The Black Magic Behind ART

Anupam Singh

Sr Android Dev
Urbanclap

Co-host
Androidiots

Index

- Let's talk basics first
- ART, DVM blah bla
- Evil multidexing
- Deep Dive into current scheme of ART things
- Quick walk through over history of ART
- List of things we cannot cover today

please note: 30 mins are not enough

What is an APK ?

- Android Application Package
  - A Package of ?
    - Manifest
    - Assets
    - Resources
    - Classes.dex
- AAPT allows us to view, update,create and apk.

Lets Compile

- Java byte code and JVM
    - javac converts java source code (class, interface, enum) to .class
    - These .class contains java bytecode which can be executed on JVM
- Why Dex ?
    - All android devices with jvm to run java byte code ? : NO
    - DVM : for handheld devices, with processing power, battery,memory constraints
    - . dex is optimized for minimal memory footprint

Why Dex?

- Dx tool compiles all .class files into a single classes.dex file
    - Classes.dex : bytecode of all source code and libraries
- Dalvik uses 16 bit instruction set vs 8 bit stack based instruction set of JVM
    - Reduces instruction count and interpreter speed
- Smaller memory footprint : .dex files are smaller than .jar
- Dalvik VM is replaced by ART from Android Lollipop
    - Art still uses the .dex format
- Android Studio has switched to D8 compiler from DX

ART(DVM) vs JVM

- Android : Application specific implementation of linux
  - Every app is a seperate user ,so every app needs a separate process hence every app needs a seperate VM.
    - Dalvik is optimized for running multiple VM instances with as much shared memory as possible
    - Shared memory does not means shared VM
  - Every byte code boils down as an instruction to CPU with classification
- ART is Register based vs JVM is stack based
  - Stack based machines use instructions to load and manipulate data on stack hence use more instructions compared to register based machines
  - Register based machines need to encode the data (stored in registers) in its instruction hence instruction size is larger , that's ***why Dalvik needs 16 bit instruction set vs 8 bit instruction set used in JVM***
  - Stack based design makes a few assumptions about the target hardware(register, cpu) hence its easier to implement on wide variety of hardware
  - So which one is better ?: That"s a talk for another day

ART vs DVM

- ART was introduced in 4.4 but was not enabled by default, it replaced dalvik in 5.0
- Two main features
  - AOT vs JIT
    - AOT : apps are compiled into native code and stored on the devices and it runs native code not bytecode
    - JIT : compiles bytecode to native code on the fly adding both latency and memory pressure
  - Improved garbage collection :
    - Concurrent mark and sweep : One pause compared to two
    - Generational GC : effective garbage collection
    - heap compaction and less floating garbage

But you said .dex is optimised

- *Optimisations does not stop on .dex creation*
  - In Dalvik Dexopt runs in .dex files and gives us an .odex (Optimized dex) file
    - Similar to .dex with optimisez instruction set
  - In ART dex2Oat takes a .dex and compile it to native code
    - Result is an elf file that is executed natively
      - So instead of bytecode which is interpreted by VM it has native code which can be executed by processor

Multidexing : A necessary Evil

- Big app ???
  - And you gave multidex enabled
  - Does TransformException, ClassNotFoundException, NoClassDefFound sounds familiar ?
- But why ??
  - Dalvik executables have 16 bit instruction set,this imposes a limit on method count on a single DEX file including *android framework methods, library methods and methods in our code* which lead to **64K reference limit.**

Internals of multidexing

- Android build tools generates the PRIMARY dex files and supporting dex files (classes1.dex, classes2.dex)
- All these dex files are bundled in apk
- At runtime Multidex Apis use special class loaders to search for all available dex files for the specific method instead of just the PRIMARY dex
- All direct and transitive dependencies must go in the Application Start
    - Else app crashes with NoClassDefFound error
    - And this does not always happens
        - if the code uses introspection or invocation of Java methods from native code, then those classes might not be recognized as required in the primary DEX file.
    - Solution : manually provide class path in seperate file and set ***multidexKeepFile*** property, and set ***multidexKeepProguard*** property fror proguard

Enough with the ancient history and jargons show me what's happening now and some images please

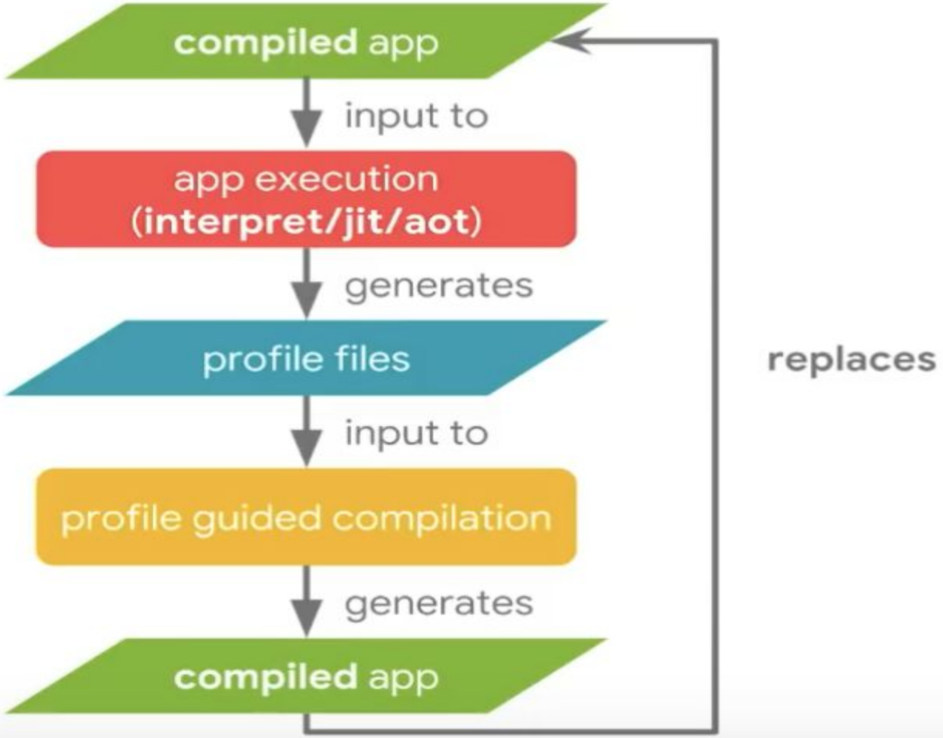
ART:

Android Runtime

Runs Android framework and applications
Interpreter, JIT, profile based compiler

Manages application memory
Memory allocator, Garbage collector

Application Lifecycle since Nougat



Profile-guided compilation
Idle-time optimizations

How is profile guided compilation helping application startup?

- Compiles startup methods
- Lays out dex and compiled code
- Generates an application image
- Pre-initializes classes
- Doesn't compile "cold" code (~80%)

Application images: Generation

Application startup

Improving application images

String interning was observed to be **expensive** during startup. This operation is used for string literals.

Idea

Resolve string literals ahead-of-time.

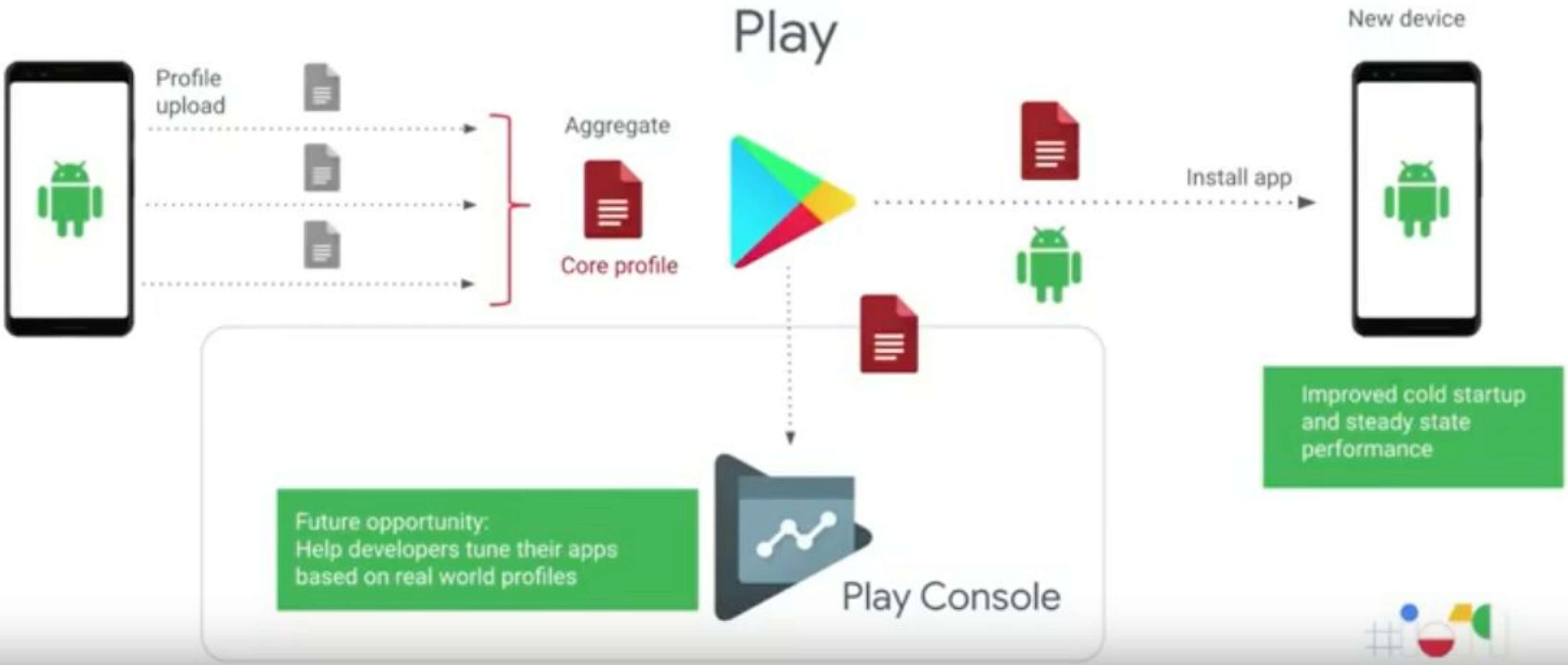
Improving startup directly after install

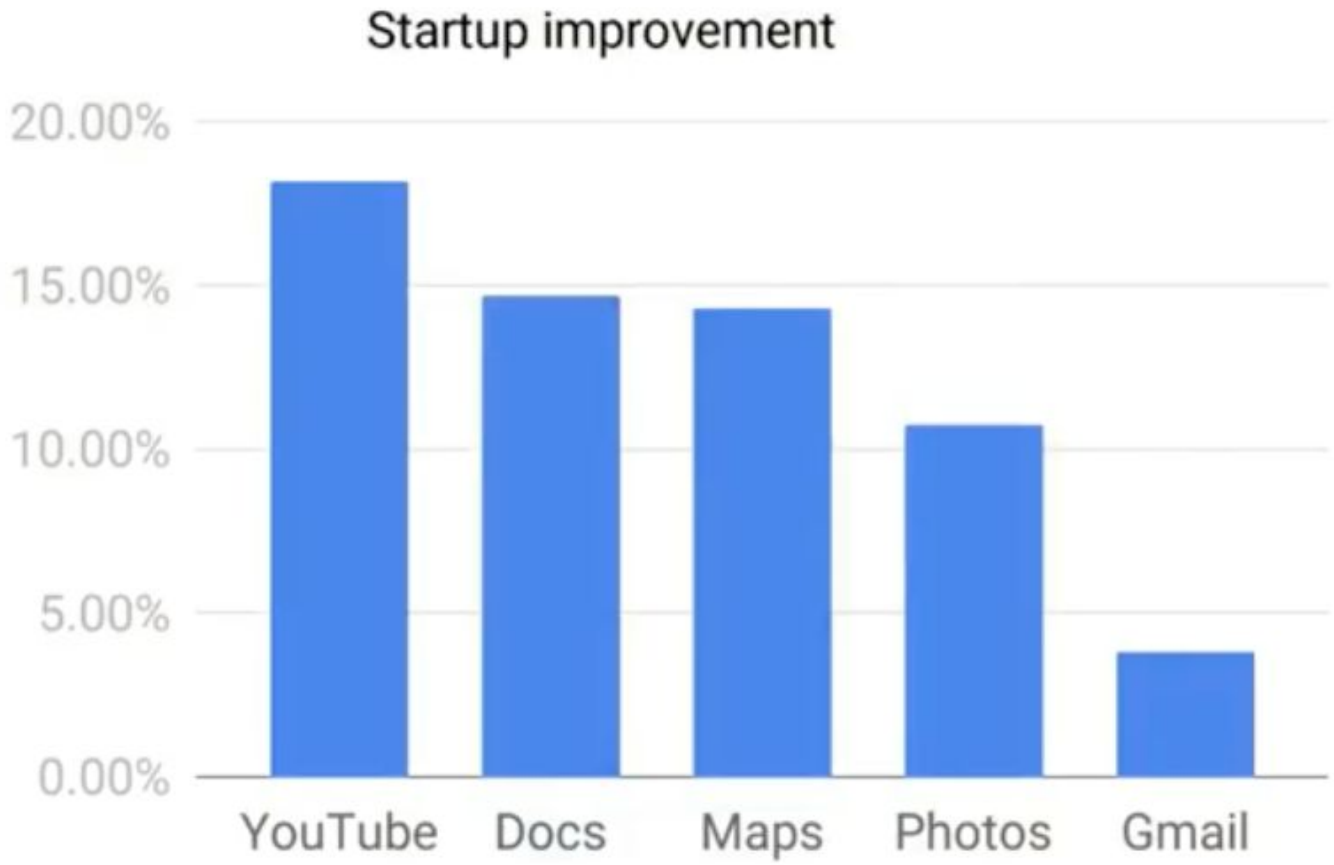
Profiles in the cloud are now downloaded alongside application installs for Android P+ devices.

These profiles improve app startup right after install by **~15%.**

Process

Existing devices





Startup improvement
20.00%
15.00%
10.00%
5.00%
0.00%
YouTube
Docs
Maps
Photos
Gmail

Application startup

Pre-forking application processes

Forks processes from the Zygote off the critical path

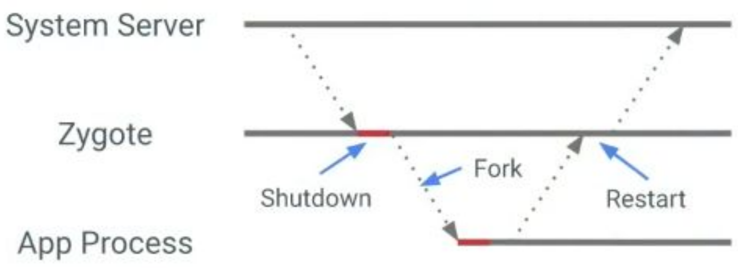
Adds a pool of unspecialized app processes

Average speedup of ~5 ms on Pixel 2

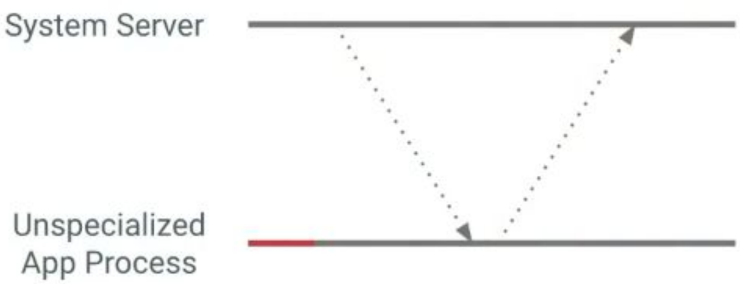
Idea

Anticipate the start of an application

Pre-forking application processes



Previous versions of Android



Android Q

Application startup

Introspection

New tools: StartupAnalyzerKt †

Use `reportFullyDrawn()`‡ to identify application startup endpoint

Idea

Deep analysis of startup events

ART History

Major Android Runtime Evolutions

Dalvik up to K

- Interpreter
- Trace-based JIT
- Conservative, stop the world GC

ART in L

- Ahead-of-Time Compilation
- Precise, generational GC

ART in N and O

- Profile-guided compilation
- Hybrid JIT/AOT execution
- Concurrent GC (in O)

ART Optimizations From Dalvik

Performance

- SSA Compiler
- Dynamic optimizations
- Thread-local buffers

Jank

- AOT framework code
- Generational GC
- Profile guided compilation

Application Startup

- **Zygote**
- AOT framework code
- Profile guided compilation

Battery

- AOT Compiled code
- Profile guided compilation

Disk space

- Uncompressed dex
- Compact dex
- Profile-guided compilation

RAM

- **Zygote**
- Concurrent GC
- Compact dex

Boot times

- AOT framework code
- No "optimizing apps"

Install times

- JIT on first use
- Uncompressed dex

ART Optimizations in Lollipop

Performance

- **SSA Compiler**
- Dynamic optimizations
- Thread-local buffers

Jank

- **AOT framework code**
- **Generational GC**
- Profile guided compilation

Application Startup

- Zygote
- **AOT framework code**
- Profile guided compilation

Battery

- **AOT Compiled code**
- Profile guided compilation

Disk space

- Uncompressed dex
- Compact dex
- Profile-guided compilation

RAM

- Zygote
- Concurrent GC
- Compact dex

Boot times

- **AOT framework code**
- No "optimizing apps"

Install times

- JIT on first use
- Uncompressed dex

ART Optimizations in Nougat / Oreo

Performance
- SSA Compiler
- **Dynamic optimizations**
- **Thread-local buffers**

Jank
- AOT framework code
- Generational GC
- **Profile guided compilation**

Application Startup
- Zygote
- AOT framework code
- **Profile guided compilation**

Battery
- AOT Compiled code
- **Profile guided compilation**

Disk space
- Uncompressed dex
- Compact dex
- **Profile-guided compilation**

RAM
- Zygote
- **Concurrent GC**
- Compact dex

Boot times
- Zygote
- AOT framework code
- **No “optimizing apps”**

Install times
- **JIT on first use**
- Uncompressed dex

ART Optimizations in Pie

Performance

- SSA Compiler
- Dynamic optimizations
- Thread-local buffers

Jank

- AOT framework code
- Generational GC
- Profile guided compilation

Application Startup

- Zygote
- AOT framework code
- Profile guided compilation

Battery

- AOT Compiled code
- Profile guided compilation

Disk space

- **Uncompressed dex**
- **Compact dex**
- Profile-guided compilation

RAM

- Zygote
- Concurrent GC
- **Compact dex**

Boot times

- AOT framework code
- No “optimizing apps”

Install times

- JIT on first use
- **Uncompressed dex**

Things we are not covering today : Performance

- Kotlin Optimisations
  - Inlining
  - Side Effect analysis
  - Code sinking
  - Code Layout
- Compiler Optimisations
  - Constant folding
  - Instruction simplifier
  - Bound Check elimination

References

- https://www.youtube.com/watch?v=vU7Rhcl9x5o
- https://medium.com/androidiots/androidiots-podcast-5-the-black-magic-behind-android-runtime-part-i-1803dcebf1db
- https://medium.com/androidiots/androidiots-podcast-6-the-black-magic-behind-android-runtime-part-2-9390f751eef0
- https://www.youtube.com/watch?v=1uLzSXWWfDg

Thanks

Google , romi chandra and amanjeet.