Einblick in die Dalvik Virtual Machine

Die Dalvik Virtual Machine ist der eigentliche Motor des Android-Systems. Neben den Unterschieden gegenüber der Standard Java Virtual Machine werden wir drei Themen genauer betrachten: (i) Die Rolle des Programms Zygote beim Starten der Dalvik Virtual Machine; (ii) die Rolle der Registerstruktur der Dalvik Virtual Machine auf die allgemeine Performanz des Android-Programmierungsmodells und (iii) der Einfluss des neuen Class-File-Formats von Dalvik auf die Effizienz der virtuellen Maschine.

Carlo U. Nicola | carlo.nicola@fhnw.ch

Android ist ein Betriebssystem und auch eine Software-Plattform für mobile Geräte wie Smartphones, Mobiltelefone und Netbooks. Es basiert auf dem Linux-Kernel 2.6 und wird von der Open Handset Alliance entwickelt, wobei ein grosser Teil der Software frei und quelloffen ist [Wiki]. Die Dalvik Virtual Machine (DVM) ist der eigentliche Motor des Android-Systems. Sie ist von Anfang an für anspruchsvolle Java-Applikationen (Version 5.0 bzw. 1.5) konzipiert worden, mit dem Ziel, diese Anwendungen auf extrem bescheidenen Hardware-Ressourcen möglichst effizient laufen zu lassen. Die bescheidenen Ressourcen lassen sich wie folgt zusammenfassen: eine CPU mit maximaler Taktfrequenz von 500 MHz; 64 Megabyte Speicher, von denen nur 20 Megabyte für Applikationen zur Verfügung stehen; ein Linux Betriebssystem ohne *swap space* 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, obschon Java Bytecode nicht direkt auf der DVM lauffähig ist. Java Bytecode wird für die Dalvik Virtual Machine speziell kompiliert und in einer Dex-Datei (Dateinamenserweiterung .dex) abgelegt.

In den folgenden Abschnitten werden wir den Fragen nachgehen, wie sich die die Dalvik Virtual Machine von der Standard Java Virtual Machine unterscheidet, welche Rolle die Registerstruktur der DVM auf die allgemeine Performanz des Android-Programmierungsmodells hat und wie sich das neue Bytecode-Dateiformat (Dex-Format) auf die Effizienz der DVM auswirkt.

Starten von Android-Anwendungen

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:

1. Der *init.rc*-Prozess startet verschiedene *daemons*.
2. Anschliessend wird der Service Manager und Zygote aufgerufen. Zygote wird im weiteren Systemablauf dafür sorgen, dass jede Android-Anwendung als normaler Linux-Prozess über den bekannten *fork*-Mechanismus gestartet wird.
3. 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 1 zeigt diese drei Schritte schematisch und schlägt auch die Brücke zum Start der eigentlichen Android-Anwendungen. Wie bereits erwähnt, sorgt der Prozess Zygote dafür, dass eine Android-Anwendung als normaler Linux-Prozess mittels *fork()* gestartet wird. Gleichzeitig wird zu jedem Prozess eine DVM mit eigenem Heap und Stack gestartet und die notwendigen Bindungen zu den Java *Core Libraries* hergestellt. 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 können. Die Auswirkungen dieses Designentscheids auf die Arbeit des *Garbage Collectors* werden wir später im Abschnitt „Shared Memory und Garbage Collection" behandeln. Schliesslich soll an dieser Stelle noch bemerkt werden, dass jeder Linux-Prozess die *bionic library* (eine ganz gewöhnliche C-Standard-Library) mitschleppt.

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;

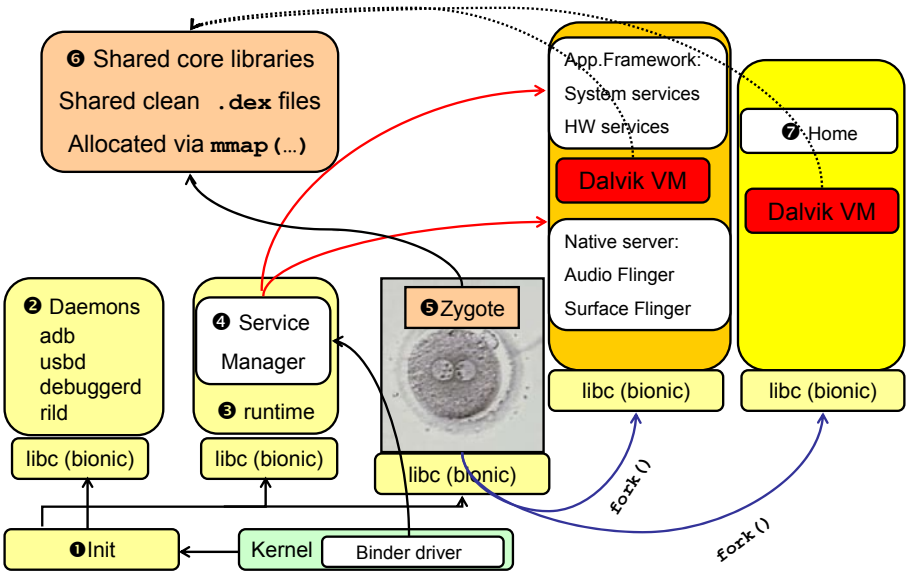

Abbildung 1: Der init.rc-Prozess in Android. Glossar der Abkürzungen: adbd: Android debugger; usbd: USB Schnittstelle; rild: radio interface layer; debugerd: debug system. Weitere daemons werden gestartet, sind aber in der Abbildung nicht dargestellt. Audio und Surface Flinger: eine Google Bezeichnung für MP3- bzw. graphische Dienste (modifiziert aus [Bern08]). Die eingekreisten Ziffern geben die zeitliche Reihenfolge der Prozesse an.

- *execute:* die durch den Bytecode dargestellte Funktion ausführen. Diese Phase beansprucht freilich am meisten Zeit.

In dieser Implementierung (siehe Listing 1) 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[1].

Register- vs. Stack-Maschine

Die DVM und JVM sind grob gesehen sehr ähnlich, wobei sie sich aber in zwei Hinsichten grundlegend unterscheiden: Die DVM ist nicht als Stack-Maschine, sondern als Register-Maschine realisiert, und die Längen der Opcodes betragen 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 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 *„prima facie"* 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 bekanntlich 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 auf komplizierte *Just In Time* (JIT) 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- bzw. Hauptspeicher abgebildet werden müssen. Java Methoden, die mehr als 16 Argumente und Parameter benötigen, sind jedoch extrem selten, so dass üblicherweise fast alle Argumente und Parameter bei einem Methodenaufruf in den 16 16-Bit Registern des ARM-Prozessors Platz finden.

Interessanter wird, 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

1 Anmerkung für Hackers: Das Interpreter-Programm soll mit gcc (ab v. 4) kompiliert werden.

```
#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: Einfache Struktur eines C-Interpreter-Programmes. 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.

Register VM müssen die beiden Operanden der Addition in Register kopiert werden. Bei der DVM 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.

Java VM 1.6	Register VM	Dalvik VM (16 Bit Opcade)
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	
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.

Wie das kleine Beispiel in Tabelle 1 schön zeigt, wird die theoretisch mögliche Bytecode-Einsparung einer Register VM von der DVM üblicherweise nicht erreicht. 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 laden kann 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 Verschenkung von Ressourcen. 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 kleiner Vorteil einer Register VM ist, dass die typische Nebenbedingung einer Stack VM ersatzlos wegfällt, nämlich dass der Opcode Stack am Ende eines Methodenaufrufes im gleichen Zustand wie am Anfang sein soll.

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 [Bern08].

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 [Gos95, Ag97]. Man hat dies ab Java 1.5 korrigiert. Die DVM 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,

```
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.

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 [AP08a] optimiert:

- *Alignement*: Die Opcodes und die Daten müssen nach 16 Bit ausgerichtet sein. Spezielle Ausrichtungen wie z.B. 64 Bit werden explizit in Assembler-Code vorgenommen. *dexopt* führt zusätzliche *nop* (*no operation*) Instruktionen 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 herunter lädt, welche von allen zukünftigen Instanzen der DVM benutzt werden.
- *Effizientere Opcodes:* Einzelne Opcodes können durch effizientere ersetzt werden, wie zum Beispiel *invoke-virtual-quick*, welcher besonders effizient mit der *vtable* des Zielobjektes arbeitet.

Die so optimierten Dex-Dateien werden mit der Dateinamenserweiterung *.odex* bezeichnet.

Das 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 VM ≤ 1.4

```
0  aload_0
1  invokevirtual #6
4  jsr 14
7  return
8  astore_1
9  jsr 14
12 aload_1
13 athrow
14 astore_2
15 aload_0
16 invokevirtual #5
19 ret 2
```

Java VM ≥ 1.5

```
0  aload_0
1  invokespecial #17
4  goto 14 (+10)
7  astore_1
8  aload_0
9  invokespecial #20
12 aload_1
13 athrow
14 aload_0
15 invokespecial #20
18 return
```

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

Segment Name	Format	Beschreibung
header	header_items	Neben der Versionsnummer, der obligaten magischen Zahl und gewisser Sicherheitsmerkmale findet man hier Angaben über die Grösse der verschiedenen Segmente und deren Startadressen.
string_ids	string_id_item[]	Enthält alle Strings, die von dieser Datei entweder als interne Bezeichner (z.B. von Typen) oder als konstante String-Objekte benutzt werden.
type_ids	type_id_item[]	Bezeichner für alle Typen (Klassen, Arrays, Elementartypen) die in dieser Datei referenziert werden, unabhängig davon, ob sie in dieser Datei definiert werden oder nicht.
proto_ids	proto_id_item[]	Signaturen aller Methoden, die in den verschiedenen Klassen referenziert sind. Beispiel: wrapItUp()V
fields_ids	fields_id_item[]	Bezeichner für alle Instanzvariablen, die in dieser Datei referenziert und benutzt werden, unabhängig davon, ob sie in dieser Datei definiert werden oder nicht.
methods_ids	methods _id_ item[]	Bezeichner für alle Methoden, die in dieser Datei referenziert und benutzt werden, unabhängig davon, ob sie in dieser Datei definiert werden oder nicht.
class_ids	class _id_item[]	Liste der Klassendefinitionen. Die Liste muss die Basisklasse und die implementierten Interfaces vor der Klasse, welche diese benutzt, auflisten.
data	ubyte[]	Datenbereich, worin die Informationen für alle oben definierten Tabellen gespeichert sind.
link_data	ubyte[]	Datenbereich für die statische Bindung von weiteren Dateien.

Tabelle 2: Die verschiedenen Segmente des Dex-Formats

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. Die Tabelle 2 fasst diese Segmente zusammen und beschreibt sie kurz. In Abbildung 2 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*.

Während die einzelnen Bytecode-Dateien (.class) eines für die Standard JVM kompilierten Java-Programmes üblicherweise in einem (komprimierten) Java-Archiv (.jar) zusammengefasst werden, so braucht es für die DVM kein zusätzliches Archiv-Format, da das Dex-Format sowohl einzelne Klassen als auch 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 (Abb. 3). 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 [DWARF07]. Zum Beispiel wird die in dieser Spezifikation definierte LEB128-Codierung (eine variable Bit-Codierung für die Darstellung von ganzen Zahlen) stark benutzt, um so genannte *encoded-value* Elemente der Dex-Datei kompakt darzustellen.

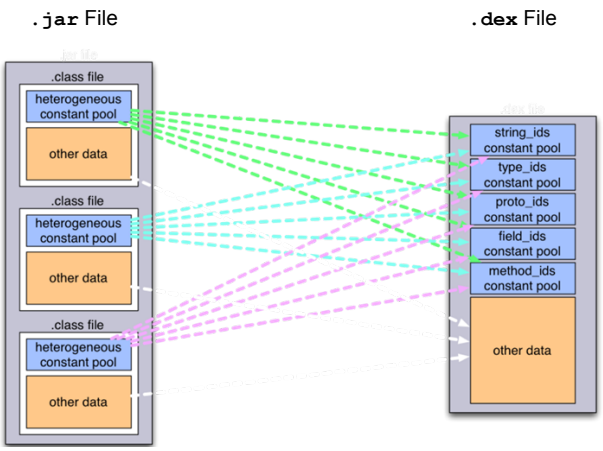

Abbildung 2: Beispiel eines konkreten Header-Segmentes einer Dex-Datei

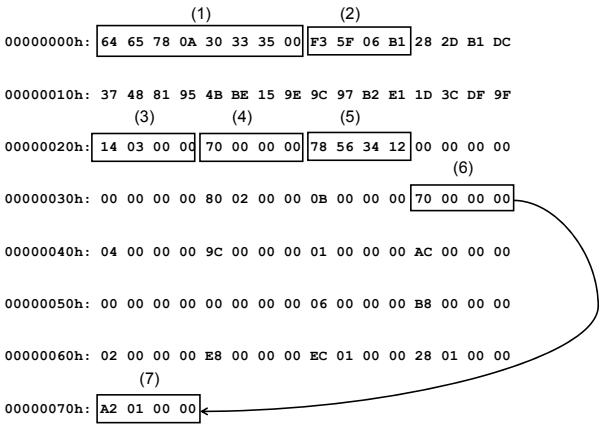

Abbildung 3: Das Dex-Format fasst alle heterogenen constant_ pool Bereiche zusammen, die eine jar-Datei für jede mitgeschleppte Klasse neu definiert (aus [Bern08]).

```
public class MemInfo extends Activity {
    private TextView displayResults;
    private Debug.MemoryInfo dMemInfo;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        dMemInfo = new Debug.MemoryInfo();

        // Set output for results
        displayResults = (TextView) findViewById(R.id.results);

        // Add button listener. Watch for button clicks.
        Button getButton = (Button) findViewById(R.id.get);
        getButton.setOnClickListener(getListener);

        Button setButton = (Button) findViewById(R.id.set);
        setButton.setOnClickListener(new OnClickListener(){
            @Override
            public void onClick(View v) {
                System.getProperties().put(this.getClass().
                    getName(),"Android");
            }
        });
    }
    private OnClickListener getListener = new OnClickListener() {
        public void onClick(View v) {
            Debug.getMemoryInfo(dMemInfo);
            displayResults.setText(
                «Shared/Private dirty/clean Memory\n\n»
                + «Dalvik proportional set size [kB] «
                + Integer.toString(dMemInfo.dalvikPss) + "\n"
                + "Dalvik private dirty [kB] "
                + Integer.toString(dMemInfo.dalvikPrivateDirty) + "\n"
                + "Dalvik shared dirty [kB] "
                + Integer.toString(dMemInfo.dalvikSharedDirty) + "\n");
        }
    };
}
```

Listing 4: Android-Applikation, die laufend die Grösse ihrer verschiedenen Speicherbereiche darstellt. Dies zeigt konkret das Zusammenspiel zwischen Zygote shared Speicherbereichen und denen, die die Applikation lokal definiert hat (private dirty).

Verifizierung und Optimierung von Dex-Dateien

Die Überprüfung (*verify*) der Dex-Dateien weicht sehr stark vom Standard Java-Verfahren ab. Der Grund liegt darin, dass viele Klassen und Applikationen (so genannte *boostrap classes*) von Zygote ins *dalvik-cache* Verzeichnis ohne Beteiligung eines *Class Loaders* mittels Memory-Mapping geladen werden. 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 werden die Dex-Dateien mit *dexopt* optimiert und erhalten dadurch die neue Dateierweiterung *.odex.* Das Ganze dient auch dazu, die Android-Applikationen schneller starten zu können.

Zygote muss während der Vorverifizierung einige vernünftige Annahmen treffen, um nicht später mit dem *Class Loader* der DVM in Konflikt zu geraten [AP08b]. Als Beispiel gehen wir von ei ner Applikation *MyApp.apk*[2] 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 bzw. 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* z. B. eine neue Version der *core.jar* Klassen (vielleicht mit einem Virus-Programm bestückt!) laden kann.

2 Android Package Files verwenden die Dateierweiterung .apk. Es handelt sich dabei um ZIP-komprimierte Ordner analog zu Java Archives (.jar). Die core libraries werden im Verzeichnis dalvik-cache als system@framework@core.jar@classes.dex abgelegt.

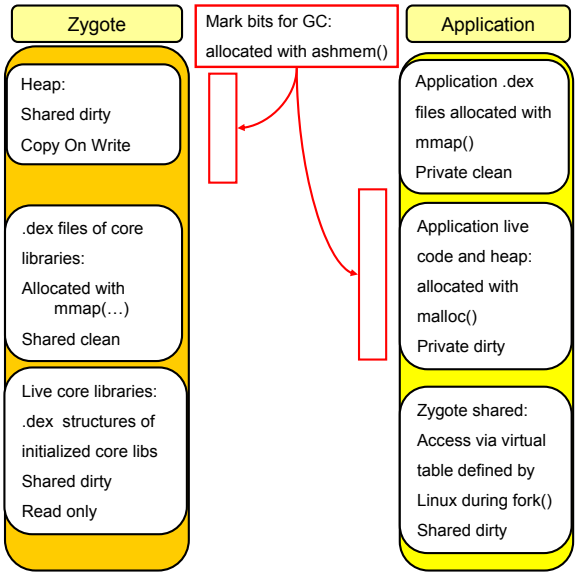

Abbildung 4: Zusammenhang zwischen den globalen (shared) Bereichen von Zygote und denjenigen einer beliebigen Android-Applikation.

Neben der Lösung des eben beschriebenen Problems soll *dexopt* weitere Checks durchführen [AP08b]. 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 weitere 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 solcher *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 [AP08a]:

- Optimierungen, welche Instruktionen und Daten auf 16-Bit ausrichten. Oft bedeutet dies das 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 (z.B. Hash-Werte) im Voraus berechnen, um diese Berechnungen in der DVM zu vermeiden.

Shared Memory und Garbage Collection

Im Allgemeinen klassifiziert jedes Betriebssystem den Speicher in vier Qualitätsklassen, die sowohl nach der Art seiner Allokation als auch nach der Art seiner 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[3].

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* aller Applikationen 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: Ein *shared dirty* (aber *read only*) für das Speichern der Dex-Strukturen und ein ebenfalls *shared dirty* Heap-Bereich, worin die Klassenobjekte realisiert werden und allen anderen Applikationen zur Verfügung gestellt werden. Dieser Zygote-Heap soll aber möglichst selten verändert werden. Wenn eine Applikation ein Klassenobjekt vom Zygote-Heap referenziert,

3 Im Listing 4 findet man eine Android-Applikation, die die Zunahme bzw. die Abnahme der verschiedenen Speicherbereiche darstellt.

wird dieses Objekt in den eigenen (*private dirty*) Heap kopiert (*copy on write*). Ein dritter Bereich (*shared clean*) beinhaltet alle Dex-Dateien der sogenannten Kernbibliotheken, die in *core.jar* definiert worden sind. Wenn eine Applikation Pakete aus diesem Bereich benötigt, werden die beiden bereits diskutierten *shared dirty* Speicherbereiche entsprechend erweitert. Die Abbildung 4 zeigt noch einmal die Zusammenhänge zwischen den globalen Speicherbereichen von Zygote und denjenigen einer beliebigen Android-Anwendung.

Das Zusammenspiel zwischen *shared* Speicherbereichen von Zygote und Android-Applikationen setzt der Funktionalität eines *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 beim *mark and sweep* Verfahren entweder zusammen mit den Objekten im Heap oder in einem vom Heap 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 bereits 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* Berich werden (wenn überhaupt) nur von einem speziellen GC von Zygote eingesammelt und eventuell gelöscht, nachdem alle GCs der Applikationen nach einem vollständigem *sweep* gestoppt wurden (siehe auch Abb. 4).

Das Android Linux Betriebssystem 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 Android Linux Kernfunktionen gelöscht werden können. Dies ist freilich notwendig, damit der GC von Zygote seine Arbeit verrichten kann.

Zusammenfassung

In diesem Übersichtsartikel haben wir einige der Hindernisse aufgezeigt, 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 Tabubruch 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 HW anzupassen. Die Hauptannahme des Google-Entwicklungsteams, dass die Dalvik VM (im Gegensatz zur Sun HotSpot JVM) keine *Just In Time* (JIT) Kompilierung benötigt, hat sich dagegen besonders im mobilen Bereich als falsch erwiesen. Zwei Entwicklungen zeugen von diesem Sinneswandel bei Google: Einerseits die Freigabe der Android Native Libraries (die eine bessere Einbindung von C-Programmen in Android Java-Programme erlaubt) und anderseits die Mitteilung, dass wirklich *really soon* die Dalvik VM mit einem JIT-Kompiler erweitert wird.

Referenzen

[Ag97] Agesen, O., Detlefs, D. Finding References in Java Stacks. OOPSLA97 Workshop on Garbage Collection and Memory Management, October 1997.

[AP08a] The Android Open Source Project, Dalvik Optimization and Verification With dexopt, 2008: http://github.com/android/platform_dalvik/blob/c1b54205471ea7824c87e53e0d9e6d4c30518007/docs/dexopt.html.

[AP08b] The Android Open Source Project, Dalvik Bytecode Verifier Notes, 2008 http://github.com/android/platform_dalvik/blob/c1b54205471ea7824c87e53e0d9e6d4c30518007/docs/verifier.html.

[Bern08] Bernstein, D. Dalvik VM Internals. IO-Google Conference, May 29th, 2008.

[DWARF07] Eager, M. J.: Introduction to the DWARF Debugging Format, (2007)

http://dwarfstd.org/

[Gos95] Gosling, J. Java Intermediate Bytecodes. ACM SIGPLAN Workshop on Intermediate Representations, 111-118, 1995.

[Wiki] Android (Betriebssysstem): http://de.wikipedia.org/wiki/Android_(Betriebssystem).