Dalvik Virtual Machine

Hjälper eller stjälper?

A K E D H I N D I
o c h M I C H A E L L I N D B L O M

KTH Datavetenskap
och kommunikation

Examensarbete
Stockholm, Sverige 2010

Dalvik Virtual Machine

Hjälper eller stjälper?

A K E D H I N D I
o c h M I C H A E L L I N D B L O M

Examensarbete i datalogi om 15 högskolepoäng
vid Programmet för datateknik
Kungliga Tekniska Högskolan år 2010
Handledare på CSC var Cristian Bogdan
Examinator var Mads Dam

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2010/hindi_aked_OCH_lindblom_michael_K10042.pdf

Kungliga tekniska högskolan
Skolan för datavetenskap och kommunikation

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

FÖRORD

Denna rapport tar upp hur Google tillsammans med Open Handset Alliance optimerad Dalvik Virtual Machine för mobila enheter. Många av de ord och benämningar som används i rapporten är skrivna direkt på engelska, detta för att inte förvirra läsaren då en svensk motsvarighet kan vara svår att hitta eller ej existerar. Arbetet är utfört av Aked Hindi och Michael Lindblom; elever vid Kungliga Tekniska högskolan i Stockholm. Arbetet har delats upp jämt så att arbetsbördan legat på en likvärdig nivå.

Aked Hindi har skrivit: Inledning, Metoder och Java Virtual Machine.

Michael Lindblom har skrivit: Bakgrund och Dalvik Virtual Machine.

Jämförelsen mellan Java och Dalvik Virtual Machine samt de experiment vi utfört har gjorts tillsammans för att till fullo utnyttja våra kunskaper. Diskussion och slutsats har skrivits i samarbete mellan varandra för att samordna våra konklusioner samt få så pass många infallsvinklar som möjligt.

SAMMANFATTNING

Android är ett relativt nytt operativsystem som skapats av Google tillsammans med Open Hanset Alliance. Android's utvecklingsmiljö är på många sätt lik den i java, där även båda använder en Virtual Machine. Likaså många av de bibliotek som används är identiska med de som idag används i Java. Android har från grunden utvecklats och designats för mobila enheter. Anpassningar har därför gjorts för att förbättra batteritiden, minneshanteringen samt exekveringshastigheten. Detta har lett till att Google valt att utveckla en helt ny Virtual Machine som de valt att kalla Dalvik. Denna rapport tar upp hur Dalvik Virtual Machine anpassats för mobila enheter, hur Dalvik skiljer sig mot Java Virtual Machine och om de anpassningar som gjorts verkligen givit resultat. Detta undersökt med hjälp av ett antal experiment framtagna för att lyfta fram de viktigaste arkitektiska skillnaderna. Resultaten av experimenten visar att Dalvik har anpassats för mobila enheter och att de arkitektiska beslut som tagits är väl motiverade.

ABSTRACT

DALVIK VIRTUAL MACHINE

HELP OR HINDER?

Android is a relatively new operating system created by Google together with Open Handset Alliance. The development environment in Android has many similarities to Java's programming platform, whereby both platforms use Virtual Machines. Even many of the libraries used are identical to those currently used in Java. Android has been developed from ground up with mobile devices in consideration. Adjustments have therefore been made to improve battery life, memory management and execution speed. As a result, Google decided to develop a completely new Virtual Machine that they have chosen to call Dalvik. This report addresses how Dalvik Virtual Machine has been adapted for mobile devices, how Dalvik contrasts with Java Virtual Machine and if the adjustments made really paid off. This is analyzed by using a number of experiments designed to highlight the main architectural differences. The result of the experiment show that Dalvik has been adapted for mobile devices and the architectural decisions taken are well motivated.

Innehållsförteckning

1. Inledning

Android är ett relativt nytt operativsystem i den splittrade marknaden för mobila enheter. Idag finns operativsystem såsom Symbian, Windows Mobile, iPhone OS och det nyligen lanserade Android. Alla har de sina styrkor och svagheter men gemensamt för dem alla är att de är utvecklade för ett mobilt användande. Detta medför en hel del begränsningar och då framförallt för hårdvaran. En mobiltelefon har idag ungefär samma prestanda som en stationär dator tillverkad vid slutet av 90-talet. Även energikonsumtionen för en mobiltelefon är kraftigt begränsad då den enda bestående energikällan är dess batteri. Programvara för mobila enheter måste därmed köras på hårdvara som prestandamässigt ligger 10 år bakåt i tiden samtidigt som energiförbrukningen bara är en bråkdel av en stationär enhets energikonsumtion.

Den här rapporten tar upp hur Open Hanset Alliance med Google i spetsen har valt att tackla dessa problem. Hur kan en mobil enhet med Android som operativsystem hantera avancerade applikationer med dessa begränsningar?

Google har valt en utvecklingsmiljö som på många sätt liknar Java. Syntaxen och stora delar av det bibliotek Android använder sig av är identiska med de bibliotek som idag finns i Java Development Kit (JDK). Även om syntaxen och biblioteken i de flesta fall är identiska med Java så finns det en hel del skillnader, främst under själva programkörningen. I Android har man valt att inte använda sig av den mycket vanligt förkommande Java Virtual Machine (Java VM) som mer eller mindre är standard då javakod körs. Google har istället valt att utveckla sin egen VM som fått namnet Dalvik. Genom att använda en egenutvecklad VM har Google lyckats angripa de begränsningar mobila enheter lider utav. Begränsningar Java VM aldrig anpassats för då dess primära användningsområde fortfarande är för stationära enheter.

Fokus för denna rapport kommer därför ligga kring Dalvik VM. Följande frågeställningar kommer ligga till grund för de tester och undersökningar rapporten beskriver.

- Varför behövs Dalvik VM?
- Vad skiljer Dalvik och Java VM?
- Hur presterar Dalvik jämfört med Java VM?

För att få svar på dessa frågor beskrivs Dalvik VM ur en arkitektisk synvinkel, vilka anpassningar har gjorts för att förbättra prestandan?

Dalvik beskrivs i detalj och jämförs med Java VM för att lyfta fram de arkitektiska beslut som tagits för att anpassa Android och Dalvik VM för mobila enheter. Med hjälp av egna testresultat jämförs Dalvik med Java VM. Experimenten är framtagna för att illustrera och utvärdera de mest tongivande arkitektiska förändringarna Dalvik försetts med för att förbättra

minnes och processorbelastningen. Den arkitektiska djupdykningen tillsammans med resultaten av de tester vi utfört ligger till grund för de konklusioner vi kommit fram till.

2. Metoder

Google har tillsammans med Open Handset Alliance optimerat Dalvik för att möta de behov ett mobilt användande ställer på operativsystemet och i fallet med Android, även för Dalvik VM. En mobiltelefon har idag mycket begränsade resurser jämfört med bärbara och stationära datorer. Begränsningarna innefattar bl.a. processorhastighet, minne samt energiförbrukning. *Tabell 2.1* visar förhållandet mellan hårdvaran för en mobiltelefon och en stationär dator idag.

	Mobila enheter	Stationär dator
Processor	250-1000MHz (singelkärnig)	2000-3000MHz (flerkärnig)
Ramminne	64-512Mb	2-8Gb
Lagringsminne (för applikationer)	20-100Mb	100Gb-500Gb
Energiförbrukning	10-500mA	20-40A

Tabell 2.1: Jämförelse mellan en mobil och stationär enhets hårdvara

Dessa begränsningar sätter spår i arkitekturen för Dalvik då förutsättningarna för en hållbar plattform för applikationer kraftigt har reducerats i jämförelse med stationära enheter. Genom ett antal experiment avser vi att testa om dessa arkitektiska förändringar givit några resultat. Som referenspunkt används Java VM då detta ses som en naturlig jämförelse; utvecklingsmiljön för Android och Dalvik liknar på många sätt den miljö javautvecklare använder sig av.

Experimenten har till syfte att testa och utvärdera de anpassningar som gjorts för att förbättra minnesallokeringen och processorbelastningen för Dalvik. För att ge en så klar bild av resultatet som möjligt har experimenten delats upp på följande sätt.

- Experiment 1: Undersöker de optimeringar som gjorts för att förbättra minnesallokeringen.
- Experiment 2: Undersöker de optimeringar som gjorts för att minska processorbelastningen.

Utvecklingen av Dalvik sker i rasande fart då operativsystemet Android fortfarande är relativt ungt. Större förändringar är därför att vänta både för Android och för Dalvik. Flera indikationer[1] pekar på att Dalvik inom en snar framtid kommer att uppdateras med en Just In Time (JIT) kompilator[2]. Den senaste versionen av Android har redan idag en experimentell version av JIT-kompilator men som i dagläget inte är aktiverad[3]. Då källkoden för Android är tillgänglig för alla kan vem som helst aktivera JIT-kompilatorn. Google påstår att den

[1] Google. *Speakers @ Google I/O*. 2010.
[2] För mer information angående JIT-kompilatorer, (se punkt *4.3.3 JIT-Kompilator* för mer information).
[3] H-online. *Android's Dalvik to be JIT boosted*. 2009.

potentiella prestandavinsten med JIT-kompilator i Dalvik inte är lika betydande som prestandaförbättringen man ser i Java VM. För att ge en tidig indikation på hur pass mycket snabbare Dalvik är med JIT-kompilatorn aktiverad har vi därför utfört ett tredje experiment.

- Experiment 3: Undersöker om tidig version av JIT-kompilatorn ger förbättrat resultat. Resultatet jämförs sedan med den inbyggda JIT-kompiltorn i Java VM.

Resultatet av detta experiment kan skilja sig radikalt mot den slutgiltiga versionen av JIT-kompilatorn då den version som finns tillänglig idag inte är färdigutvecklad. Experimentet ger därför bara en fingervisning för vad vi i framtiden kan förvänta oss av Dalvik.

Innan vi går vidare med mer djupgående beskrivningar av Dalvik och Java VM är det viktigt att förstå i vilken kontext de befinner sig i. Nästkommande sidor kommer därför beskriva Android, Java samt andra relaterade plattformar för att ge dig som läsare bättre förståelse kring miljön Dalvik VM agerar i.

3. Bakgrund

3.1 Android

Android är ett öppet operativ system för PDA och Smartphones ursprungligen är utvecklat av Google; lansering skedde officiellt den 15 november 2007. Android utvecklas numera av en allians vid namnet ”Open handset Alliance”; en grupp bestående av 65 framstående teknologiska företag inom mobilteknologi. Google valde att använda sig av Linux då de utvecklade Android. Mycket av den underliggande processhanteringen samt säkerheten hanteras därför av Linux kernel vilket ger en stabil och trygg grund att stå på. Att Linux bygger på öppen källkod samt att operativsystemet är relativt resurssnålt gjorde troligtvis inte beslutet svårare. Android har det senaste året gjort ett starkt avtryck inom den mobila industrin och anses idag vara en stark konkurrent till andra mobila operativsystem såsom Windows mobile, Symbian och iPhone OS. Tillväxten för Android är mycket stark och Android är för tillfället det snabbast växande operativsystemet för smartphones[4]. Detta ger Android en unik och eftertraktad position hos utvecklare som ser möjligheten att lansera program på en relativt ny plattform som växer i rasande fart.

3.2 Java

Programmeringsspråket Java utvecklades ursprungligen av Sun Microsystems i början av 90-talet och var främst avsett att användas i intelligenta elektroniska komponenter som t.ex. mikrovågsugnar och tvättmaskiner[5]. Språket fick först namnet Oak,(efter en ek som växte utanför utvecklaren James Goslings fönster) men eftersom det redan fanns ett annat programmeringsspråk med samma namn byttes namnet till Java . Målet med att utveckla Java var att ta fram ett språk som var plattformsoberoende och mer lättanvänt jämfört med C++ konstruktioner då dessa ansågs vara onödigt komplicerade och svåra att använda. Ironiskt nog bygger syntaxen i Java fortfarande delvis på C++. Idag anses Java vara ett av de mest populära programmeringsspråken och används av tusentals programmerare världen över[6]

[4] Geeky gadgets. *Android Fastest Growing Smartphone OS. 2010.*
[5] Sectordata. *Om java.* 2008.
[6] Langpop. *Programming Language Popularity. 2010.*

3.3 Andra plattformar

Den mobila marknaden är idag en mångfacetterad marknad, kraftigt uppdelad i flera olika aktörer. Stora förändringar sker på de olika plattformarna med bara några års mellanrum för att förstärka sin position i den kraftiga konkurrensen. Av dessa anledningar kan det vara svårt att följa de förändringar som sker gällande arkitekturen för operativsystemen samt programexekvering för respektive plattform. Under denna rapport kommer aktuella plattformar tas upp *(2010)*; plattformar som kan vara utbytta och föråldrade bara om ett par år. Följande operativsystem och VMs är beskrivna mycket ytligt, ytterligare fördjupning hänvisas till egen efterforskning av läsaren, fotnoter och referenser.

3.3.1 Symbian

Smartphones är ett begrepp för avancerade mobiltelefoner och den kategori av mobila enheter som växer snabbast[7]. Tack vare en tidig introduktion har Nokia med Symbian en stark position och nära 50% av smartphone marknaden[8] även om de senaste åren har börjat visa vikande siffror[9] för den finska mobiltillverkaren som numera äger Symbian. Till skillnad från många andra mobila operativsystem har Symbian ingen ”storebror” för stationära enheter. Symbian har utvecklats enbart för mobila enheter och har på så sätt optimerats för begränsade resurser tidigt i utvecklingsstadiet. Applikationsutveckling för Symbian kan ske på flera olika språk[10], vanligast är dock att man använder sig av C++. Då program kompileras görs det direkt till maskinkod, dvs. Symbian använder sig inte av någon typ av VM. Fördelen med att gå direkt till maskinkod är att koden bör köras väsentligt mycket fortare än om man skulle använda sig av någon typ av VM. Den vinst man gör i effektivitet är till bekostnaden av processisolering; att varje program kan köras i sin egen miljö. Detta gör att dåligt skrivna program kan påverka hela operativsystemet och i värsta fall orsaka systemkrascher. Symbian som helhet är dock ett stabilt operativsystem för mobila enheter som effektivt har anpassats för de krav mobilt användande ställer. Framtiden för Symbian ser emellertid oviss ut då konkurrensen tätnar och flera stora uppdateringar krävs för att hålla operativsystemet konkurrenskraftigt[11].

[7] Forrester Research: 2009: The Year Of High-End Phones
[8] Canalys. Press and research releases. 2009.
[9] Appleinsider. *Iphone rim taking over smartphone market*. 2009
[10] Symbian developer. *Symbian Documentation*. 2010.
[11] Nexus404. *Nokia Symbian v3 Phones Coming in Q3 2010. 2010.*

3.3.2 Windows Mobile

Microsoft har länge försökt ta sig in på den mobila marknaden. Detta har gjorts i olika former med varierat resultat. Den mobila satsningen sker i dag i form av Windows Mobile som bygger på Windows CE v5.2. För att effektivisera operativsystemet skiljer sig Windows Mobile markant från den stationära versionen av Windows som vi alla säkert stött på i någon form. Effektiviseringar i form av låg minnesförbrukning och stöd för olika processorarkitekturer har gjort det möjligt för operativsystemet att hävda sig i den tuffa konkurrensen. De senaste åren har emellertid intresset för Windows mobile sjunkit. En viss del kan skyllas på att Windows Mobile bygger på just Windows CE v5.2 som först lanserades år 2004. Stöd för funktioner i nya ARM processorer saknas vilket gör att prestandan kan bli lidande[12]. Precis som med Symbian sker programutveckling främst i C++ vilket ger effektivt utnyttjande av hårdvaran då kod kompileras direkt till maskinkod. Tyvärr sker denna prestandavinst till bekostnad av systemstabilitet då det oftast är svårare att isolera processer utan VM. Framtiden för Windows Mobile är i dagläget osäker då Microsoft, precis som Nokia med Symbian, valt att totalt omarbeta hela operativsystemet för att möta de krav marknaden ställer[13].

3.3.3 Blackberry OS

Research in Motion eller RIM som det förkortas är en mobiltelefontillverkare som gjort stora framgångar; främst i USA där de har ca 50 % av marknaden[14]. I Europa har de inte haft lika stor genomslagskraft men är ändå en stor aktör på marknaden för mobila enheter pga. av sina framgångar på andra sidan Atlanten. RIMs enheter bygger på ett egenutvecklat operativsystem som länge legat före konkurrenter när det gäller att anpassa operativsystemet för tjänster som e-post och webbläsare. Operativsystemet heter Blackberry OS och använder Java som utvecklingsverktyg för externa program[15]. RIM har däremot valt att inte utveckla någon egen VM utan håller sig till standard versionen, dvs. Java VM. Marknaden för mobila enheter är tuff med många globala internationella företag; det kan därför bli svårt för RIM att hävda sig i konkurrensen.

[12] Mobiletechworld. *HTC HD2 review the best smartphone ever*. 2009
[13] Electronista. *Ballmer on win mobile 6.5*. 2009.
[14] Blackberrysync. *Blackberry dominating U.S. smartphone market shares*. 2009.
[15] Blackberry. *Developers*. 2010.

3.3.4 iPhone OS

Apple har med iPhone på kort tid plockat marknadsandelar och är numera en av marknadens mest populära smartphones[16]. Denna framgångssaga beror mycket på iPhones operativsystem som av många upplevs som mycket användarvänligt och grafiskt tilltalande[17]. iPhone OS bygger på Mac OSX; det operativsystem Apple använder sig av för sina bärbara och stationära datorer i MacBook och iMac serien. Därmed har iPhone OS en gedigen och stabil grund att stå på då operativsystemet bygger på Unix. Modifieringar har självklart gjorts, bl.a. för att anpassa operativsystemet för ARM arkitekturen processorn i iPhone använder sig av och inaktivering av multitasking[18] pga. den relativt svaga hårdvaran . Utvecklingsmiljön för iPhone är även den likartad med Apples övriga produkter. Utveckling sker i Xcode och kompileras direkt mot hårdvaran, dvs. program skrivna för Mac OSX kan inte köras på iPhone OS trots att de är skrivna i samma utvecklingsmiljö. Apple har med iPhone en mycket stark och lojal användarbas och har därmed goda förutsättningar för fortsatt utveckling av iPhone och operativsystemet iPhone OS.

3.3.5 Adobe Flash Player

Den i särklass vanligaste metoden för att visa interaktivt och grafiskt tilltalande hemsidor är idag Adobe Flash Player. Upp till 96 % av dagens stationära och bärbara datorer har idag stöd för Flash[19]. Windows, Mac OSX och Linux har alla stöd för Flash; även flera mobila operativsystem såsom Symbian, Windows Mobile och Android har numera Flash funktionalitet. Detta multi-plattforms stöd är möjligt tack vare att Adobe Flash Player bygger på en VM och har bidragit till den starka positionen Flash har idag. För att utveckla programvara för Flash använder man ett programmeringsspråk kallat ActionScript. Till en början var ActionScript ett mycket enkelt och i många fall ineffektivt scriptspråk[20]. Utvecklingen har däremot skett i rasande takt för att möta marknadens krav. Idag är språket objektorienterat med stöd för avancerade 3D funktioner. Förändringarna har vart så pass stora att bakåtkompabilitet blivit ett problem. Detta har gjort att Flash idag bygger på två olika VMs; en som hanterar tidigare Flash versioner och en som hanterar dagens version. Då nya avancerade funktioner implementerats samtidigt som mobila enheter börjat få stöd för Flash har prestanda varit i fokus för utvecklingen. Med den senaste version av ActionScript VM har Adobe även valt att implementera en JIT-kompilator. För att förbättra prestandan ytterligare finns även planer på att implementera Graphic Processing Unit (GPU) accelerator för ActionScript VM[21]. Trots den starka position Flash har idag ser framtiden oviss ut.

---

[16] Cnet. *iPhone 3G crowned most popular phone in U.S.* 2008.
[17] Bloomberg. *Apple's IPhone Generates Buzz That May Top Mustang.* 2007.
[18] Multitasking innebär att ett system kan hantera flera applikationer samtidigt.
[19] Reitmaier Rick. *Adobe Flash Player ActionScript Virtual Machine. 2006.*
[20] Vivisectingmedia. *Flash Internals: The ActionScript Virtual Machine. 2007.*
[21] Anandtech. *AnandTech Tests GPU Accelerated Flash 10.1 Prerelease. 2009.*

Säkerhetsbuggar och kvalitetsproblem har sargat Flashs rykte[22]. Detta kan ligga till grund för att stora profiler inom teknikbranschen tagit avstånd från Flash och numera väntar på HTML5[23].

3.3.6 Java ME

I ett försök att förena utvecklingsmiljön för mobila enheter har Sun valt att skapa en mobil variant av Java som de kallar Java ME. Tanken var att med hjälp av Java och dess VM skapa en universell utvecklingsplattform, oberoende av vilket operativsystem den mobila enheten använder sig av. Java ME har flera likheter med sin storebror Java, men för att anpassas för mobila enheter har Java ME kapats i funktionalitet. Under slutet av 90-talet, då Java ME introducerades, var prestandan för mobila enheter en bråkdel av vad den är idag. Detta gjorde att Sun var tvungen att sätta en mycket låg nivå för de bibliotek som var obligatoriska i Java ME. Få bibliotek gjorde det möjligt att använda Java ME för enheter med mycket låg prestanda och minnesstorlek[24]. Mobiltillverkare fick istället själva utöka och anpassa detta bibliotek för sina produkter vid behov. Kontentan av detta blev en splittrad marknad där varje mobiltillverkade valde sin implementation av Java ME; tvärt emot dess ursprungliga syfte. Idag har i stort sett varje mobilt operativsystem sin egen utvecklingsmiljö och behovet för en unifierad plattform har kommit i skymundan.

3.3.7 VMware

VMware grundades 1998 och baserar sin kärnverksamhet kring VMs. Idag har VMware flera produkter för virtualisering med olika egenskaper och syfte, gemensamt mellan produkterna är att de bygger på mer eller mindre samma VM. Deras produkter anses av många tillhöra den absoluta toppklassen för sin marknad[25]. För att förbättra prestandan har VMware valt en lite annorlunda väg jämfört med flera av deras konkurrenter. En vanlig metod för att emulera processorinstruktioner är att helt enkelt ta sig an en instruktion i taget, en annan metod är att emulera stora block av instruktioner som körs därefter. VMware selekterar istället delar av koden och emulerar inte instruktioner för hårdvara som inte fysiskt är närvarande[26]. Detta kan öka prestandan avsevärt[27]. För att förbättra prestandan ytterligare emulerar VMware inte processor instruktioner som kan köras direkt, utan behov av översättning. Instruktioner som

[22] Theregister. *Adobe apologizes for festering Flash crash bug. 2010.*
[23] Theregister. *Steve Jobs dubs Google's 'don't be evil' motto 'bulls**t'. 2010.*
[24] Oracle. Oracle's *Support for Open Source and Open Standards*.2010.
[25] Nytimes. *Challenging Microsoft With a New Technology. 2009.*
[26] Vmware. *The Architecture of VMware ESXi.* 2007.
[27] Capitalhead. *Benchmarking VMware ESX Server 2.5 vs Microsoft Virtual Server 2005 Enterprise Edition. 2006.*

behöver bli omskrivna skrivs då om dynamiskt, något som VMware kallar ”binary translation”. Detta gör att vid optimala förhållanden försämras prestandan bara 0-6%[28].

Nästkommande sidor kommer att beskriva Dalvik och Java VM, detta för att du som läsare lättare ska förstå hur dessa VMs fungerar samt vad som skiljer dem åt.

[28] Vmware. *A Performance Comparison of Hypervisors.* 2007.

4. Virtual Machines

Begreppet Virtual Machine grundades av Popek och Goldberg i en publicerad artikel år 1974[29]. I artikeln beskrivs en VM på följande sätt; ”an efficient, isolated duplicate of a real machine". Fritt översatt till svenska betyder detta; ”en effektiv, isolerad kopia av en riktig maskin”. Med en VM kan man med andra ord ”kopiera” en maskin och använda den kopierade maskinen på vilken plattform man vill.

Idag finns det två huvudkategorier av VM.

- System VM – Kan användas som en helt fristående plattform där ett helt operativsystem kan köras.
- Process VM – Används för att köra ett enskilt program, dvs. en enskild process.

VMs började först användas i form av System VM. Att kunna köra flera olika operativsystem isolerade från varandra på samma hårdvara vår något som lockade utvecklare. Idag är System VM vanligt förekommande på servrar och används flitigt i flera olika sammanhang.

Process VM kan hittas i flera av de mest använda programmeringsspråken. Den stora fördelen med Process VM är att program kan köras isolerat i sin egen miljö på valfritt operativsystem. Detta plattformsoberoende gör att VM blivit mycket populära då utvecklingskostnader och utvecklingstid kraftigt kan reduceras.

4.1 Dalvik Virtual Machine

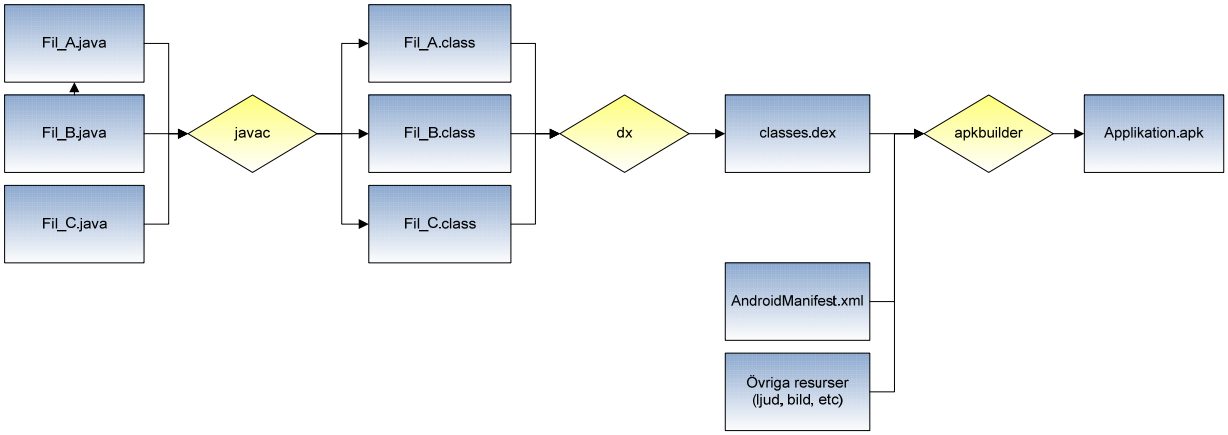
För att anpassa Android för mobila enheter har Google valt att ersätta Java VM med en egenutvecklad process VM. Arbetet har pågått sedan år 2002 men först år 2008 lanserades den första mobiltelefonen med Android tillsammans med Dalvik VM[30]. Dan Bornstein, Dalviks skapare, namngav Dalvik efter den by på Island hans förfäder härstammar ifrån. Dalvik VM har ända sedan lanseringen av Android varit en av de viktigaste grundstenarna för operativsystemet och har genomgott omfattande optimeringar med fokus att förbättra prestandan.

Alla applikationer tillängliga för Android använder Dalvik VM för att kunna exekveras. Dalvik är den process som håller samman hårdvaran och operativsystemet med övriga applikationer och ser därmed till att kod skriven för Android kan köras.

[29] Gerald J. Popek and Robert P. Goldberg. "Formal Requirements for Virtualizable Third Generation. Sida 412. 1974.
[30] Googleblog. *The first Android-powered phone*. 2008.

4.1.1 Kompilering

För att kunna exekvera program skrivna för Dalvik VM behöver man först kompilera koden. Detta görs i två steg. Med hjälp av kompilatorn ”javac” kompileras koden ner till bytekod. Bytekoden man får ut av javac har filändelsen ”class” och är endast läsbar av Java VM. För att Dalvik VM ska kunna läsa koden behöver class-filen bearbetas av ännu en kompilator, nämligen ”dx”. Verktyget är skapat för att sammanfoga flera class-filer till en ny fil med filnamnstillägget ”dex”. Den nya filen skapad av dx paketeras med ”apkbuilder” tillsammans med andra stödfiler och får sitt slutgiltiga format i form av en apk-fil. *Figur 4.1.1* ger en förenklad beskrivning av hur kompileringen går till.

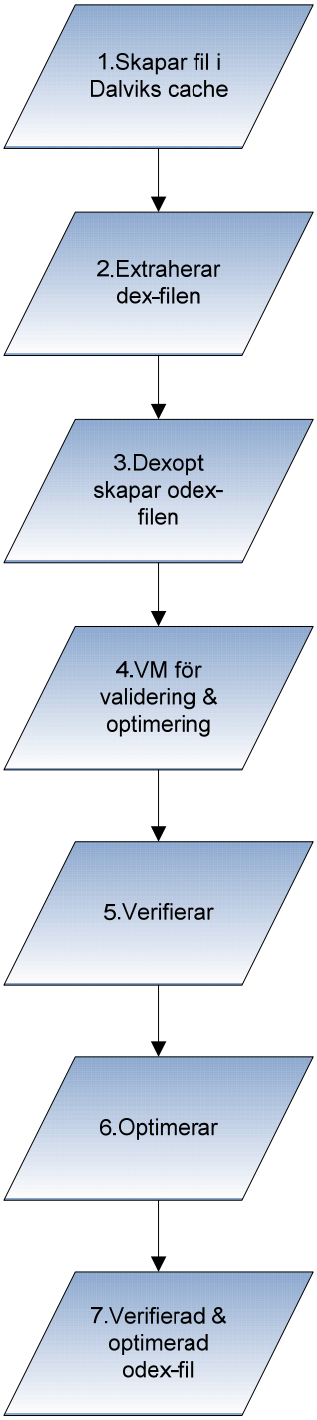


4.1.1: Illustration av kompileringsprocessen; från java till apk-fil.

4.1.2 Installation

Dalvik VM har optimerats för enheter med begränsad hårdvara; de flesta optimeringar sker under programkörning men arbetet med att förbättra prestandan börjar redan vid installationen av ny programvara. Applikationer avsedda för Android har filändelsen ”apk”; dessa filer är inget annat än en komprimerad mapp innehållande en dex-fil med tillhörande resurser i form av ljud, bild, etc. samt manifestfilen[31], *se figur 4.1.1* . Vid installation av en ny applikation görs en förberedande verifiering samt optimering av applikationens dex-fil. Efter lyckad verifiering och optimering kommer den körbara filen att ha filändelsen ”odex”. Optimeringen och valideringen binder odex-filen till den specifika enheten; det går med andra ord inte att flytta över odex-filen till en annan enhet och köra programmet därifrån. Verifiering och optimering görs på följande sätt[32]:

[31] Manifestfilen innehåller väsentlig information rörande applikationen till Android
[32] Netmite. *Dalvik optimization and verification with dexopt.2008.*



1. En fil skapas i dalviks cache, denna fil kommer innehålla information som senare kommer att behövas för att skapa odex-filen. Dalviks-cache är av säkerhetssjäl normalt sett skrivskyddad men vid installation av ny programvara får systemet tillfällig rättighet att använda cachen.

2. Dex-filen extraheras ur apk-filen. För att kunna ha lätt och snabb tillgång till dex-filen flyttas filen in i minnet. Datastrukturen anpassas och struktureras om för att lättare få tillgång till den information som behövs. Filens offset och datans index bör ligga i samma intervall, även detta kontrolleras.

3. Med hjälp av verktyget ”dexopt” skapas nu odex-filen, samtidigt raderas dex-filen ur apk-filen.

4. En separat virtuell maskin skapas för att validera och optimera varje klass i odex-filen.

5. Syftet med verifieringen är att säkerställa att ingen instruktion går utanför ramarna för vad programvara får göra i Android, t.ex. kringgå systemets säkerhet. Varje instruktion i bytekoden verifieras steg för steg för att skydda systemet.

6. Optimeringsprocessen av dexfilen görs på följande sätt.

   - Ersätta metodens index med en vtabell[33], för metodanrop.
   - Ta bort tomma metoder eller metoder som inte används.
   - Tillämpa statisk länkning[34] om möjligt.
   - Utföra alla förberedande beräkningar av datan, vilket sparar heap-plats[35] samt snabbar upp exekveringshastigheten.

7. Den nu verifierade och optimerade odex-filen flyttas till sin slutgiltiga position i systemets applikationskatalog.

4.1.2: Illustration av installationsprocessen; från apk till odex-fil.

[33] Wikipedia. *Virtual method table*. 2010
[34] Wikipedia *Static library. 2010.*
[35] Plats i minnet som ändvänds av applikationers minnesallokering, applikationers lokala variabler (data).

Efter att installationen är genomförd är alla förberedelser klara inför exekveringen. Installationsprocessens syfte, att optimera, säkerställa och verifiera koden medför att dessa uppgifter troligtvis inte behöver utföras vid programexekvering. Mer om detta i nästa punkt.

4.1.3 Exekvering

Androids installationsprocess hanterar valideringen och optimeringen av applikationer. Vid programexekvering utförs i normalt fall inte validering och optimering då detta redan genomförts. Att koden är validerad och säkerställd är mycket viktigt då detta annars kan riskera systemets säkerhet. För att garantera att ingen modifikation skett med programmet efter installationen jämförs checksumman[36] mellan odex-filen och systemets Bootclasspath[37]. Varje odex-fil har beroenden till systemfiler i Android. Dessa beroenden sparas med hjälp av en checksumma i odex-filen. Om checksumman i odex-filen inte överensstämmer med checksumman för systemfilerna måste installationsprocessen köras igen innan exekveringen kan fortsätta. Vid uppdatering av operativsystemet kan viktiga systemfiler modifieras, detta är ett exempel på när verifieringen och optimeringen måste köras igen.

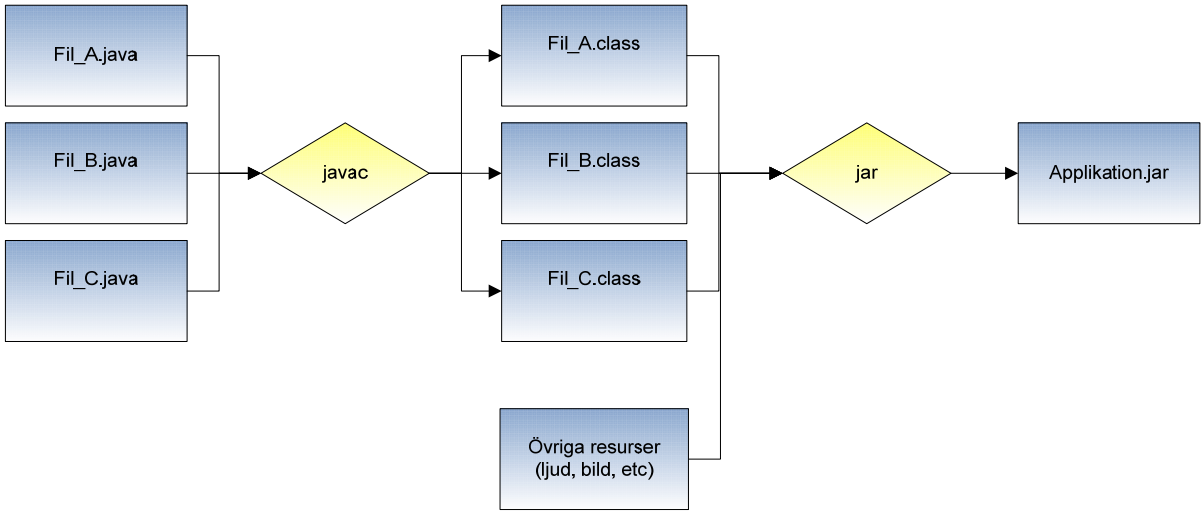
4.2 Java Virtual Machine

Sun har med Java VM en mycket kompetent och populär VM som enligt dem själva finns i fler än 4.5 miljarder enheter[38]. Java VM används för att exekvera Java bytekod och är en mycket viktigt komponent i JDK. Syften med Java VM är att skapa en plattform för applikationer, oberoende av vilket operativsystem som körs under. Därav deras slogan ”write once, run anywhere”; fritt översatt till svenska betyder detta, ”skriv en gång, kör var som helst”.

[36] Unik identifieringssträng.
[37] En lista med sökvägen till alla applikationer installerade på systemet.
[38] Java. *Learn About Java Technology. 2010.*

4.2.1 Kompilering

Precis som med de flesta andra programmeringsspråk behöver kod skriven för Java VM först kompileras. Med hjälp av kompilatorn ”javac” kompileras koden till bytekod som representeras med filnamnstillägget ”class”. Den nya filen skapad av kompilatorn kan nu köras av Java VM men för att underlätta för slutanvändaren paketerats class-filerna tillsammans med tillhörande resurser till en jar-fil[39]. Detta görs med verktyget ”jar”. Paketeringen ser till att koden med tillhörande resurser lätt och smidigt kan exekveras från en och samma fil. Se *figur 4.2.1* för en förenklad beskrivning av hur kompileringen går till.



4.2.1: Illustration av kompileringsprocessen; från java till jar.

[39] Java Sun. *Java Archive (JAR) Files*.2010.

4.2.2 Exekvering

Vid exekvering av programvara skriven för Java VM måste koden först bearbetas i ett antal steg. Dessa steg säkerställer att koden inte är skadlig för systemet samt förbereder programmet inför initieringen. Detta liknar på flera sätt installationsprocessen applikationer skrivna för Android måste genomgå men som i Java VM istället görs vid exekveringen. Följande steg utförs vid exekvering av en jar-fil.

- **Uppladdning**: Bytekoden laddas upp i heapen; dock bara från den klass som innehåller main-metoden.
- **Länkning**: består av tre steg
    - **Verifiering**: Kontrollerar att bytekoden inte innehåller skadlig kod; detta för att garantera att ingen operation bryter mot säkerheten eller att koden leder till systemkrascher.
    - **Förberedelse**: Här utförs minnesallokering av statiska fält i heapen som behövs av bytekoden.
    - **Resolution**: Verifierar att referenser från nuvarande klass är korrekt länkat till de andra klasserna.
- **Initiering:** Klassens metoder och statiska fält initieras, men innan detta sker skapas en kedjereaktion. Klassens superklass måste först initieras vilket leder till att superklassens superklass måste initieras. Detta fortsätter tills objektklassen har initieras då objektklassen ligger högst upp i kedjan.

Efter initiering av klassen kan exekvering av övriga klasser länkade till huvudklassen påbörjas. Detta görs efter samma regler beskrivna ovan.

4.3 Dalvik vs Java

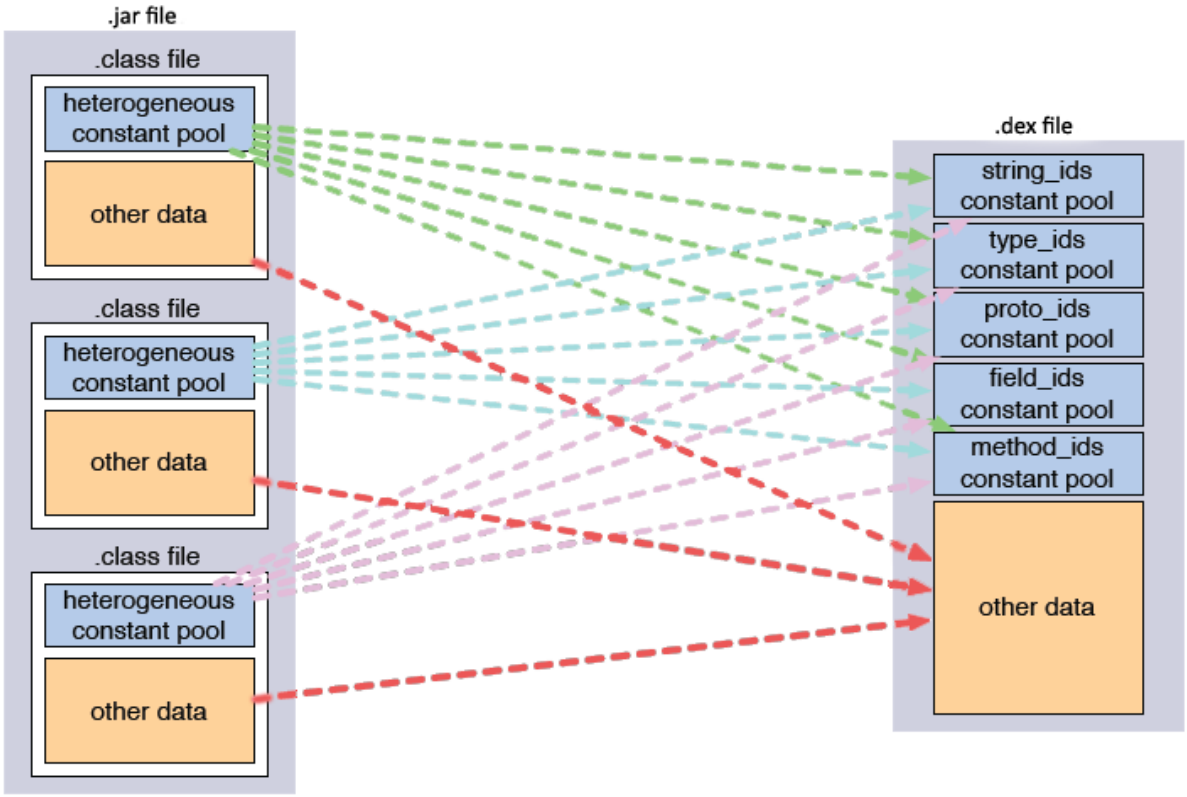
4.3.1 Minimal redundans

Redundans är ett vanligt problem inom de flesta områden. Att samma information lagras på olika platser är ett slöseri på resurser och bör undvikas i största möjliga mån. Om informationen istället kan delas mellan de komponenter som behöver den, sparas värdefullt lagringsutrymme samtidigt som funktionaliteten bibehålls. I Dalvik VM har man därför gjort ansträngningar för att minimera förekomsten av redundans i bytekoden.

Vid kompilering av java-filer får varje klass i källkoden sin representation av bytekod i form av en class-fil. Redundant kod tas bort klassvis, däremot finns det ingen algoritm som jämför

klasserna med varandra eller någon metod för att slå ihop likartade klasser. Man säger därför att Java VM har en ”constant pool” [40] på klass-nivå.

Vid utvecklingen av Dalvik har man sett ett behov och en möjlighet att slå ihop class-filerna och därmed minimera förekomsten av redundant kod. Vid kompilering med hjälp av vektyget dx sammanfogas därför class-filerna till en och samma dex-fil. Redundant kod tas bort applikationsvis och har därför en constant pool på program-nivå.



4.3.1: Illustration av constant pool av jar och dex fil.[41]

Figur 4.3.1 ovan beskriver hur information överförs från class-filerna i en jar-fil till dex-formatet Dalvik VM använder sig av. Class-filerna slås samman till en och samma dex-fil; redundant information mellan class-filerna tas bort då information kan delas mellan klasserna i en dex-fil.

[40] Constant pool är en benämning för den plats variabler lagras i bytekoden.
[41] Bornstein Dan. *Presentation Of Dalvik VM Internals*. Sida 15. 2008.

4.3.2 Reducerad opcode

En mycket viktig skillnad mellan Java och Dalvik VM är den grundläggande arkitekturen för bytekoden. Java är stackbaserad medans Dalvik VM är registerbaserad. Valet av arkitektur påverkar antalet opcodes[42] processorn behöver hämta. Kod kompilerad till en registerbaserad VM innehåller i stort sett alltid färre instruktioner än samma kod kompilerad till en stackbaserad.

Lokala variabler i en stackbaserad VM sparas i en array, för att kunna läsa eller skriva krävs därför att variablerna först pushas på stacken innan de kan användas. I en registerbaserad VM sparas variablerna direkt i registret, färdiga att användas på en gång. Ett enkelt sett att beskriva detta är genom en simpel operation; c = a + b.

Stackbaserad VM	**Registerbaserad VM**
0: iload_0 #A 1: iload_1 #B 2: iadd 3: istore_2 #C	add-int v0, v1, v2

Tabell 4.3.2: Exempel på stack och registerbaserad bytekod.

I den stackbaserade versionen finns det tre lokala variabler; a,b,c. Innan värdet av c kan beräknas måste a och b först pushas på stacken. När beräkningen utförs poppas de från stacken för att summeringen ska kunna genomföras, summeringen lagras sedan i lokala variabeln c. För att utföra den enkla beräkningen c = a + b krävs således fyra instruktioner. Den registerbaserade versionen sparar de lokala variablerna direkt i registret. Beräkningen kan på så sätt utföras med en gång utan att först behöva spara variablerna i registret. På så sätt kommer man även delvis runt problemet kring långsamt ramminne. Mycket av informationen som används vid beräkningar kan hämtas direkt från processorn istället för att först hämtas från ramminnet. Utan att gå in på detaljerad beskrivning av processorarkitekturen använder ARM processorer i de flesta fall pipelines. Registerbaserad bytekod kan köras väsentligt mycket snabbare på en processor som använder sig av pipelines[43]. I stort sett alla mobila enheter använder idag ARM processorer vilket innebär att registerbaserad bytekod bör kunna utnyttjas bättre.

Registerbaserade VMs har dock en del mindre bra egenskaper. Vid exekvering tar varje instruktion två bytes istället för en byte i en stackbaserad VM. Den första byten är till för matematiska operationer medan den andra är till för operanderna[44]. Detta innebär att bytekoden tar upp mer lagringutrymme än stackbaserad arkitektur; något som däremot inte

[42] Opcode står för ”operation code” vilken är en bytekodinstruktion.
[43] The shape of code. *Register vs. stack based VMs*. 2009.
[44] Stevens Ashley. *Developing for Android on ARM*. 2009.

bör påverka prestandan då Dalvik VM kan hämta två byte i taget. Den utökade filstorleken kompenseras av andra tekniker, bl.a. användandet av constant pool på program-nivå.

4.3.3 JIT-kompilator

En JIT-kompilator kompilerar delar av bytekoden direkt eller under programkörning till maskinkod för att förbättra prestandan. Maskinkod jobbar direkt mot hårdvaran och är på så sätt avsevärt mycket snabbare än bytekoden som först måste interpreteras. JIT-kompilatorer har utvecklats för att minska det prestandamässiga övertag programmeringsspråk skrivna direkt mot hårdvaran har.

För att förbättra prestandan har man i Java VM sedan länge implementerat en JIT-kompilator. Då en JIT-kompilator översätter bytekod direkt till maskinkod kan detta försämra uppstartstiden för programvara. En VM med JIT-kompilator tar oftast längre tid att starta men körs sedan snabbare. Sun har med Java VM därför valt en implementation av JIT-kompilator som själv avgör vilken kod som används mest och översätter endast den till maskinkod. Då bytekod exekveras i Java VM aktiveras en räknare som håller reda på hur ofta en metod i bytekoden körs. Om en metod används relativt ofta aktiveras JIT-kompilatorn och översätter metoden till maskinkod. Nästa gång metoden anropas exekveras maskinkoden direkt, utan att först behöva interpreteras av Java VM. I händelse av att metoden används väldigt ofta kan JIT-kompilatorn aktiveras igen, den här gången översätts bytekoden på ett effektivare sätt med flera optimeringar för att ytterligare förbättra prestandan. Vissa metoder kan därmed bli översatta till maskinkod relativt tidigt under programkörningen medan andra kanske inte översätts alls. På så sätt arbetar JIT-kompilatorn under programmets körning och förbättrar prestandan under programmets livstid.

Google har med Dalvik VM valt att inte använda någon form av JIT-kompilator. Detta är en sanning med modifikation då från och med version 2.0 av Android har en experimentell version av JIT-kompilator implementerad med som inte aktiverats. Officiellt menar Google att Dalvik klarar sig bra utan en JIT-kompilator då flera av Android bibliotek redan är skrivna i maskinkod och som kan användas med det inbyggda Java Native Interface (JNI) ramverket[45]. Biblioteken är skrivna i C eller C++ för att förbättra exekveringshastigheten. Några exempel är bl.a. det grafiska biblioteket Android använder sig av samt det bibliotek som används för att få tillgång till den inbyggda SQL databasen.

Den potentiella prestandavinsten bör på så sätt inte vara lika betydande som för Java VM. Google har än så länge inte officiellt lanserat sin JIT-kompilator; presentationen kommer ske i samband med Google IO 2010, vilket är bekräftade uppgifter[46]. Information tillgänglig idag är därmed bristfällig och otillräcklig. Indikationer pekar på att JIT-kompilatorn har liknande

[45] JNI är ett ramverk som används för att utöka standardbiblioteket med externa bibliotek skrivna direkt i maskinkod.
[46] Google. *A JIT Compiler for Android's Dalvik VM. 2010.*

funktionalitet som Java VMs. Troligt är att bara den kod som används regelbundet kommer att översättas till maskinkod av JIT-kompilatorn[47].

[47] H-online. *Android's Dalvik to be JIT boosted.*2009.

5. Datainsamling

Dalvik VM har optimerats för ett mobilt användande. Anpassningar har gjorts för att reducera filstorlek och instruktioner för att på så sätt kunna använda mobilens hårdvara på ett mer effektivt sätt. För att bevisa att dessa optimeringar verkligen har givit resultat har ett antal experiment genomförts. Experimenten har utförts på följande hårdvara och operativsystem.

	Google Nexus one (mobiltelefon)	**Stationär dator**
Processor	1GHz Cortex A8 (singelkärnig, ARM)	2.67GHz Core i7 (4-kärnig, X86)
Ramminne	512Mb	3Gb
Operativsystem	Android v2.1	Windows 7 (32-bit)
Virtual Machine	Dalvik VM v.2.1	Java v.6

Tabell 5: Den hårdvara som använts för experimenten.

Övriga specifikationer för hårdvaran är irrelevant då det endast är processorn och ramminnet som kan påverka resultatet av experimenten. För att bevisa att dessa optimeringar verkligen har givit resultat har följande experiment genomförts.

5.1 Experiment 1: Minimal redundans

Google har vidtagit åtgärder för att minimera redundans i Dalviks bytekod. Genom att minimera redundans kan bytekoden effektiviseras samtidigt som lagringsutrymme bevaras. För att undersöka om dessa optimeringar givit resultat har vi konstruerat följande experiment. Tre class-filer har skapats.

- Klassfil 1: Tar in en lista av tal som parameter och skriver ut summan av listan
- Klassfil 2: Innehåller en metod som tar in en lista av strängar och skriver ut antal strängar i listan.
- Klassfil 3: Innehåller en metod som tar in en lista av strängar, lägger ihop samtliga strängar till en sträng, skriver ut strängen samt det ursprungliga antalet strängar.

Källkoden till programmet finns under Bilaga A.

Samtliga klasser har samma metodnamn, arraynamn och strängnamn. Vid kompilering till class-filer bör redundans uppstå mellan klasserna. Kompilering till dex-filer förväntas avlägsna förekomsten av redundans då constant pool ligger på program-nivå istället för klass-nivå. Class och dex-filerna undersöks i detalj för att se om de förändringar som gjorts verkligen fungerar i praktiken.

Den minimerade redundansen bör generera dex-filer som storleksmässigt är mindre än class-filerna de är kompilerade från. Ytterligare fyra applikationer kommer därför kompileras till class och dex-filer för att sedan komprimeras till jar respektive apk-filer. Applikationerna är skapade under tidigare kurser vi genomfört på Kungliga Tekniska Högskolan. Filstorleken kommer att jämföras mellan filformaten för att på så sätt kunna avgöra om den reducerade bytekoden producerar mindre filer.

5.2 Experiment 2: Reducerad opcode

För att reducera antalet instruktioner i program skrivna för Android är Dalvik VM registerbaserad. För att testa om Androids registerbaserade VM verkligen ger färre instruktioner har vi kompilerat ett spel till Java respektive Dalvik VM. Spelet är variant av damspel vilket bl.a. innehåller, loopar, arrayer, strängar och heltal. Följande tre class-filer skapar spelet.

- Board: Ritar upp spelbordet.
- Pawn: Hanterar spelpjäsernas förflyttningar.
- Piece: Objekt av spelpjäserna.

Källkoden till spelet finns under Bilaga B.

Spelet kompileras till Java och Dalvik VM för att kunna analysera bytekoden. Vid kompilering till dex-fil förväntas antalet instruktioner minska jämfört med class-filerna då registerbaserad kod generellt sett ger färre instruktioner.

5.3 Experiment 3: JIT-kompilator

För att förbättra prestandan använder Java VM en JIT-kompilator. En experimentell version av JIT-kompilator finns idag implementerad i senaste versionen av Android. Genom att aktivera JIT-kompilatorn i Android kan vi få fram tidiga resultat för hur kompilatorn kommer att prestera när den färdiga versionen lanseras. Google hävdar att den potentiella prestandavinsten med JIT-kompilator i Dalvik inte är lika betydande som prestandaförbättringen man ser i Java VM. Syftet med experimentet är därför att mäta prestandaförbättringen med den tidiga versionen av JIT-kompilatorn i Dalvik. Resultatet kommer sedan jämföras med Java VM och prestandaförbättringen JIT kompilatorn ger där. För att kunna utföra experimentet har vi skapat tre prestandatest.

- **Fibonnacci**: Mäter tiden i millisekunder som krävs för att beräkna ett visst Fibonnaccital[48]. Algoritmen är baserad på Fibonnaccis formel.

[48] Webbmatte. *Fibonaccis talföljd.*2010.

- **PI:** Mäter tiden i millisekunder som krävs för att beräkna PI med ett visst antal decimaler. För att beräkna detta använder vi Salamin–Brent algoritmen[49] för PI och Newtons metod[50] för roten ur.
- **Flyttalsoperationer:** Beräknar antalet flyttalsoperationer per sekund[51] som kan utföras av en processor. Här använder vi ett färdigskrivet prestandatest som kan laddas ner från följande länk[52].

Källkoden till prestandatesten finns under Bilaga C.

Varje testprogram har körts om tio gånger, därefter har medelvärdet räknats ut för att på så sätt få så pass opåverkat resultat som möjligt.

5.4 Verktyg

För att få fram tillförlitliga resultat har vi använt oss av olika verktyg. Dessa verktyg har bl.a. använts för att kompilera koden till bytekod, få fram dump-filer[53] av bytekod eller för att köra Java och Dalvik VM med eller utan JIT-kompilator.

- *Java till class-fil:* För kompilering av java till class-filer har vi använt oss av verktyget ”javac” som ingår i Suns JDK.
    - *Kommando:* Javac fil.java

- *Class till dex-fil:* För kompilering av class till dex-filer har vi använt oss av verktyget ”dx” som ingår i Googles Android Development Kit (ADK).
    - *Kommando:* dx –dex –output=fil.dex fil.class

- *Class till jar-fil:* För att sammanfoga class-filerna med tillhörande resurser används verktyget ”jar” son ingår i Suns JDK.
    - *Kommando: jar cf fil.jar fil.class*

- *Dex till apk-fil:* För att sammanfoga dex-filen med tillhörande resurser används verktyget ”apkbuilder” som ingår i Googles ADK.
    - *Kommando:* apkbuilder fil.apk –u –f fil.dex

- *Dump av class-fil:* För att få ut läsbar information ur en class-fil har vi använt oss av två vektyg. Javap som ingår i Suns JDK används för att få fram bytekode-

[49] Wikipedia. *Gauss–Legendre algorithm.* 2010.
[50] Amcgowan. *Newton’s Method for finding roots*. 2010.
[51] Flyttalsoperationer är en benämning för att antalet beräkningar av reella tal per sekund.
[52] Sourceforge. *Java Benchmark (Test PC).* 2009
[53] Dump-fil används för att få fram läsbar information ur en annars oläsbar fil.

instruktioner; verktyget ”clsd” som är skapat av företaget aQute[54] används för att få fram en detaljerad beskrivning av metodnamn, lokalavariabler mm ur class-filen; den såkallade ”constant pool”
    - *Kommando:* javap –verbose fil > fil.dump
    - *Kommando:* java –jar clsd-1.0.0.jar fil.class > fil.dump

- *Dump av dex-fil:* För att få ut läsbar information ur en dex-fil har vi använt verktyget ”dx” som ingår i Googles ADK. Då dx-tool får ut all nödvändig information behövs inget annat verktyg.
    - *Kommando*: dx –dex –dump-to=fil.dump fil.class

- *Avaktivering av JIT-kompilator för Java VM:* Vid exekvering av class-filer kan JIT-kompilatorn avaktiveras med hjälp av en parameter till verktyget ”java” som ingår i Suns Java Development Kit.
    - *Kommando:* java –Xint fil.class # JIT kompilator avaktiverad
    - *Kommando:* java fil.class # JIT kompilator aktiverad

- *Aktivering av JIT-kompilator för Dalvik VM:* För att aktivera den experimentella JIT-kompilatorn i Dalvik krävs modifikationer av Androids operativsystem. En förutsättning för detta är att operativsystemet är upplåst[55]. Aktivering av JIT-kompilatorn görs med följande tre steg.
    - Ersättning av systemfiler i Android.
    - Ändra behörigheten för de nya systemfilerna.
    - Aktivering av JIT kompilatorn i Androids build.prop-fil.

En detaljerad beskrivning för hur detta går finns på sidan[56].

5.4.1 Tillförlitlighet

De flesta av de verktyg vi använder oss av är hämtade från antingen Suns JDK eller Googles ADK. Vi förutsätter att dessa verktyg har genomgått omfattande kvalitetskontroller för att säkerställa pålitliga resultat. Ytterligare undersökningar för att garantera verktygens kvalité anses därför vara överflödigt och ej relevant för denna rapport.

Vi var dock tvungna att titta på verktyg utanför de standardiserade utvecklingsverktygen då verktyget ”javap” ur Suns JDK inte gav detaljerad information angående ”constant pool” . Verktyget ”clsd” är skapat av företaget aQute gav oss den information vi sökte. aQute är ett konsultföretag specialiserat inom OSGi-platformen[57] och Java. De ger ut kurser, erbjuder

[54] Aqute. *Software Consultancy*. 2010.
[55] Cyanogenmod. *Full Update Guide - Nexus One Firmware to CyanogenMod. 2010.*
[56] Forum xda developers. *Dalvik with JIT enabled for CyanogenMod 5.0.5.x*. 2010.
[57] OSGI. *The Dynamic Module System for Java*. 2010.

konsultering inom mjukvarudesign och strategiska affärsbeslut. I deras kundkrets ingår multinationella företag såsom Intel, Ericsson, Adobe, OSGi Alliance, Deutsche Telekom, IBM och Telia. Vår bedömning är därför att tillförlitligheten för verktyget clsd är hög. För att ytterligare säkerställa tillförlitliga resultat duplicerade vi test utförda av Google. Om vårt utfall överensstämmer med Googles bör det inte finnas något tvivel för erhållna resultat. Det exempel vi avsåg att duplicera hade följade källkod[58]

```
public interface Zapper {
        public String zap(String s, Object o);}

public class Blort implements Zapper {
        public String zap(String s, Object o) {...;}}

public class ZapUser {
        public void useZap(Zapper z) {z.zap(...);}}
```

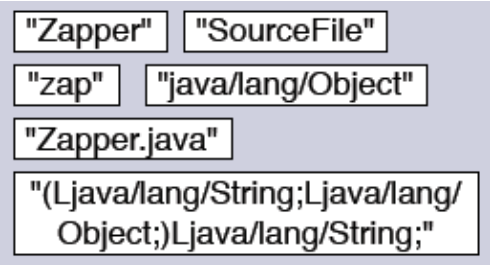
För att kunna kompilera koden var vi tvungna att ersätta ”…” med kompilerbar kod. Följande kod ersätter punkterna i klass Blort och ZapUser.

```
Blort: return s
ZapUser: null,null
```

5.4.2 Verifiering av utdata

Figurerna nedan är hämtade från Google officiella presentation av Dalvik VM[59]. Figurerna jämförs med den utdata vi fick av verktyget ”clsd”.

Google Dalvik_VM Internals

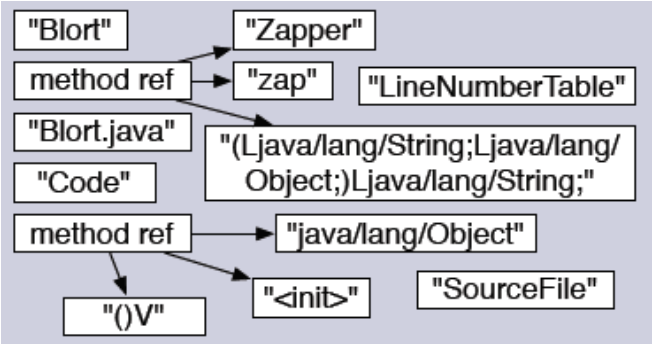


Figur 5.4.2.1: Constant pool av class Zapper.

aQute class dump utility (clsd)

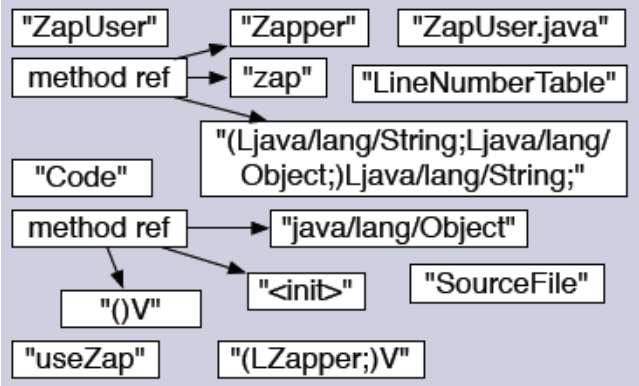
```
1 tag(7) constant classs 7
2 tag(7) constant classs 8
3 tag(1) utf8 'zap'
 4 tag(1) utf8
'(Ljava/lang/String;Ljava/lang/Objec
t;)
Ljava/lang/String;'
5 tag(1) utf8 'SourceFile'
6 tag(1) utf8 'Zapper.java'
7 tag(1) utf8 'Zapper'
8 tag(1) utf8 'java/lang/Object'
```

[58] Bornstein Dan. *Presentation Of Dalvik VM Internals*. Sida 16. 2008.
[59] Bornstein Dan. *Presentation Of Dalvik VM Internals*. Sida 17. 2008.



Figur: 5.4.2.2: Constant pool av class Blort.

1 tag(10) method ref 3/13
2 tag(7) constant classs 14
3 tag(7) constant classs 15
4 tag(7) constant classs 16
5 tag(1) utf8 '<init>'
6 tag(1) utf8 '()V'
7 tag(1) utf8 'Code'
8 tag(1) utf8 'LineNumberTable'
9 tag(1) utf8 'zap'
10 tag(1) utf8
'(Ljava/lang/String;Ljava/lang/Obje
ct;)
Ljava/lang/String;'
11 tag(1) utf8 'SourceFile'
12 tag(1) utf8 'Blort.java'
13 tag(12) name and type 5/6
14 tag(1) utf8 'Blort'
15 tag(1) utf8 'java/lang/Object'
16 tag(1) utf8 'Zapper'



Figur 5.4.2.3: Constant pool av class ZapUser.

1 tag(10) method ref 4/13
2 tag(11) interface and method ref
14/15
3 tag(7) constant classs 16
4 tag(7) constant classs 17
5 tag(1) utf8 '<init>'
6 tag(1) utf8 '()V'
7 tag(1) utf8 'Code'
8 tag(1) utf8 'LineNumberTable'
9 tag(1) utf8 'userZap'
10 tag(1) utf8 '(LZapper;)V'
11 tag(1) utf8 'SourceFile'
12 tag(1) utf8 'ZapUser.java'
13 tag(12) name and type 5/6
14 tag(7) constant classs 18
15 tag(12) name and type 19/20
16 tag(1) utf8 'ZapUser'
17 tag(1) utf8 'java/lang/Object'
18 tag(1) utf8 'Zapper'
19 tag(1) utf8 'zap'
20 tag(1) utf8'
(Ljava/lang/String;Ljava/lang/Object
;)
Ljava/lang/String;'

Då Google inte presenterade sitt resultat i textform var vi tvungna att jämföra vårt textbaserade resultat med figurerna ovan. Figurerna illustrerar de metodnamn, objekt,

variabler etc. som varje klass innehåller och hur de refererar till varandra. Därav kan man ganska lätt jämföra figurerna med det textbaserade resultatet verktyget clsd gav oss. Utan att beskriva varje figur med tillhörande textresultat i detalj kan följande exempel illustrera hur resultatet bör tolkas.

I *figur 5.4.2.3 Constant pool av class ZapUser*, kan vi se att den översta metod-referensen pekar på ”Zapper”, ”zap” och ”(Ljava/lang/String;Ljava/lang/Object;)Ljava/lang/String”. Nu söker vi samma resultat i det textbaserade utdatan verktyget ”clsd” gav oss. På rad två i det textbaserade resultatet hittar vi, ”2 tag(11) interface and method ref 14/15” vilket pekar vidare på rad 14 och 15. Rad 14 hänvisar sedan vidare till rad 18 som innehåller klassen ”Zapper”. Rad 15 hänvisar även den vidare till rad 19 och 20 som innehåller metoden ”zap” respektive ett objekt av klassen String vilket var det vi sökte.

6. Resultat

6.1. Experiment 1: Minimal redudans

Tabell 6.1.1 är uppdelad i tre kolumner, första kolumnen innehåller den information vi söker i class och dex-filerna medan kolumn två och tre visar hur informationen är uppdelad. Detta är bara en liten del av den information som finns i tillänglig. Informationen nedan är tagen för att belysa de skillnader constant pool på program-nivå ger.

Redundant information	**3 classfiler innehåller**	**1 dexfil innehåller**
Metod namn (”sumArray”)	3	1
Metodens return typ ”()V”	3	1
'[Ljava/lang/String;'” String array (lista)	2	1
Strängen ’THE SIZE OF THE LIST IS’	2	1
Ett objekt av printStream ”Ljava/io/PrintStream;”	3	1
Filheader (sammanfattning av vad filen innehåller)	3	1

Tabell 6.1.1: Jämförelse för redundans mellan class och dex-filer.

Resultatet av *tabell 6.1.1* visar att redundans klasserna emellan har upphört att existera då Dalvik VM har en constant pool på program-nivå. De tre klassfilerna som har identiska metodnamn, arraynamn och strängnamn som sparas lokalt i varje class-fil. Vid kompilering till dex-fil sammanfogas klasserna så att redundansen minimeras samtidigt som funktionaliteten bibehålls.

Tabell 6.1.2 är uppdelat i fem kolumner. Första kolumnen visar de program som har används för experimentet medan resterande kolumner visar filstorleken för respektive format. Procentsatsen är i förhållande till storleken av class-filerna.

Storlek av classfiler	Storleken av komprimerade classfiler (jar)	Storleken av en dexfil	Strolek av en komprimerad dexfil (apk)
Klassfilerna i tabell 6.1.1: 2,39 kB	2,18 kB – 8.6%	1,63 kB – 31.6 %	0,99 kB – 59.5 %
Aes: 18,1	11,5 kB – 36.4 %	10,0 kB – 44.5 %	6,25 kB – 65.5 %
Zulu: 16,9	10,7 kB – 37 %	13,1 kB – 22.6 %	6,64 kB – 60.1 %
Dam: 10,3 kB	7,40 kB – 28.2 %	8,43 kB – 18.3 %	4,35 kB – 57.8 %
TOTAL 47,7 kB	31,8 kB – 33.5 %	33,1 kB – 30.5 %	18,23 kB – 61.9 %

Tabell 6.1.2: Jämförelse för filstorlek mellan class, jar, dex och apk-filer.

En okomprimerad dex-fil har i snitt nästan samma filstorlek som en komprimerad jar-fil. Apk-filer är överlag ca 60% mindre än en class-fil medan liknande jämförelse för en jar-fil bara är ca 30%. Att hålla constant pool på den högre program-nivån verkar ha gett resultat. Den minskade filstorleken kan göra vässäntlig skillnad för enheter med begränsat lagringsutrymme.

6.2. Experiment 2: Reducerad opcode

Tabell 6.2.1 nedan är uppdelad i tre kolumner, första kolumnen visar de metoder som används medan kolumn två och tre visar antalet opcode instruktioner för class respektive dex-filer. Ett flertal metoder har inre loopar, antalet instruktioner innanför looparna visas inom parantes efter antalet instruktioner för metoden.

Metod namn Returnvärde metodnamn(parameter)	Antal instruktion (class)	Antal instruktion (dex)
void clear()	22	14
int getMoves()	5	3
Piece getPiece(int)	42	36
Piece getPiece(String, String)	64 (loop : 60)	44 (loop: 41)
int getSquare(String)	28	23
String getTurn()	5	3
boolean movePiece(int, int)	444	334
void addPiece(int, Piece)	66	51
void setup()	122 (loop: 118)	85 (loop: 80)
pawn		
HashSet<Integer> moves()	120 (loop: 87)	78 (loop: 57)
HashSet<Integer> captures()	272 (loop: 237)	174 (loop: 137)
Piece		
boolean move(int)	37	35
boolean hasPattern(int)	39 (loop: 24)	18 (loop: 11)
HashSet<Integer> moves()	125 (loop: (111\91))	71 (loop: (50\30))
HashSet<Integer> captures()	272 (loop: 237)	174 (loop: 137)
boolean shouldPomote()	48	19
Totalt	**1711** **(loop: 874)**	**1161 (32%mindre)** **(loop: 513** **(41%mindre))**

Tabell 6.2.1: Jämförelse för antalet opcode instruktioner mellan jar och dex-filer.

Resultatet visar att antalet instruktioner kraftigt har reducerats med registerbaserad bytekod. Antalet instruktioner har totalt sett minskat med 32% och för loopar 41%. Då loopar i regel körs mer än en gång innebär detta att det verkliga resultatet troligtvis är någonstans mellan 32-41%. Detta är bara en uppskattning då det är svårt att veta hur många gånger varje enskild loop kommer att köras.

6.3 Experiment 3: JIT-kompilator.

Java VM använder en JIT-kompilator för att förbättra prestandan under programkörning. *Tabell 6.3.1* visar att Java VM kraftigt förbättrat sitt resultat med hjälp av JIT-kompilatorn medan Dalvik VM inte får samma prestandahöjning.

Experiment	Java Virtual machine		Dalvik Virtual machine	
	Med JIT	Utan JIT	Med JIT	Utan JIT
Fibonacci (35 fibonacci serier)	125 ms (14,4 gånger snabbare)	1804 ms	2532 ms (3.3 gånger snabbare)	8304 ms
Beräkna π med 10000 decimaler	9652 ms (7,9 gånger snabbare)	76498 ms	10075 ms (1.2 gånger snabbare)	12065 ms
Flops (flyttalsoperationer)	2904,38 MFlops\s (31,6 gånger snabbare)	91.89MFlops\s	18,46 MFlops\s (2,9 gånger snabbare)	6.28 MFlops\s
Genomsnitt	18 gånger snabbare		2,5 gånger snabbare	

Tabell 6.3.1: Hastighetsjämförelse mellan Java och Dalvik VM med JIT-kompilator aktiverad eller avaktiverad.

Prestandahöjningen varierar kraftigt mellan de olika testprogrammen och de virtuella maskinerna. Den genomsnittliga hastighetsökningen för Java VM med JIT kompilatorn aktiverad är 18 gånger hastigheten, motsvarade siffra för Dalvik är endast 2,5 gånger snabbare.

För att ytterligare förtydliga skillnaden visar *diagram 6.3.2* samma resultat fast i diagramform. Staplarna illustrerar prestandaförbättringen JIT-kompilatorerna ger för Java och Dalvik VM.

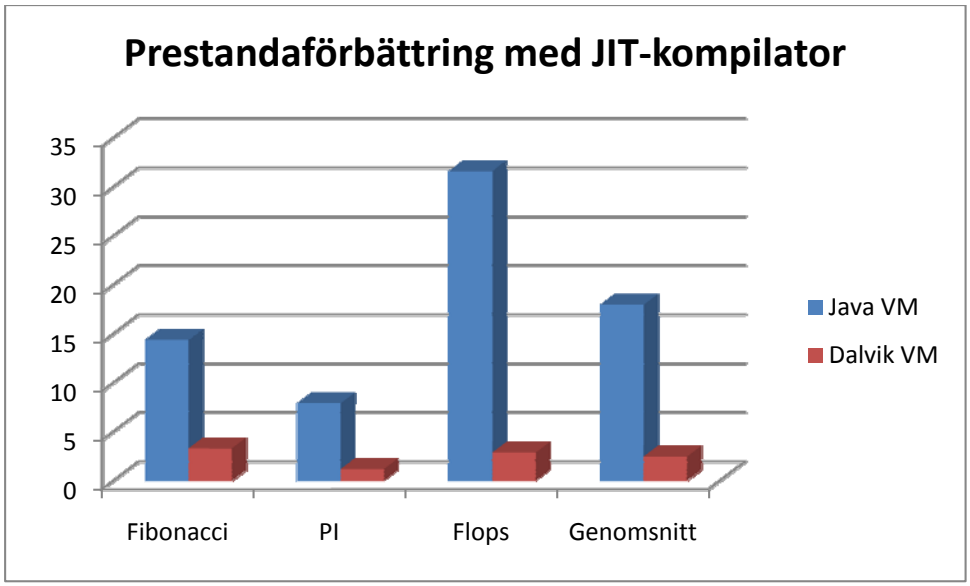


Diagram 6.3.2: Jämförelse för prestandaförbättringen mellan JIT-kompilatorn i Java och Dalvik VM.

Här kan vi tydligt se hur JIT-kompilatorn i Java VM ger betydande prestandaförbättringar under programkörning. Dalvik VM presterar bättre med JIT-kompilatorn aktiverad men prestandaförbättringen är bara en bråkdel av den förbättring vi ser i Java VM.

7. Diskussion

Dalvik är en mycket kompetent och väl fungerande VM som anpassats för mobila enheter. Anpassningar har gjorts för att förbättra prestandan för den relativt svaga hårdvara mobila enheter använder sig av.

Optimeringen av den kod som ska köras på Dalvik VM börjar redan vid installationsprocessen. Validering och optimering vid installationen säkerställer att koden inte är skadlig för systemet samt att koden exekveras på snabbast möjliga sätt. Med Java VM görs denna process vid applikationsexekvering då Java-applikationer inte har någon installationsprocess. För Dalvik VM innebär detta lägre energiförbrukning samt snabbare exekveringstid då validering samt optimering bara görs vid installationen.

Dalvik VM körs på enheter med kraftigt begränsat lagringsutrymme. Att effektivisera användandet av lagringsutrymmet är därför av största vikt. Effekten av att använda constant pool på program-nivå istället för klass-nivå har kraftigt reducerat filstorleken. Att resultatet är så gynnsamt är något som förvånar oss. Trots att Dalvik VM använder sig av en registerbaserad arkitektur minskar filstorleken med 30% då kompilering sker fån class till dex-filer. Det utökade lagringsutrymmet gör det möjligt att installera fler applikationer samt att kunna köra flera program samtidigt utan att behöva avsluta processer. Hur stor effekt denna reducering av filstorlek har är däremot svårare att bedöma då de experiment vi utfört inte tar hänsyn till flera faktorer. Dagens program är sällan textbaserade, det är mer regel än undantag att programmen har grafiska gränssnitt med tillhörande resurser i form av ljud och bild. Dessa resurser kan ta upp en stor del av applikationernas totala minnesanvändning, därmed minskar betydelsen av den reducerade filstorleken hos bytekoden. Oavsett hur stor den slutgiltiga effekten blir för användaren är en reducerad filstorlek aldrig negativ. Att hushålla med resurserna är alltid någon man bör eftersträva.

Stackbaserad bytekod har länge använts av bekvämlighetssjäl. Det är lättare att utveckla en kompilator som genererar stackbaserad bytekod samtidigt som bytekoden tar upp mindre plats[60]. Att Google därför valt att använda en registerbaserad arkitektur visar att de var beredda att offra en del lagringsutrymme för bättre prestanda. Filstorleken reduceras redan kraftigt tack vare att de valt att slå ihop class-filerna till en dex-fil och därmed använda constant pool på program-nivå. En annan mycket trolig anledning till att Google valt en registerbaserad arkitektur kan vara för att förbereda Dalvik VM för en framtida JIT-kompilator. Processorer är idag registerbaserade, därför bör även maskinkoden vara registerbaserad. En JIT-kompilator bör därför kunna implementeras utan större besvär samtidigt som den är mindre resurskrävande[61]. Ett mycket förenklat sätt att se på det hela är att en del av jobbet redan är gjort innan JIT kompilatorn börjar arbeta. Resultatet i experiment 2 visar att registerbaserad bytekod genererar färre instruktioner än den stackbaserade bytekoden för Java VM. Därmed har Google lyckats få ner antalet instruktioner vilket var målet med övergången från den stackbaserade bytekoden. Hur mycket snabbare den

[60] Squawks of the Parrot. Registers vs stacks for interpreter design. 2003.
[61] Paller Gabor. *Understanding the Dalvik bytecode with the Dedexer tool*. (sid 16). 2009.

reducerade bytekoden kan exekveras är något våra tester lämnar obesvarat då det i dagsläget inte finns någon möjlighet att köra stackbaserad bytekod på Dalvik. Effekten av att koden kan köras snabbare bör förlänga batteritiden då processorbelastningen sjunker när koden kan köras mer effektivt.

Den version av JIT-kompilator som idag finns implementerad i Dalvik VM är inte den färdiga versionen som kommer att inkluderas med framtida versioner av Android. Resultaten vi visar kan därför komma att skilja sig mot den slutgiltiga version vilket medför att de slutsatser vi drar här kan komma att revideras. Google hävdar under sin officiella presentation av Dalvik VM att en JIT-kompilator inte ger samma prestandavinst som för Java VM pga. att många av deras bibliotek redan är skrivna i maskinkod samt att Dalvik redan är relativt snabb. De experiment vi utfört stärker denna uppfattning då JIT-kompilatorn i dagsläget inte ger samma prestandahöjning som för Java VM. JIT-kompilatorn ger ändå betydande prestandahöjning vilket innebär att den har ett potentiellt värde för Dalvik VM även om en JIT-kompilatorn teoretiskt sätt kan försämra batteritiden då den extra kompileringen är resurskrävande. Detta kan emellertid vägas upp av den snabbare exekveringstiden då processorn tidigare kan lägga sig i viloläge efter utfört arbete. Resultaten av våra tester ger inget svar angående batteritiden varvid det därför är svårt att dra mer konkreta slutsatser. Den prestandahöjning vi sett av experimenten vi genomfört samt det faktum att Google redan har implementerat en experimentell version av JIT-kompilator i Dalvik VM leder oss till den slutsatsen att Google alltid haft för avsikt att implementera en JIT-kompilator för Dalvik. Den ökade prestandan ger betydande förbättringar även om de inte är lika omfattande som för Java VM. Mer avancerade applikationer kommer att dra nytta av den förhöjda prestandan och därmed ge utvecklare större utrymme att implementera nya funktioner.

8. Slutsatser

Dalvik är en VM som från grunden byggts upp och designats för mobila enheter. En mobiltelefon idag har kraftigt reducerad minneskapacitet och processorstyrka jämfört med dagens datorer vilket skapar helt andra förutsättningar för den kod som ska exekveras. Av dessa anledningar har Google tillsammans med Open Handset Alliance valt att skapa en helt ny VM som anpassats för begränsad energikonsumtion, minneskapacitet samt processorkraft. I dagsläget finns det ingen möjlighet att testa Java VM på Android eller Dalvik VM på en stationär dator. Det kan därför vara svårt att göra en rättvis bedömning för hur Dalvik VM presterar jämfört med Java VM. Resultaten av de experiment vi utfört samt en djupdykning i Dalvik och Java VMs arkitektur har emellertid övertygat oss att Dalvik är bättre anpassat för mobila enheter.

Dalvik VM har designats från kompileringsprocessen till applikationsexekvering för att på bästa sätt använda de resurser en mobil enhet förfogar. Alla steg på vägen mellan kompilering till exekvering har något att bidra med för att göra Dalvik VM så pass effektiv som möjligt. Filstorleken har reducerats, validering och optimering görs under installationsprocessen samt att koden kan köras snabbare pga en registerbaserad arkitektur. Allt detta gör Dalvik till en mycket effektiv och specialanpassad VM för mobila enheter.

Då Dalvik fortfarande är en relativt ung VM innebär detta att utvecklingen alltjämt sker i rasande fart. Stora förändringar är att vänta som i bästa fall kan flerdubbla prestandan. Experimentella test vi utfört visar redan nu att en JIT-kompilator kombinerat med Dalvik VM genererar imponerande resultat. Hur vidare den slutgiltiga versionen levererar förväntat resultat återstår att se. Vad vi kan säga idag är Google tillsammans med Open Handset Alliance skapat en VM som presterar utöver förväntan och som anpassats väl för de begränsade resurser en mobil enhet kan tillhandahålla. De arkitektiska beslut som tagits är därmed väl motiverade för att applikationer på Android ska ha så bra förutsättningar som möjligt.

9. Källförteckning

1. Amcgowan (2010). *Newton's Method for finding roots*. 4 april 2010 [www]. Hämtat från <http://www.amcgowan.ca/blog/computer-science/newtons-method-for-finding-roots/> 26 april 2010.

2. Anandtech (2009). *AnandTech Tests GPU Accelerated Flash 10.1 Prerelease*. 11 november 2009 [www]. Hämtat från <http://www.anandtech.com/show/2876> 22 april 2010.

3. Appleinsider (2009). *Iphone rim taking over smartphone market*. Canalus Q3 den 3 November 2009 [www]. Hämtat från <http://www.appleinsider.com/articles/09/11/03/canalys_q3_2009_iphone_rim_taking_over_smartphone_market.htm> 20 april 2010.

4. Aqute (2010). *Software Consultancy*. [www]. Hämtat från <http://www.aqute.biz/Main/HomePage> 8 april 2010.

5. Blackberry (2010). *Developers*. [www]. Hämtat från <http://na.blackberry.com/eng/developers/> 13 mars 2010.

6. Blackberrysync (2009). *Blackberry dominating U.S. smartphone market shares.* juni 2009 [www]. Hämtat från <http://blackberrysync.com/2009/06/blackberry-dominating-us-smartphone-market-shares/> 21 april 2010.

7. Bloomberg (2007). *Apple's IPhone Generates Buzz That May Top Mustang*. 28 juni 2007 [www]. Hämtat från <http://www.bloomberg.com/apps/news?pid=20601087&sid=ajgLmjh2Ukzo&refer=home> 21 april 2010.

8. Bornstein Dan (2008). *Presentation Of Dalvik VM Internals*. Googles officiella presentation den 29 maj 2008 [www]. Hämtat från <http://sites.google.com/site/io/dalvik-vm-internals> 19 april 2010.

9. Canalys (2009). Press and research releases. [www]. Hämtat från <http://www.canalys.com/pr/2009/r2009112.html> 20 april 2010.

10. Capitalhead (2006). *Benchmarking VMware ESX Server 2.5 vs Microsoft Virtual Server 2005 Enterprise Edition*. 19 april 2006 [www]. Hämtat från <http://capitalhead.com/articles/benchmarking-vmware-esx-server-25-vs-microsoft-virtual-server-2005-enterprise-edition.aspx> 21 april 2010.

11. Cnet (2008). *iPhone 3G crowned most popular phone in U.S.* 10 november 2008 [www]. Hämtat från <http://news.cnet.com/8301-1035_3-10092629-94.html> 20 april 2010.

12. Cyanogenmod (2010). *Full Update Guide - Nexus One Firmware to CyanogenMod.* 20 april 2010 [www]. Hämtat från <http://wiki.cyanogenmod.com/index.php/Full_Update_Guide_-_Nexus_One_Firmware_to_CyanogenMod> 26 april 2010.

13. Electronista (2009*). Ballmer om win mobile 6.5*. 5 mars 2009 [www]. Hämtat från <http://www.electronista.com/articles/09/03/05/ballmer.on.win.mobile.6.5/> 20 april 2010.

14. Forum xda developers (2010). *Dalvik with JIT enabled for CyanogenMod 5.0.5.x.* 26 mars 2010 [www]. Hämtat från <http://forum.xda-developers.com/showthread.php?t=652911> 16 april 2010.
15. Geeky gadgets (2010). *Android Fastest Growing Smartphone OS.* 25 mars 2010 [www]. Hämtat från <http://www.geeky-gadgets.com/android-fastest-growing-smartphone-os-25-03-2010/> 1 april 2010.

16. Google (2010). *A JIT Compiler for Android's Dalvik VM*. [www]. Hämtat från <http://code.google.com/intl/sv-SE/events/io/2010/sessions/jit-compiler-androids-dalvik-vm.html> 4 april 2010.

17. Google (2010). *Speakers @ Google I/O*. [www]. Hämtat från <http://code.google.com/intl/sv-SE/events/io/2010/speakers.html> 22 april 2010.

18. Googleblog (2008). The first Android-powered phone. 23 september 2008 [www]. Hämtat från <http://googleblog.blogspot.com/2008/09/first-android-powered-phone.html> 24 april 2010.

19. H-online (2009). *Android's Dalvik to be JIT boosted*. 17 november 2009 [www]. Hämtat från <http://www.h-online.com/open/news/item/Android-s-Dalvik-to-be-JIT-boosted-861870.html> 22 april 2010.

20. Java (2010). *Learn About Java Technology*. [www]. Hämtat från <http://java.com/en/about/> 28 mars 2010.

21. Java Sun (2010). *Java Archive (JAR) Files*. [www]. Hämtat från <http://java.sun.com/javase/6/docs/technotes/guides/jar/index.html> 25 april 2010.

22. Langpop (2010). *Programming Language Popularity*. 22 april 2010 [www]. Hämtat från <http://www.langpop.com/> 27 april 2010.

23. Mobiletechworld (2009). *HTC HD2 review the best smartphone ever. 18 december 2010 [www]*. Hämtat från <http://www.mobiletechworld.com/2009/12/18/htc-hd2-review-the-best-smartphone-ever/comment-page-1/> 20 april 2010.

24. Netmite (2008). *Dalvik optimization and verification with dexopt*. [www]. Hämtat från <http://www.netmite.com/android/mydroid/dalvik/docs/dexopt.html> 21 april 2010.

25. Nexus404 (2010). *Nokia Symbian v3 Phones Coming in Q3 2010*. 2 februari 2010 [www]. Hämtat från <http://nexus404.com/Blog/2010/02/02/nokia-symbian-v3-phones-coming-in-q3-2010-symbian-v4-arriving-too-new-nokia-smartphones-to-run-symbian-v3-v4-by-the-end-of-the-year/> 18 april 2010.

26. Nytimes (2009). *Challenging Microsoft With a New Technology*. 9 augusti 2009 [www]. Hämtat från <http://www.nytimes.com/2009/08/31/technology/business-computing/31virtual.html?_r=4&pagewanted=2&partner=rss&emc=rss> 21 april 2010.

27. Open handset alliance (2010). *Developers*. [www]. Hämtat från <http://www.openhandsetalliance.com/developers.html> 15 mars 2010.

28. Oracle (2010). Oracle's *Support for Open Source and Open Standards*. [www]. Hämtat från <http://www.oracle.com/us/technologies/open-source/index.htm> 21 april 2010.

29. OSGI (2010). *The Dynamic Module System for Java*. [www]. Hämtat från <http://www.osgi.org/Main/HomePage> 13 april 2010.

30. Paller Gabor (2009). *Understanding the Dalvik bytecode with the Dedexer tool*. 2 december 2009 [www]. Hämtat från< http://www.slideshare.net/paller/understanding-the-dalvik-bytecode-with-the-dedexer-tool> 29 april 2010.

31. Squawks of the Parrot (2003). Registers vs stacks for interpreter design. 14 maj 2003 [www]. Hämtat från< http://www.sidhe.org/~dan/blog/archives/000189.html> 28 april 2010.

32. Reitmaier Rick (2006). *Adobe Flash Player ActionScript Virtual Machine*. Adobe presentation den 6 december 2006 [www]. Hämtat från <http://www.docstoc.com/docs/885533/Adobe-Flash-Player-ActionScript-Virtual-Machine---Tamarin/> 21 april 2010.

33. Sectordata (2008). *Om java* [www]. Hämtat från <http://sectordata.se/laromedel/java/javasite4_02.html> 19 april 2010.

34. Sourceforge (2009). *Java Benchmark (Test PC).* 24 december 2009 [www]. Hämtat från <http://sourceforge.net/projects/opt/> 25 april 2010.

35. Stevens Ashley (2009). *Developing for Android on ARM*. [www]. Hämtat från <http://arm.dgfactor.net/F1_Developing%20Android_Ashley%20Stevens_20091119.pdf> 14 april 2010.

36. Symbian developer (2010). *Symbian Documentation*. [www]. Hämtat från < http://developer.symbian.org/main/documentation/index.php > 23 april 2010.

37. The shape of code (2009). *Register vs. stack based VMs.* 17 september 2009 [www]. Hämtat från <http://shape-of-code.coding-guidelines.com/2009/09/ >15 paril 2010.

38. Theregister (2010). *Adobe apologizes for festering Flash crash bug*. 9 february 2010 [www].Hämtat från <http://www.theregister.co.uk/2010/02/09/adobe_flash_crash_bug/> 21 april 2010.
39. Theregister (2010*). Steve Jobs dubs Google's 'don't be evil' motto 'bulls**t'*. 1 february 2010 [www].Hämtat från <http://www.theregister.co.uk/2010/02/01/steve_jobs_on_google/> 21 april 2010.
40. Vivisectingmedia (2007). *Flash Internals: The ActionScript Virtual Machine*. 7 November 2007 [www]. Hämtat från <http://blog.vivisectingmedia.com/2007/11/flash-internals-the-actionscript-virtual-machine-part-three/> 21 april 2010.
41. Vmware (2007). *A Performance Comparison of Hypervisors*. [www]. Hämtat från <http://www.vmware.com/pdf/hypervisor_performance.pdf> 21 april 2010.

42. Vmware (2007). *The Architecture of VMware ESXi*. [www]. Hämtat från <http://www.vmware.com/files/pdf/vmware_esxi_architecture_wp.pdf> 21 april 2010.

43. Webbmatte (2010). *Fibonaccis talföljd*. [www]. Hämtat från <http://www.webbmatte.se/display_page.php?id=60&on_menu=366&page_id_to_fetch=939&lang=arabic&no_cache=99423554> 26 april 2010.

44. Wikipedia (2010). *Gauss–Legendre algorithm*. 22 april 2010 [www]. Hämtat från <http://en.wikipedia.org/wiki/Gauss%E2%80%93Legendre_algorithm> 25 april 2010.

45. Wikipedia (2010). *Static library*. 1 april 2010 [www]. Hämtat från <http://en.wikipedia.org/wiki/Static_library> 25 april 2010.

46. Wikipedia (2010). Virtual machine. 22 april 2010 [www]. Hämtat från <http://en.wikipedia.org/wiki/Virtual_machine> 25 april 2010.

47. Wikipedia (2010). *Virtual method table*. 15 april 2010 [www]. Hämtat från <http://en.wikipedia.org/wiki/Vtable> 25 april 2010.

Böcker

48. Burnette Ed. (2010). *Hello,Android introducing Google's mobile development platform.* Upplaga 3. Los Angeles, Beta Books.

49. Popek J Gerald. (1974) *Formal requirements for virtualizable third generation architectures.* Upplaga 7. New York, ACM.

Bilagor

Följande bilagor tillhör de experiment vi utfört. All källkod inklusive dump-filer kommer även att läggas upp på sidan:

http://www.f.kth.se/~hindi/dkand10/

Bilaga A: Källkoden till experiment 1

- Klassfil 1: Tar in en lista av tal som parameter och skriver ut summan av listan
- Klassfil 2: Innehåller en metod som tar in en lista av strängar och skriver ut antal strängar i listan.
- Klassfil 3: Innehåller en metod som tar in en lista av strängar, lägger ihop samtliga strängar till en sträng, skriver ut strängen samt det ursprungliga antalet strängar.

```
public class klassfil1 {
          public static void sumArray(int[] arr){
                    long sum = 0;
                    String resultat ="The sum is :";
                    for(int i : arr){
                              sum+=i;
                    }System.out.println(resultat+sum);

          }}


public class klassfil2 {
          public static void sumArray(String[] arr){
                    String string =" THE SIZE OF THE LIST IS: ";
                    long size =0;
                    for(String str : arr){
                              size++;
                    }System.out.println(string+ size);

          }}

public class klassfil3 {
          public static void sumArray(String[] arr){
                    long size = 0;
                    String string ="THE SIZE OF THE LIST IS: ";
                    String listAppend = "";
                    for(String str : arr){
                              listAppend+=str;
                              size++;
                    }System.out.println("The appended list is "+
listAppend + " THE SIZE OF THE LIST IS: "+ size);

          }}
```

Bilaga B: Källkoden till experiment 2

- Board: Ritar upp spelbordet.
- Pawn: Hanterar spelpjäsernas förflyttningar.
- Piece: Objekt av spelpjäserna.

```
import java.io.*;
import java.util.*;

public class Board
{
          private static final boolean DEBUG = false;
          private boolean[] checkedSquare;

          private int moves;
    private String turn = "white";
    private Piece[] board;
    private ArrayList<String> history;
    private ArrayList<Piece> captures;

    public Board() {
        board = new Piece[128];
        checkedSquare = new boolean[128];
        history = new ArrayList<String>();
        captures = new ArrayList<Piece>();
        setup();

    }

    /**
     * Returns the amount of moves done.
     * @return the number of moves.
     */
    public int getMoves() {
        return moves;
    }

    public String getTurn() {
          return turn;
    }

    public ArrayList<String> getHistory() {
          return history;
    }

    public ArrayList<Piece> getCaptures() {
          return captures;
    }

    /**
```

```
     * Moves chosen piece to given square.
     * @param fromSquare
     * @param toSquare
     * @throws IllegalArgumentException if square is invalid.
     */
   public boolean movePiece(int fromSquare, int toSquare) throws
IllegalArgumentException {

          if (!(isValidSquare(fromSquare) &&
isValidSquare(toSquare)))
                throw new IllegalArgumentException("Invalid square
specified.");

        Piece piece = board[fromSquare];

        if (piece == null) throw new
IllegalArgumentException("Square's empty: " + fromSquare);
        if (!piece.color.equals(turn)) throw new
IllegalArgumentException("Must move " + turn + " piece");

        int prevNumMoves = moves;


        if (piece.move(toSquare)) {

          int diff =(toSquare-fromSquare);
          int captured=toSquare;
          if(diff==30) captured-=15;
          if(diff==34) captured-=17;
          if(diff==-30) captured+=15;
          if(diff==-34) captured+=17;

          System.out.println("Skillnaden is:" +(fromSquare-
toSquare));
          if (piece.captures().contains(captured))
                   captures.add(getPiece(captured));
          board[captured] = null;
            board[toSquare] = piece;
            board[fromSquare] = null;
            Piece piece2 =board[toSquare];
          if(!piece2.hasMoved()&& piece2.shouldPomote()){
                   piece.setMoved(true);

          }
            history.add(getSquare(fromSquare) + " " +
getSquare(toSquare));

            //
            if(((diff==30)||(diff==34)||(diff==-30)||(diff==-34))&&
(!piece2.captures().isEmpty())){
                moves++;
                }
          else{

                   nextTurn();
          }
```

```
        } else {
            throw new IllegalArgumentException("Invalid move: " +
                                                getSquare(fromSquare)
+ " to " +
                                                getSquare(toSquare) +
                                                (DEBUG ? ( ". (valid
moves for piece: " +

piece.moves() + ")") : ""));
        }
        if (DEBUG) {
            System.out.println("Moved " + piece.getName() +
                               " on square " + getSquare(fromSquare)
+
                               " to square " + getSquare(toSquare));
            System.out.println("It's now " + turn + "'s time to
move.");
        }

        return moves > prevNumMoves;
    }


    public boolean isCheckedSquare(int square) {
          return checkedSquare[square];
    }


    public boolean[] getCheckedSquares() {
          return checkedSquare;
    }
    public void setCheckedSquares(boolean[] to) {
          checkedSquare = to;
    }
    public void setCheckedSquare(int square) {
          checkedSquare[square] = true;
    }
    public void unsetCheckedSquares() {
          checkedSquare = new boolean[128];
    }

    /**
     * Returns the piece on specified board square.
     * @param square
     * @return the piece on given square.
     * @throws IllegalArgumentException if square is invalid.
     */
    public Piece getPiece(int square) throws
IllegalArgumentException {
        if (!isValidSquare(square)) throw new
IllegalArgumentException("Invalid square: " + square);
        return board[square];
    }

    public Piece getPiece(String name, String color) {
```

```
        for (int i = 0; i < 120; i++) {
            Piece piece = board[i];
            if (isValidSquare(i) && piece != null &&
                    piece.getName().equals(name) &&
                    piece.getColor().equals(color))
                return piece;
        }
        return null;
    }

    public int getFile(int square) {
        return (square % 8);
    }

    public int getRank(int square) {
        return (1 + square/16);
    }

    public int getSquare(String square) {
        int file = (int)square.charAt(0) - 97;
        int rank = Integer.parseInt(square.substring(1,2)) - 1;
        return (rank*16 + file);
    }

    public String getSquare(int square) {
        return (new String() + (char)((getFile(square))+ 97) +
getRank(square));
    }

    public void clear() {
        board = new Piece[128];
        turn = "white";
        moves = 0;
    }

    public boolean isEmpty(int square) {
        return getPiece(square) == null;
    }

    /**
     * Sets the pieces up for a classic game of checkers.
     */
    public void setup() {
        // pawns
        for (int i = 0; i < 8; i++) {
            addPiece(i, new Pawn(this, "white"));
            addPiece(i+17, new Pawn(this, "white"));
            addPiece(i+32, new Pawn(this, "white"));
            i++;
            addPiece(i+80, new Pawn(this, "black"));
            addPiece(i+95, new Pawn(this, "black"));
            addPiece(i+112, new Pawn(this, "black"));
        }

    }
```

```
    /**
     *
     * @param square
     * @param piece
     * @throws IllegalArgumentException if square is invalid.
     * @throws NullPointerException if piece is null
     */
    public void addPiece(int square, Piece piece)
            throws IllegalArgumentException {
        if (!isValidSquare(square))
            throw new IllegalArgumentException("Invalid square: " +
square);

        if (piece == null)
            throw new NullPointerException("Piece object was null");


        board[square] = piece;

        if (piece != null) piece.setCurrentSquare(square);

        if (DEBUG) {
            System.out.println("Added a " + piece.getColor() + " " +
                               piece.getName()+" to square
"+getSquare(square));
        }
    }

    public boolean isValidSquare(int square) {
        return ((square & 0x88) == 0);
    }

    private void nextTurn() {
        turn = (turn.equals("white") ? "black" : "white");
        moves++;
    }
}
```

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.util.HashSet;

public class Pawn extends Piece
{
    public Pawn(Board board, String color) {
        super(board, color);
        pattern = isWhite() ?
            new int[] { 15,17,-15,-17,30,34,-30,-34} : new int[] {-
15,-17,15,17,-30,-34,30,34,};
    }

    public Pawn(Board board, String color, String square) {
        super(board, color, square);
        pattern = isWhite() ?
            new int[] {15,17,30,34,-30,-34} : new int[] {-15,-17,-
30,-34,30,34,};
    }



    @Override
    public HashSet<Integer> moves() {

        HashSet<Integer> moves = new HashSet<Integer>();
        int size = 2; //can only move 2 squares on first move
        if(hasMoved()){
          size=4;
                    System.out.println("yes alredy moved");
          }
        for (int i = 0; i < size; i++) {
          //System.out.println("Square nummber is: "+square);
          int index = square + pattern[i];
          //System.out.println("Index nummber is: "+index);

          if (legalMoves.size() > 0 && !legalMoves.contains(index))
continue;

          if (!board.isValidSquare(index) ||
                                board.getPiece(index) != null) //
cannot move onto another piece) // cannot jump over another piece

                    continue;


                    moves.add(index);

        }
        return moves;
    }

    @Override
    public HashSet<Integer> captures() {
```

```
        HashSet<Integer> captures = new HashSet<Integer>();
        int size = 6; //can only move 2 squares on first move
        if(hasMoved()){
          size=8;

                    System.out.println("yes alredy moved");
          }
        for (int i = 4; i < size; i++) {
            int index = square + pattern[i];
            if ((index & 0x88) != 0)
                    continue;

          if (legalMoves.size() > 0 && !legalMoves.contains(index))
continue;
          Piece piece = board.getPiece(index);
            if (piece != null) continue;
            if (board.getFile(square) != board.getFile(index)) {

                    if(pattern[i]==30){
                              piece = board.getPiece(index-15);
                    }
                    if(pattern[i]==34){
                              piece = board.getPiece(index-17);
                    }
                    if(pattern[i]==-30){
                              piece = board.getPiece(index+15);
                    }
                    if(pattern[i]==-34){
                              piece = board.getPiece(index+17);
                    }

                    //System.out.println("maybe captures");
                if (piece != null &&
!piece.getColor().equals(color))
                    captures.add(index);

                continue;
            }
        }
        return captures;
    }

}
```

```
import java.util.HashSet;

public abstract class Piece
{
    protected Board board;
    protected int square;
    protected String color;
    protected int[] pattern; //FIXME
    protected boolean moved, sliding;

    public HashSet<Integer> legalMoves = new HashSet<Integer>();

    public Piece(Board board, String color) {
        this.board  = board;
        this.color  = color;
    }

    public Piece(Board board, String color, String square) {
        this(board, color);
        this.square = board.getSquare(square);

        board.addPiece(this.square, this);
    }

    protected Piece() {
          // for serialization to work.
    }

    public boolean hasPattern(int pattern) {
          for (int pat : this.pattern) {
                    if (pat == pattern) return true;
          }
          return false;
    }

    public void setLegalMoves(HashSet<Integer> moves) {
          legalMoves = moves;
    }
    public void unsetLegalMoves() {
          legalMoves.clear();
    }
    /**
     * Returns all valid moves for this piece.
     * FIXME: code duplication and perhaps slow.
     * @return set of valid moves.
     */
    public HashSet<Integer> moves() {

        HashSet<Integer> moves = new HashSet<Integer>();
        for (int i = 0; i < pattern.length; i++) {
            for(int index = square; (index & 0x88) == 0; index +=
pattern[i]) {

                    if (legalMoves.size() > 0 &&
!legalMoves.contains(index)) continue;
```

```
                Piece piece = board.getPiece(index);
                if (piece == this) continue;
                if (piece != null) break;
                moves.add(index);
                if (!isSliding()) break;
            }
        }
        return moves;
    }
    /**
     * Returns all valid captures for this piece.
     * FIXME: code duplication and perhaps slow.
     * @return set of valid captures for this piece.
     */
    public HashSet<Integer> captures() {
        HashSet<Integer> captures = new HashSet<Integer>();
        int size = 6; //can only move 2 squares on first move
        if(hasMoved()){
          size=8;

                  System.out.println("yes alredy moved");
          }
        for (int i = 4; i < size; i++) {
            int index = square + pattern[i];
            if ((index & 0x88) != 0)
                    continue;

          if (legalMoves.size() > 0 && !legalMoves.contains(index))
continue;
          Piece piece = board.getPiece(index);
            if (piece != null) continue;
            if (board.getFile(square) != board.getFile(index)) {

                    if(pattern[i]==30){
                              piece = board.getPiece(index-15);
                    }
                    if(pattern[i]==34){
                              piece = board.getPiece(index-17);
                    }
                    if(pattern[i]==-30){
                              piece = board.getPiece(index+15);
                    }
                    if(pattern[i]==-34){
                              piece = board.getPiece(index+17);
                    }

                    //System.out.println("maybe captures");
                if (piece != null &&
!piece.getColor().equals(color))
                    captures.add(index);

                continue;
            }
        }
        return captures;
    }
```

```
    public final String getColor() {
        return color;
    }
    public String getName() {
        String className = getClass().getName();
        return className.substring(5, className.length());
    }

    protected final boolean isWhite() {
        return this.color.equals("white");
    }

    public final void setCurrentSquare(int square) {
        this.square = square;
    }

    public final boolean move(int toSquare) {
        if (moves().contains(toSquare) ||
captures().contains(toSquare)) {
          setCurrentSquare(toSquare);
            //setMoved(true);
            return true;
        }
        return false;
    }
    public int getSquare() {
          return this.square;
    }
    public final void setMoved(boolean value) {
        moved = value;
    }
    public boolean shouldPomote() {
          if (color.equals("white") && square > 103) return true;
          else if (color.equals("black") && square < 16) return
true;
          return false;
    }

    /**
     * Returns a a string representation the this piece.
     * Usually the first letter of the class name
     * (this might be a stupid way to do it, but i digress).
     * @return a string representation the this piece.
     */
    public String toString() {
        String str = getClass().getName().substring(5,6);
        return isWhite() ? str : str.toLowerCase();
    }

    public final boolean isSliding() {
        return sliding;
    }

    public final boolean hasMoved() {
        return moved; }}
```

Bilaga C: Källkoden till experiment 3

- **Fibonnacci**: Mäter tiden i millisekunder som krävs för att beräkna ett visst Fibonnaccital. Algoritmen är baserad på Fibonnaccis formel.
- **PI:** Mäter tiden i millisekunder som krävs för att beräkna PI med ett visst antal decimaler. För att beräkna detta använder vi Salamin–Brent algoritmen för PI och Newtons metod för roten ur.

```
public class Fibo {
    public static void main(String args[]) {

                    int N = 40;
                    long startTime = System.currentTimeMillis();
                    System.out.println(fib(N));
                    long finishTime = System.currentTimeMillis();
                    System.out.println(finishTime - startTime);
    }
    public static int fib(int n) {
                    if (n < 2) return(1);
                    return( fib(n-2) + fib(n-1) );
    }
}
```

```
import java.math.BigDecimal;
import static java.math.BigDecimal.*;

class Pi_1 {
  private static int ROUND_MODE = ROUND_DOWN;
  private static final BigDecimal ZERO = BigDecimal.ZERO;
  private static final BigDecimal ONE = BigDecimal.ONE;
  private static final BigDecimal TWO = new BigDecimal(2);
  private static final BigDecimal FOUR = new BigDecimal(4);


  public static void main(String[] args) {
            long startTime = System.currentTimeMillis();
              System.out.println(Pi_1(Integer.parseInt(args[0])));
                    long finishTime = System.currentTimeMillis();

    System.out.println("Elapsed time: " +
                        (finishTime - startTime) + " msecs");
  }

  // Salamin-Brent Algorithm
  public static BigDecimal Pi_1(final int digits) {
    final int SCALE = 10 + digits;
    BigDecimal a = ONE;
    BigDecimal b = ONE.divide(sqrt(TWO, SCALE), SCALE, ROUND_MODE);
    BigDecimal t = new BigDecimal(0.25);
    BigDecimal p = ONE;
    BigDecimal tmp;

    while (!a.equals(b)) {
      tmp = a;
      a = a.add(b).divide(TWO, SCALE, ROUND_MODE);
      b = sqrt(b.multiply(tmp), SCALE);
      t =
t.subtract(p.multiply(tmp.subtract(a).multiply(tmp.subtract(a))));
      p = p.multiply(TWO);
    }

    return a.add(b)
      .multiply(a.add(b))
      .divide(t.multiply(FOUR), SCALE, ROUND_MODE)
      .setScale(digits, ROUND_MODE);
  }

  // square root method (Newton's)
  public static BigDecimal sqrt(BigDecimal A, final int SCALE) {
    BigDecimal x_0 = ZERO;
    BigDecimal x_1 = new BigDecimal(Math.sqrt(A.doubleValue()));

    while (!x_0.equals(x_1)) {
      x_0 = x_1;
      x_1 = A.divide(x_0, SCALE, ROUND_MODE);
      x_1 = x_1.add(x_0);
      x_1 = x_1.divide(TWO, SCALE, ROUND_MODE);
    }
    return x_1; }}
```