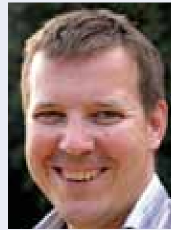
# Så byggs nya Android

*Körmiljön i Android – det populära ansiktet till många inbyggda system – får en rejäl uppgradering.*



**Av Robert Niemi, Qodes AB**

**Robert Niemi** är Androidexpert på konsultföretaget Qodes AB, vilket är en del av QGroup AB. Han har jobbat med Androidplattformen sedan källkoden släpptes 2008 och har utbildat närmare tusen ingenjörer i Android och annan hårdvarunära programmering. Robert är en mycket uppskattad mentor och har hjälpt flera företag att ta fram Androidbaserade produkter och göra anpassningar till befintliga system.

ART – Android RunTime – är den nya exekveringsmiljön för appar i Android. Föregångaren – Dalvik – har dragits med flera begränsningar på grund av de tidiga Androidsystemen och nu är det dags för en ersättare.

Arbetet med ART har pågått under ganska lång tid – den första ändringen som syns i källkoden gjordes sommaren 2011, alltså för mer än tre år sedan. Men den blev inte känd utanför utvecklingsteamet förrän den släpptes som beta i Android 4.4 KitKat hösten 2013. I den senaste versionen, Android 5.0 Lollipop, släpps den skarpt.

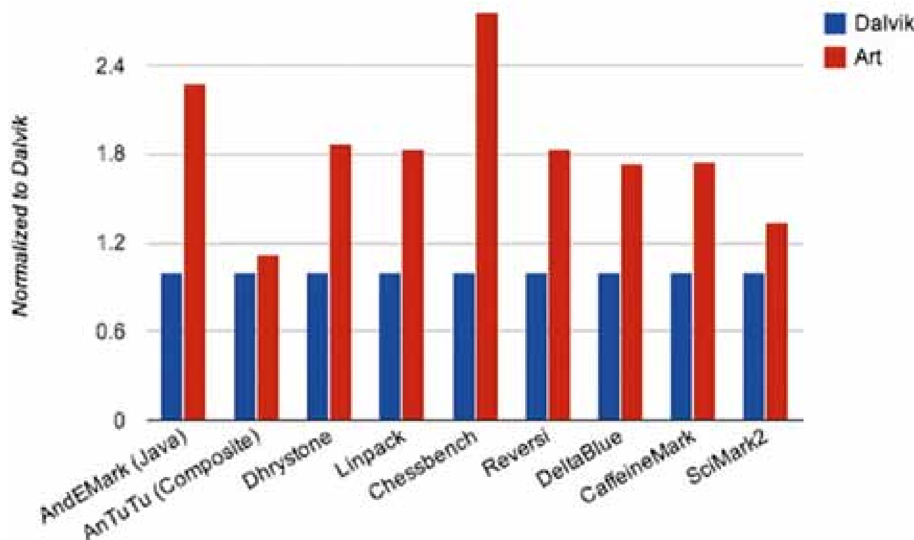
Anledningarna till att man valt en ny exekveringsmiljö är flera.

Till att börja med förbereds koden för körning på ett helt annat sätt än i Dalvik. I Android behandlas koden i flera steg. Det första sker i samband med utvecklandet, där källkoden kompileras med en javakompilator till bytekod och sedan ytterligare en gång till Androids eget format – dexkod. För att göra en korrekt jämförelse mellan Android och andra exekveringsmiljöer är det hanteringen av dexkod man ska titta på.

**FÖLJANDE ÄR DE VANLIGASTE** sätten att exekvera kod:

- Interpreterande. Koden körs instruktion för instruktion. Det klassiska exemplet är programspråket Basic. I de första versionerna av Android användes en interpreterande version av Dalvik. Fördelen är mindre minnesåtgång och mindre lagringsutrymme. Men det går långsammare att köra koden.
- JIT – Just in time-kompilering. Koden kompileras i block i samband med att koden körs. Detta använder mer arbetsminne och eftersom de översatta kodblocken sparas en tid används också mer lagringsutrymme. Men prestandan ökas genom att kodblock finns färdigöversatta till maskinkod när det är dags att köra dem. JIT användes med programspråket Smalltalk redan på 80-talet. Android fick en JIT-kompilator i och med version 2.2.
- AOT – Ahead of Time. Koden kompileras i sin helhet till maskinkod innan den körs första gången. Detta kräver ännu mer lagringsutrymme, men vinsten är prestandaökningar. För språk som exempelvis C och C++ så sker detta när utvecklaren bygger programmet. I Android så sker det vid installationen av en app. AOT är nytt för Android i samband med övergången till ART.

I Googles egen telefon Nexus 5 besegrar Art Dalvik i alla tester.




**FÖR ART SKER SÅLEDES** kompileringen i tre steg: från källkod till javabytekod och vidare till dexkod i samband med utvecklandet, och dexkod till maskinkod i samband med installation i Androidsystemet. Anledningen till att det sista steget körs lokalt i varje enhet är att dexkoden är generell, medan maskinkoden beror på vilken cpu enheten kör. Idag finns Android för Arm, Mips och x86. Varje processorfamilj har dessutom flera varianter.

Lite förenklat kan man alltså säga att exekveringsmiljön har bytt modell allt eftersom Androidenheterna har får mer minne.

Kompileringen hade man förmodligen kunnat lösa i Dalvik, men ytterligare en stor begränsning gjorde att Android-teamet valde att ersätta Dalvik: strukturella problem med hanteringen av dynamiskt allokerat minne i det som kallas heapen.

Garbage collectorn – som städar upp dynamiskt reserverat minne – har också varit med om en hel del förändringar genom åren. I de första versionerna av Android kunde inget av städningen ske parallellt, utan hela appen stannade upp medan städningen kördes. Dessa frekventa stopp i körningen gav en mycket dålig användarupplevelse då appen upplevdes ryckig.

**MED VERSION 2.3 GJORDES** en mycket stor omarbetning av städrutinerna och man införde en så kallad ”concurrent garbage collector”, som gjorde det mesta av städningen parallellt med att appen körde som vanligt. Ryckigheten som orsakades av städningen försvann nu nästan helt. Men heapen behövde fortfarande låsas under en kort tid i samband med att minnet var städat och skulle återlämnas till heapen. Dessa låsningar stoppade fortfarande upp appen i sin helhet något även om det inte alls var så ofta som i tidigare versioner.

# om från grunden

Det fanns dock kvar en typ av låsningar som fortfarande kunde få en app att frysa, och det var om minnet började ta slut samtidigt som appen behövde mer. Då stoppades exekveringen helt och hållet medan städrutinerna försökte hitta minnesblock som gick att frigöra och återanvända.

Båda dessa problem har i princip eliminerats i ART. Under normal körning har mycket av städningen flyttats över till respektive minnesanrop, vilket gör att den specifika städtråden inte behöver köra så ofta.

Minnesallokeraren i ART skiljer dessutom på små och stora objekt. Ett välprövat antagande är att större objekt typiskt innehåller grafik och annan data som är ganska långlivad. Därför allokeras dessa minnesblock från en annan del av heapen än mindre minnesblock, som allokeras och frigörs betydligt mer frekvent. Dessutom är små minnesblock ofta lokala per tråd, vilket man också tagit hänsyn till i ART. Block som allokeras och frigörs i samma tråd kan hanteras helt utan att påverka övriga trådar.

Slutligen har man även lagt in funktionalitet för att flytta minnesblock. Processer som inte körs för tillfället kommer helt enkelt att få sitt minne ommöblerat, för att snabba upp framtida användande. Detta är en av de få delar som programutvecklare måste ta hänsyn till i appar som innehåller både javakod och native-kod – man är tvungen att se till att inte hålla kvar minnesreferenser mellan anrop, för de kanske inte stämmer efteråt.

**FLYTTEN AV BLOCK,** samt den egentliga städningen av frigjorda minnesblock sker så långt det är möjligt medan skärmen är avslagen, så användaren kommer att märka mycket lite av dessa två operationer.

Sättet appar kompileras på, AOT, har även positiva effekter på minnesanvändandet. Eftersom apparna nu laddas i form av ELF-filer kan Linuxfunktionen KSM, eller Kernel Samepage Merging, även spara en del minne. Det fungerar helt enkelt så att minnesblock med identiskt innehåll kan delas mellan appar.

Den tredje och sista stora förändringen, övergången till 64 bitar, hade till stor del kunnat göras i Dalvik, men då beslutet redan var fattat sedan länge har man valt att inte uppdatera Dalvik för 64 bitar.

Många av fördelarna med 64 bitar skall egentligen tillskrivas processorn, men några anpassningar har gjorts i ART, varav framförallt en är värd att nämnas – anropstabellen har begränsats till 32 bitar. Detta innebär att även om hela adressrymden på 64 bitar kan användas så läggs exekverbar kod i den första delen av minnet – den som kan adresseras med endast 32 bitar. Fördelen med den här begränsningen, som på engelska heter reference compression, är att storleken på den exekverbara koden kan hållas nere. Värt att notera är att alla data som behandlas i apparna fortfarande kan nyttja hela adressrymden. Detta kan tyckas vara en stor begränsning, men det finns idag inga appar som kommer i närheten av gränsen på 4 GB. Samma principer används ofta vid AOT-kompilering för Java på persondatorer och servrar.

**ART KOMMER NATURLIGTVIS** att direkt kunna dra nytta av andra fördelar med 64-bitarsarkitekurerna, som ökad prestanda vid stora beräkningar, bättre hantering av flyttal och fler register.

I dagsläget har ART stöd för ARMv8-64 och AMD64. MIPS-64 är planerat, men inget releasedatum är bekräftat.

Vad kommer då skiftet från Dalvik till ART att innebära rent praktiskt?

Slutanvändaren kommer att märka två förändringar , att apparna kommer att flyta mycket bättre och att ett fåtal appar som flyttar minnesreferenser mellan java och native på ett felaktigt sätt kommer att sluta fungera. Priset för den ökade prestandan är att det tar längre tid att installera apparna och att de tar mer lagringsutrymme i anspråk.

**GOOGLE HAR PRESENTERAT** data över exekveringstider för några exempelappar, som tyder på två till tre gånger bättre prestanda. Flera oberoende undersökningar bekräftar siffrorna.

Utvecklaren behöver se över de appar som använder native-kod, vilket i dagsläget rör sig om mindre än 15 procent av alla appar, enligt Google själva.

Eftersom ART släpptes som beta i Android 4.4 och som huvudsaklig exekveringsmiljö i Android 5.0, har programutvecklarna haft lite mer än ett år på sig att testa sina appar med ART. ■