruf von `DISPATCH()` am Ende der Zeile.



❶ Der `init.rc`-Prozess in Android. Glossar der Abkürzungen: adbd: Android debugger; usbd: USB Schnittstelle; rild: radio interface layer; debugerd: debug system. Weitere Daemons werden gestartet, sind aber in der Abbildung nicht dargestellt. Audio und Surface Flinger: eine Google Bezeichnung für MP3- beziehungsweise graphische Dienste (modifiziert aus [4]). Die eingekreisten Ziffern geben die zeitliche Reihenfolge der Prozesse an.

Register vs. Stack

Trotz sehr vieler Ähnlichkeiten unter scheiden sich die DVM und JVM in zwei Punkten grundlegend Die DVM ist nicht als Stack-Maschine, sondern als Register-Maschine realisiert, und die Längen der Opcodes beträgt bei der DVM zwei anstatt nur einem Byte. Eine auf Register basierte virtuelle Maschine holt ihre Bytecodes und Operanden aus (virtuellen) Registern. Dazu ist es natürlich erforderlich, dass die Operanden in Abhängigkeit des Opcodes in bestimmte Register geschrieben und von dort wieder gelesen werden und nicht generell vom Stack geholt werden, wie es in einer Standard JVM geschieht.

Die Vorteile einer Register- gegenüber einer Stack-Maschine sind auf den ersten Blick nicht so evident, besonders wenn man zusammen mit der obigen Bemerkung bezüglich virtueller Register auch die `DISPATCH()`-Phase des Interpreters analysiert (siehe Listing 1). Solange der Stack und die virtuellen Register als lineare Arrays implementiert sind, unterscheiden sich die Realisierungen von `DISPATCH()` nur unwesentlich. Erst wenn die virtuellen Register auch wirklich in echten Prozessorregistern abgebildet werden, kann von einem Zeitgewinn bei der Ausführung der `DISPATCH()`-Anweisung ausgegangen werden. Mit einer Abbildung von virtuellen in reale Register handelt man sich aber auch einen gewaltigen Nachteil ein: Wenn man eine VM so modelliert, dass sie eine konkrete Prozessorarchitektur annähert, dann verliert man alle Portabilitätsvorteile einer herkömmlichen Stack-Maschine, die ja auf jeder erdenklichen Prozessorarchitektur realisierbar ist.

Auf der anderen Seite kann eine VM, die besser an die Hardware angepasst ist, direkt mit Maschinensprache umgehen und somit

LISTING 1

```
#define DISPATCH() { goto *op_table[*((s)++) - 'a']; }
static void interp(const char* s) {
    static void* op_table[] = {
        &&op_a, &&op_b, &&op_c, &&op_d, &&op_e
    };
    DISPATCH();
    op_a: printf("Hell"); DISPATCH();
    op_b: printf(" or"); DISPATCH();
    op_c: printf(" Para"); DISPATCH();
    op_d: printf("dise!\n"); DISPATCH();
    op_e: return;
}
```

Listing 1: Struktur eines C-Interpreter-Programms. Der Stack ist mit einem linearen Array realisiert. Die Bytecodes, die als Argument der Funktion `interpret(...)` erscheinen, werden via `DISPATCH()` durch doppelte Indirektion zur richtigen Marke geführt, wo schließlich der dekodierte Bytecode ausgeführt wird.

auf komplizierte Just-In-Time-Kompilierung verzichten. Diesen Vorteil macht sich auch die DVM zu nutzen, da sie sich primär auf die ARM-Architektur ausrichtet. Die DVM kann auf maximal 64K virtuelle Register zugreifen, die natürlich im L2- beziehungsweise Haupt-Speicher abgebildet werden müssen. Java Methoden, die mehr als 16 Argumente und Parameter benötigen, sind jedoch extrem selten, so dass üblicherweise alle Argumente und Parameter bei einem Methodenaufruf in den 16 vorhandenen 16-Bit-Registern des ARM-Prozessors Platz finden.

Interessanter wird es, wenn der Vergleich zwischen einer Stack und einer Register basierten VM auch auf die Bytecodelänge ausgeweitet wird. Wir vergleichen dazu in Tabelle 1 die Anzahl Code-Bytes für die Funktion `tryItOut(...)` aus Listing 2, welche zwei Integer-Parameter addiert und das Resultat zurückgibt.

Bei einer hypothetischen Register-VM müssen die beiden Operanden der Addition in Register kopiert werden. Bei der Dalvik VM sorgt bereits der Aufrufer der Methode für das Ablegen der Übergabeparameter in Registern. Das Zwischenspeichern des Resultats der Addition bevor es zurückgegeben wird, ist im Java Code klar ersichtlich und wichtig, weil `ireturn` das Resultat nur aus dem Operand-Stack holen kann.

Wie das Beispiel in Tabelle 1 zeigt, erreicht die Dalvik-VM die theoretisch mögliche Bytecode-Einsparung einer Register VM üblicherweise nicht. Dabei sollte man aber nicht vergessen, dass die DVM mit 16 Bit langen Opcodes arbeitet, was in Klartext bedeutet, dass die DVM pro Lesevorgang doppelt so viele Bytes lädt wie die Standard JVM. Gerade bei heute üblichen 16-Bit, 32-Bit- oder 64-Bit-Architekturen ist die Bearbeitung von lediglich 8 Bits pro Opcode eine Ressourcenverschwendung. Da bei einer Register VM die Argumente und Parameter in echte Prozessorregister geladen werden, ist deren Zugriff jedoch entsprechend schneller als derjenige auf einen externen Stack.

Ein weiterer Vorteil einer Register VM besteht darin, dass die typische Nebenbedingung einer Stack VM ersatzlos wegfällt: Am Ende einer Methode muss der Opcode Stack nicht im gleichen Zustand wie am Anfang sein.

Zusammengefasst lässt sich sagen, dass Register VM eine bessere Effizienz bei der Bearbeitung der Opcodes versprechen, und wenn Argumente und Parameter einer Java Methode in den Prozessorregistern Platz finden, ist auch die Ausführungszeit kleiner.

Wenn man auf eine portierbare Implementierung des Interpreters verzichtet (was ja Dalvik auch tut), kann man ihn auch sehr elegant und kompakt in der jeweiligen CPU- Maschinensprache programmieren. Eine besonders gelungene Implementierung ist diejenige für die ARM-CPU-Familie, die zurzeit in allen Android Smartphones eingesetzt wird [4].

Dalvik Opcodes

Alle 220 Opcodes von Dalvik werden einheitlich mit 16 Bit definiert. Dies ist nicht nur ein bewusster Entscheid, um eventuelle juristische Streitigkeiten mit Sun zu vermeiden, sondern auch eine sinnvolle Anpassung der VM an die realen Prozessorarchitekturen, die schließlich diese VM auch verwirklichen.

Die Struktur der Opcodes lässt sich gut anhand der Methode `public void tryFinally()` aus Listing 2 illustrieren.

In Listing 3 sind #6, #17 mit `Lch/fhnw/examples/FinallyInternal;.tryItOut:()V` und #5, #20 mit `Lch/fhnw/examples/FinallyInternal;.wrapItUp:()V` äquivalent. Zum Verständnis des Java 1.4 Codes ist es hilfreich zu wissen, dass jsr die Rücksprungadresse auf den Operanden Stack rettet. Somit wird in Instruktion 4 zur Instruktion 14 gesprungen, dort die gerettete Rücksprungadresse (7 return) vom Stack in die lokale Variable 2 gespeichert, dann die Methode der `finally`-Klausel ausgeführt und schließlich mit `ret 2` an die in Variable 2 gespeicherte Adresse zurückgesprungen. Der Opcode jsr ist somit eine Art von Java-Methodenaufruf, der aber die JVM-Spezifikation verletzt [1], [5]. Man hat dies ab Java 1.5 kor-

TABELLE 1

Java VM 1.6	Register VM	Dalvik VM (16 Bit Opcode)
0 iload_1	move v11 -> v2	
1 iload_2	move v12 -> v3	
2 iadd	iadd v2 v3 -> v0	0000: add-int v0, v2, v3
3 istore_3		
4 iload_3		
5 ireturn	ireturn v0	0002: return v0
————	————	————
6 Bytes	4 Bytes	6 Bytes

Tabelle 1: Vergleich Anzahl generierte Bytecodes bei einer Stack-VM (Java VM 1.6) und einer hypothetischen Register VM. In der dritten Spalte die tatsächlichen Bytecodes, welche von Dalvik 1.5 generiert werden. Die `add-int`-Instruktion benötigt mit ihren Parametern vier Bytes.

LISTING 2

```
public class FinallyInternal {
        public void tryFinally() {
                try {
                        tryItOut(1, 2);
                } finally {
                        wrapItUp();
                }
        }

        private int tryItOut(int r1, int r2) {
                int retVal;

                retVal = r1 + r2;
                return retVal;
        }

        private void wrapItUp() {
        }

    }
```

Listing 2: Das Test-Programm, das für die Analyse in Listing 3 benutzt wurde. Hier wurde speziell untersucht, wie die Klausel `try{...} finally{...}` von der Dalvik VM übersetzt wird.

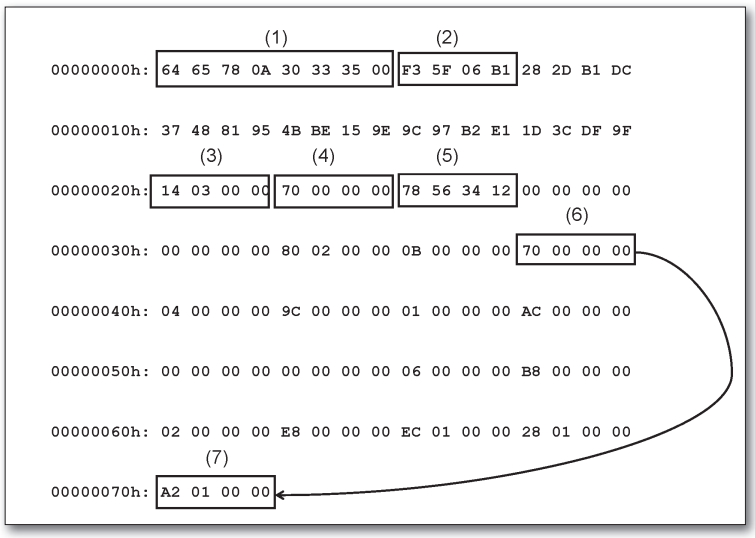
rigiert. Die DVM von Android hält die JVM-Spezifikation ein und generiert erst noch besser lesbaren Code.

Der Aufbau eines Dalvik Opcodes lässt sich wiederum gut an einem Beispiel zeigen. Die Anweisung `invoke-virtual {v4,v0,v1,v2,v3}, Foo.method6:(||||)V` ruft eine Instanzmethode der Klasse Foo auf, konkret die sechste Methode in der Methodentabelle, die neben der `this`-Referenz noch vier Parameter v0, v1, v2 und v3 benötigt und keinen Rückgabewert liefert. v4 ist dabei die `this`-Referenz. Die Anweisung wird wie folgt codiert: `0x6E53 0x0600 0x0421`. 6E codiert die eigentliche Instruktion invoke-virtual; in 53 steht die 5 für die Anzahl Parameter und 3 für v3; 0600 bezeichnet den Index in der Methodentabelle (Methode 6); 0421 bezeichnet die Parameter v0, v4, v2, v1. Dabei stellen die vx die (virtuellen) Register der DVM dar.

Dalvik spezifiziert auch Instruktionen, die erst nach der Verifizierung der Bytecode- Dateien (Dateinamenserweiterung `.dex`, siehe unten) vom Programm `dexopt` benutzt werden können. Dexopt gestaltet die Dex-Dateien effizienter. Dabei werden grundsätzlich drei Dinge [2] optimiert:

- **Alignement:** Die Opcodes und die Daten müssen nach 16 Bit ausgerichtet sein. Spezielle Ausrichtungen wie zum Beispiel 64 Bit werden explizit in Assembler-Code vorgenommen. Dexopt führt zusätzliche Nop-Instruktionen (no operation) ein, um die gewünschte Ausrichtung zu erreichen.
- **Virtuelle Methoden:** Für alle Aufrufe virtueller Methoden wird der Methodenindex durch einen Index in eine Vtable ersetzt, die die Startadresse des Code-Teils im `.dex`-File beinhaltet, Dadurch wird eine Indirektion eingespart. Dies ist besonders wichtig, weil Zygote vorkompilierte Klassen herunterlädt, die alle zukünftigen Instanzen der DVM benutzen.
- **Effizientere Opcodes:** Einzelne Opcodes können durch bessere ersetzt werden, wie zum Beispiel `invoke-virtual-quick`, welcher besonders effizient mit der Vtable des Zielobjektes arbeitet.

Die so optimierten Dex-Dateien bezeichnet man mit der Dateinamenserweiterung `.odex`.



❷ Beispiel eines konkreten Header-Segmentes einer Dex-Datei.

Dalvik Executable Format

Ein so grundlegender Abschied vom klassischen Java `.class`-Opcode-Format, war dem DVM Team auch eine willkommene Möglichkeit, Korrekturen vorzunehmen, die den aktuellen Stand der Java-Erfahrungen widerspiegelten. Man sollte nur an die gewaltigen Probleme denken, die die Sun-Ingenieure bewältigen mussten, um das Konzept von Generics in Java zu realisieren, ohne die Struktur der Java Opcodes zu verändern!

Das Dalvik Executable Dateiformat (Dex-Format) ist in Segmente unterteilt, deren Reihenfolge zwingend vorgegeben ist (siehe Details in [6]).

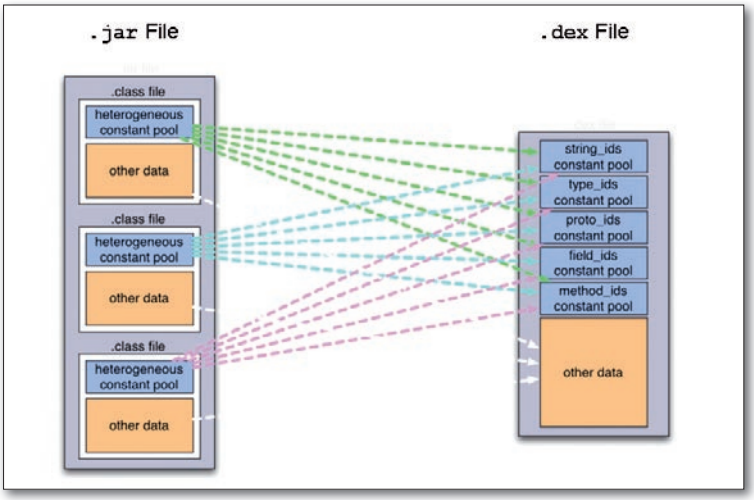
In Abbildung ❷ zeigen wir ein konkretes Beispiel eines Header-Segmentes: Die nummerierten Kästchen sind wie folgt zu interpretieren: (1) „dex?035?“: Magische Zahl und Version; (2) 32-Bit CRC Checksumme aller Bytes mit Ausnahme der ersten 12; (3) Länge der Datei in Bytes; (4) Länge des Headers in Bytes; (5) Little-Endian CPU (die umgekehrte Reihenfolge der Bytes würde eine Big-Endian CPU voraussetzen); (6) Startadresse des Segments `string_ids`; (7) Startadresse des ersten Bezeichners in der Tabelle `string_id_item`.

LISTING 3

Java VM -- 1.4	Java VM -- 1.5
0 aload_0	0 aload_0
1 invokevirtual #6	1 invokespecial #17
4 jsr 14	4 goto 14 (+10)
7 return	7 astore_1
8 astore_1	8 aload_0
9 jsr 14	9 invokespecial #20
12 aload_1	12 aload_1
13 athrow	13 athrow
14 astore_2	14 aload_0
15 aload_0	15 invokespecial #20
16 invokevirtual #5	18 return
19 ret 2	

```
Dalvik VM
0000:invoke-direct{v1}, Lch/fhnw/examples/
FinallyInternal;.tryItOut:()V
0003:invoke-direct {v1}, Lch/fhnw/examples/
FinallyInternal;.wrapItUp:()V
0006: return-void
0007: move-exception v0
0008: invoke-direct {v1}, Lch/fhnw/examples/
FinallyInternal;.wrapItUp:()V
000b: throw v0
```

Listing 3: Opcodes für die Methode `tryFinally()` generiert mit drei Java Compilern: a) Java 1.4: Anzahl Bytes 20; b) Java 1.5: Anzahl Bytes 19; c) Dalvik 1.5: Anzahl Bytes 22



❸ Das Dex-Format fasst alle heterogenen `constant_pool`-Bereiche zusammen, die eine jar-Datei für jede mitgeschleppte Klasse neu definiert (aus [4]).

Während die einzelnen Bytecode-Dateien (`.class`) eines für die Standard JVM kompilierten Java-Programmes üblicherweise in einem (komprimierten) Java-Archiv (`.jar`) zusammengefasst werden, benötigt die DVM kein zusätzliches Archiv-Format, da das Dex-Format einzelne Klassen und eine beliebige Kollektion von Klassen definieren kann.

Das wichtigste Merkmal des Dex-Formats besteht darin, dass alle Strings zur Bezeichnung von Klassen, Methoden usw. nur einmal in der gesamten Datei gespeichert werden. Alle Wiederholungen werden konsequent gestrichen und nicht wie im Jar-Format für jede neue Klasse noch einmal im `constant_pool` Bereich definiert (Abbildung ❸). Damit wird die Dateilänge einer Dex-Datei im Durchschnitt um 35% kürzer als diejenige einer äquivalenten, unkomprimierten Jar-Datei.

Da das Dex-Format Informationen für den Android Debugger (`adb`) codiert, werden diese im Data-Segment mit dem DBG-Präfix speziell gekennzeichnet. Die Art und Weise wie diese Debug-Information definiert wurde, ist maßgeblich aus der DWARF Debugging Format Spezifikation übernommen worden [7].

Dex-Dateien überprüfen

Die Überprüfung (verify) der Dex-Dateien weicht sehr stark vom Standard Java-Verfahren ab. Der Grund liegt darin, dass Zygote mittels Memory-Mapping viele Klassen und Applikationen (so genannte boostrap classes) ins `dalvik-cache`-Verzeichnis lädt ohne dazu einen Class Loader zu bemühen. Dies bedeutet, dass die komprimierten Dex-Dateien nicht nur dekomprimiert, sondern vor dem Speichern auch einer Vorverifizierung unterzogen werden. Nach der Vorverifizierung aber noch vor der Speicherung optimiert das System die Dex-Dateien mit `dexopt`. Das Ganze dient auch dazu, die Android-Apps schneller zu starten.

Zygote muss während der Vorverifizierung einige vernünftige Annahmen treffen, um nicht später mit dem Class Loader der DVM in Konflikt zu geraten [3]. Als Beispiel gehen wir von einer Applikation `MyApp.apk` aus, die eine eigene String-Klasse im Package `java.lang` unter dem Namen String definiert und somit in Konflikt mit der Standardklasse `java.lang.String` kommt. Grundsätzlich könnte man in der Vorverifizierungsphase annehmen, dass die echte Implementierung von `java.lang.String` bereits in den `core.jar`-Klassen definiert wurde. Somit könnte Zygote unser Programm `MyApp.apk` weiter überprüfen und am Ende noch optimieren.

Das ist nicht sehr klug, weil beim späteren Laden unserer Klasse in der DVM das System merken wird, dass etwas nicht in Ordnung ist. Daher wählt Dexopt eine bessere Strategie: Sobald eine Klassendefinition gefunden wird, die die gleiche Signatur einer früher vorverifizierten Klasse beinhaltet, stoppt Dexopt ohne jegliche Verifizierung oder Optimierung vorzunehmen. Es ist dann die Aufgabe der DVM diese Klassendefinition zu verifizieren. Der Verifikationsprozess in der DVM soll dabei strikt achten, dass alle Referenzen zu den Klassen entweder in unserer Applikation oder in einer früheren `bootstrap.apk` vorhanden sind, um zu vermeiden, dass ein benutzerdefinierter Class Loader zum Beispiel eine neue Version der `core.jar`-Klassen (zum Beispiel mit einem Virus bestückt!) lädt.

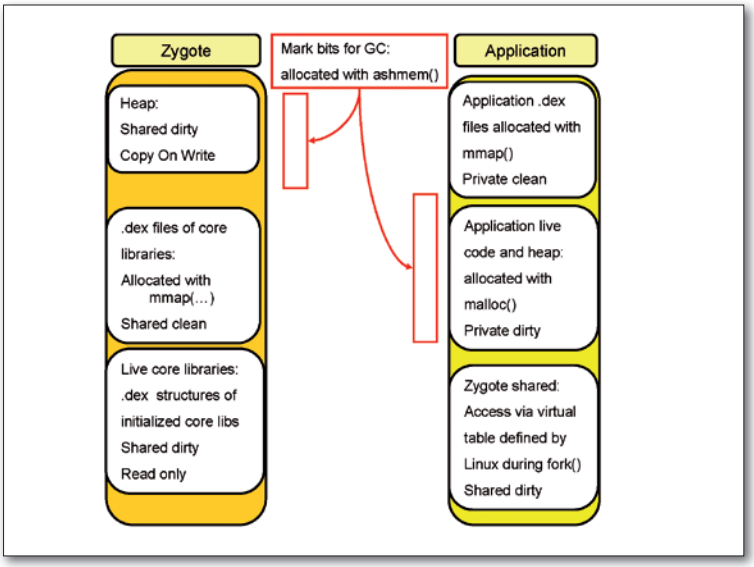
Es geht noch besser

Neben der Lösung des eben beschriebenen Problems soll Dexopt weitere Checks durchführen [3]. So soll sichergestellt werden, dass (i) nur legale Dalvik-Opcodes in den Methoden benutzt werden; (ii) `move-exception` als erste Instruktion in einem Exception Handler auftritt; (iii) `move-result*` nur unmittelbar nach einer `invoke-*` oder `filled-new-array`-Instruktion vorkommt und (iv) `dexopt` nur solche Rücksprünge auf den Stack verbietet, bei der eine `new-instance`-Instruktion auf eine nicht initialisierte Registerreferenz hinweist.

Viele Standard-Checks einer JVM sind in der DVM ohne jegliche Bedeutung. Zum Beispiel hat die DVM keinen Operanden-Stack, daher ist eine Überprüfung desselben unnötig, und die Typen-Restriktion auf Referenzen im `constant_pool` fällt in einer DVM weg, weil kein `constant_pool` in einer Dex-Datei spezifiziert ist.

Optimierungen innerhalb des Dex-Formats sind dagegen eine Besonderheit der DVM. Sie lassen sich wie folgt klassifizieren [2]:

- Optimierungen, die Instruktionen und Daten auf 16-Bit ausrichten. Oft bedeutet dies das



❹ Zusammenhang zwischen den globalen (shared) Bereichen von Zygote und denjenigen einer beliebigen Android-Applikation.

Einführen von `nop`-Instruktionen im Code-Teil und das Auffüllen (padding) der Datenbereiche bis die gewünschte Ausrichtung erreicht worden ist.

- Das Ausschneiden (prune) von leeren Methoden.
- Inline-Austausch von oft verwendeten Methoden mit direkten Aufrufen auf native Implementierungen. Anstatt virtuellen Methodenaufrufe via `invoke-virtual*` und über einen Methodenindex zu tätigen, werden Instruktionen wie `invoke-quick` verwendet, die effizienter auf absolute Adressen in einer Sprungtabelle (vtable) via Index zugreifen.
- Daten (zum Beispiel Hash-Werte) im Voraus berechnen, um diese Berechnungen in der DVM zu vermeiden.

Shared Memory

Die meisten Betriebssysteme klassifizieren Speicher in vier Qualitätsklassen, die nach Art der Allokation und nach Art der Zugänglichkeit definiert sind:

- **Shared clean:** Der Speicher ist allen Prozessen zugänglich (global). Da alle Android-Applikationen via `fork()` durch Zygote erzeugt werden, bedeutet dies, dass sie auf bestimmte Speicherregionen von Zygote zugreifen können. Clean bezeichnet die Art wie Zygote diese gemeinsamen Speicherregionen mit Informationen gefüllt hat. In diesem Fall benutzt Zygote `mmap(...)` für schnelles Laden/Löschen eines binären Speicherabbildes durch das Betriebssystem (Memory-Mapping).
- **Shared dirty:** Wie „shared clean" aber die Allokation/Freigabe erfolgt mit `malloc(...)` und `free(...)`, was langsam und von jeder Applikation abhängig ist.
- **Private clean:** Hier bedeutet private prozessspezifisch. Beispiel: Lokale Aufbewahrung der spezifischen Dex-Dateien jeder einzelnen Android-Applikation, die mit `mmap(...)` zugewiesen worden sind.
- **Private dirty:** Wie „private clean" aber die Allokation/Freigabe erfolgt mit `malloc(...)` und `free(...)`. Beispiel: Applikations-Heap .

Diese Speicherklassifizierung ist eine Grundvoraussetzung für die sparsame Speicherverwaltung, wenn Zygote mehrere Android-Applikationen starten soll. Die Devise lautet dabei, möglichst viele initialisierte Klassen im Voraus in globale (shared) Speicherbereiche zu laden, um den Memory Footprint so gering wie möglich zu halten.

Tatsächlich lädt Zygote eine mehr oder weniger geschickt gewählte Mischung von Java-Klassen im Voraus in einen globalen Speicherbereich, damit mehrere Applikation diese Klassen gemeinsam benutzen können, ohne selber lokal Platz dafür zu verbrauchen, und um das Starten der eigentlichen Android-Applikationen zu beschleunigen.

Die erwähnte Mischung von Java-Klassen soll nicht nur für möglichst viele Anwendungen nützlich sein, sondern auch während der Lebensdauer der Applikationen beinahe unverändert bleiben. Um dieses Ziel zu erreichen, bildet Zygote zwei Speicherbereiche: Einen Shared-Dirty-Bereich (aber read only) für das Speichern der Dex-Strukturen und einen ebenfalls Shared-Dirty-Heap-Bereich, worin die Klassenobjekte realisiert werden und allen anderen Apps zur Verfügung gestellt werden. Dieser Zygote-Heap soll möglichst selten verändert werden. Wenn eine App ein Klassenobjekt vom Zygote-Heap referenziert, wird dieses Objekt in den eigenen (private dirty) Heap kopiert (copy on write). Ein dritter Bereich (shared clean) beinhaltet alle Dex-Dateien der so genannten Kernbibliotheken, die in `core.jar` definiert worden sind.

Wenn eine App Pakete aus diesem Bereich benötigt, werden die beiden bereits diskutierten Shared-Dirty-Speicherbereiche entsprechend erweitert. Die Abbildung ❹ zeigt noch einmal die Zusammenhänge zwischen den globalen Speicherbereichen von Zygote und denjenigen einer beliebigen Android-Anwendung.

Garbage Collection

Das Zusammenspiel zwischen Shared-Speicherbereichen von Zygote und Android- Applikationen setzt der Funktionalität eines

IMVS FOKUS REPORT

Dieser Artikel ist eine überarbeitete und aktualisierte Version eines Aufsatzes des gleichen Authors, der ursprünglich im IMVS Fokus Report 2009 der Fachochschule Nordwestschweiz erschien.

Garbage Collectors (GC) einige Grenzen. Dazu ist wichtig zu bemerken, dass der Dalvik GC ein gewöhnliches Mark-and-Sweep-Verfahren benutzt. Die Markierungsbits (die Information, ob Objekte noch am Leben sind) können dabei zusammen mit den Objekten im Heap oder in einem getrennten Speicherbereich aufbewahrt werden. Die Verwendung eines getrennten Speicherbereichs ist für Dalvik die einzige Lösung, welche der Heap-Verwaltung in der komplexen Symbiose zwischen Zygote und Android-Applikation gerecht werden kann.

Wie erwähnt, sollen Daten auf dem Zygote-Heap selten verändert werden, da sie für möglichst viele verschiedene Applikationen gelten sollen. Daher wird vorzugsweise eine externe (globale) Struktur für die Aufbewahrung der Markierungsbits eingesetzt. Sie ermöglicht auch die Unterscheidung zwischen allgemeinen Shared-Objekten und Objekten, die die Applikation selber instanziert hat. Der GC soll nur im Heap der Applikation schalten und walten: Objekte im Shared-Bereich werden (wenn überhaupt) nur von einem speziellen GC von Zygote eingesammelt und eventuell gelöscht, nachdem alle GCs der Applikationen nach einem vollständigem Sweep-Vorgang gestoppt wurden (Abbildung 4).

Das Android Linux benutzt die Funktionen `ashmem_create_region(...)` und `ashmem_set-prot_region(...)`, um die Speicherbereiche für die Markierungsbits als anonyme Shared-Speicheregionen zwischen Prozessen zu definieren. Man beachte, dass diese Bereiche von Kernfunktionen des Linux-Systems gelöscht werden können. Dies ist freilich notwendig, damit der GC von Zygote seine Arbeit verrichten kann.

Dalvik JIT Compiler

Dank der grossen Effizienz der Dalvik VM, wird durchschnittlich nur etwa 1/3 der Ausführungszeit einer Android-Applikation im Interpreter verputzt. Im Klartext bedeutet dies, dass 2/3 der Zeit bereits ohne JIT ausschliesslich durch die Bearbeitung von ARM-Maschinenbefehlen ohne Intervention des Interpreters verbracht werden [8]. Trotzdem liegt wohl der wichtigste Grund zur Einführung eines JIT-Compilers in der relativen Langsamkeit des ARM-Prozessors – insbesonders für Apps, die viele graphischen Resourcen beanspruchen (Spiele).

Grundsätzlich werden JIT-Compiler nach zwei Kriterien klassifiziert: (i) Wann soll der JIT-Compiler aktiv werden? (ii) Auf welche Programmeinheit soll er wirken (Granularität)?

Der Dalvik-JIT nimmt die meistbenutzte Spur von Dalvik-VM-Opcodes innerhalb einer Methode als kleinste Einheit für den JIT-Prozess (trace level granularity). Diese Wahl garantiert einen kleinen Speicherplatzbedarf und ermöglicht trotzem, einige einfachen lokalen Optimierungen einzuschalten.

Die erste Frage ist auch damit implizit beantwortet, weil die Trace-Level-Granularität nur zur Zeit der Dalvik-Opcode-Interpretation funktionieren kann. Als kurze Traces sehr geeignet sind zum Beispiel Schleifen, auf die übrigens knifflige Optimierungen wie Invariant Code Motion, Induction Variable Optimization und Register Promotion ausgelebt werden können [8]. Die optimierten Spuren von Maschinenbefehlen werden dann in den translation Cache zwischengespeichert, aus dem der Dalvik-Interpreter nach Bedarf dem Prozessor direkt die weitere Bearbeitung überlässt.

Fazit

Dieser Übersichtsartikel zeigte einige der Hindernisse auf, die das Google-Team überwinden musste, um die volle Kompatibilität mit dem Java SDK 1.5 zu bewahren ohne jedoch die Sun-Spezifikation der JVM einzuhalten. Dieser Bruch ist Google nicht nur sehr gut gelungen, sondern hat auch neue Ideen in der Java-Community ausgelöst, um die Sun 1-Byte-JVM endlich an die Realitäten der heutigen Hardware anzupassen.

Den heimlichen Wunsch des Google-Entwicklungsteams, dass die Dalvik-VM so gut arbeite, dass sie keine Just-In-Time-Kompilierung benötige, hat die Android-Praxis klar wiederlegt. Google hat diesen Gedanken deshalb ab Android 2.2 fallengelassen und zuerst die Android Native Libraries (die eine bessere Einbindung von C-Programmen in Android-Java-Programme erlaubt) in einem NDK gebündelt. Zudem haben die Entwickler die Dalvik-VM um einen JIT-Kompiler erweitert. Somit erreicht die Android-Plattform 2.2 eine technische Reife und Stabilität, die sie für professionelle Entwickler äußerst attraktiv macht. (mhi) ■

INFO

1] Agesen, O., Detlefs, D. Finding References in Java Stacks. OOPSLA97 Workshop on Garbage Collection and Memory Management, October 1997.

[2] The Android Open Source Project, Dalvik Optimization and Verification With dexopt, 2008: http://github.com/android/platform_dalvik/blob/c1b54205471ea7824c87e53e0d9e6d4c30518007/docs/dexopt.html.

[3] The Android Open Source Project, Dalvik Bytecode Verifier Notes, 2008 http://github.com/android/platform_dalvik/blob/c1b54205471ea7824c87e53e0d9e6d4c30518007/docs/verifier.html.

[4] Bernstein, D. Dalvik VM Internals. IO-Google Conference, May 29th, 2008.

[5] Gosling, J. Java Intermediate Bytecodes. ACM SIGPLAN Workshop on Intermediate Representations, 111-118, 1995.

[6] The Android Open Source Project, Dalvik DEX Format 2008: http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=docs/dex-format.html

[7] Eager, M. J.: Introduction to the DWARF Debugging Format, (2007) http://dwarfstd.org

[8] Buzbee, B., Cheng, B.; A JIT Compiler for Android's Dalvik VM. Google I/O May 27th, 2010.