Examensarbete

1977
HÖGSKOLAN I SKÖVDE

ANALYS AV ANDROID-SPECIFIKA METODER FÖR ATT UPPNÅ BESTÄNDIGT TILLSTÅND

ANALYSIS OF ANDROID SPECIFIC METHODS FOR ACHEIVING PERSISTENT STATE

Examensarbete inom huvudområdet Datalogi
Grundnivå 30 högskolepoäng
Vårtermin 2013

Marcus Czövek

Handledare: Anna Syberfeldt
Examinator: Henrik Gustavsson

Sammanfattning

I detta arbete undersöks effektiviteten hos Androids medföljande metoder för att uppnå beständigt tillstånd. Undersökningen baseras på ett egenutvecklat verktyg för att mäta de olika metodernas effektivitet, med avseende på tid för att spara ner och läsa in tillståndet samt hur mycket utrymme det sparade tillståndet tar upp i sekundärminnet. Resultaten från undersökningen visar att de olika metoderna presterar olika bra beroende på tillståndets storlek samt om tillståndet mestadels består av text eller av nummer. Den metod som generellt är mest effektiv är Javas Serializable. Framtida undersökningar skulle kunna komplettera den genomförda undersökning genom att också utvärdera effektiviteten hos metoder som tillhandhålls via externa bibliotek.

Nyckelord: Android, Tidseffektiv, Serialisering, Tillstånd, SQLite

Innehållsförteckning

1 **Introduktion** 1

2 **Bakgrund** 2

2.1 Android 2
2.2 Beständigt tillstånd 3
2.3 Androids standardmetoder för att uppnå beständigt tillstånd 3
2.3.1 XML 3
2.3.2 JSON 4
2.3.3 Preferences 4
2.3.4 Javas Serializable 4
2.3.5 SQLite 4
2.4 Dalvik virtual machine 4
2.4.1 JIT 5
2.5 Flashdisk 5

3 **Problemformulering** 6

3.1 Problemspecifikation och syfte 6
3.2 Metodbeskrivning 6

4 **Genomförande** 9

4.1 Testmiljö 9
4.2 Testverktygets Implementation 9
4.2.1 Tidtagning 9
4.2.2 Testdata 10
4.2.3 Anpassa koden för DVM:en 12
4.2.4 Generella designval och implementation 12
4.2.5 Metodspecifika designval och implementation 13
4.3 Analys 16

5 **Slutsats** 20

5.1 Sammanfattning 20
5.2 Diskussion 20
5.3 Framtida arbeten 22

Referenser 24

1 Introduktion

Android har med tiden växt och blivit ett av de mest använda operativsystemen på mobila enheter (Android Developers. 2013a). Androids stora användarantal i kombination med en framgångsrik försäljningsplattform i form av Android market gör att Android är ett populärt operativsystem att utveckla applikationer och spel till. Enligt Android Developers (2013a) laddades det under första kvartalet 2013 ner ungefär 1.5 biljoner Androidapplikationer per månad.

När det gäller spelutveckling för Android är en viktig aspekt att implementera en väl fungerande tillståndshantering i spelet, då det finns många situationer i Android då spelet kan förlora sitt tillstånd (dvs. representationen av det nuvarande speltillståndet). En av anledningarna till detta är att Android avslutar processer när deras resurser anses behövas till andra högre prioriterade processer. Då processer som ligger i bakgrunden har lägre prioritet än processer i fokus innebär detta att Android har möjlighet att avallokera en spelsessions tillståndsdata ifall spelet hamnar i bakgrunden. Ett annat fall där ett spel kan förlora sitt tillstånd är när Androidenheten ändrar orienteringsläge. När detta händer så avallokeras all applikationsdata för att sedan direkt skapas på nytt i det nya orienteringslägget.

En väl fungerande tillståndshantering i spel är viktig för användarupplevelsen. Om tillståndet inte sparats korrekt så kommer spelet inte kunna återställa sig, och det kan också uppstå skärmfrysningar om sparningen och återställningen av tillståndet inte är tillräckligt tidseffektivt. Skärmfrysningar i spel kan vara speciellt frustrerande, då användaren förväntar sig korta reaktionstider. För att undvika problem är det viktigt att ett spels tillståndsdata sparas till sekundärminnet, så att speltillståndet senare kan återställas. Med denna typ av tillståndshantering kan spelet uppnå beständigt tillstånd, på engelska kallat ”persistent state”, vilket gör att spelets tillstånd alltid kommer att kunna återställas oavsett vad som händer.

I detta arbete undersöks och jämförs olika metoder i Android för att uppnå beständigt tillstånd. Syftet med arbetet är att identifiera effektiviteten hos de olika metoderna, och på så sätt underlätta valet av metod vid implemention av Androidspel. I jämförelsen ingår alla de metoder för att uppnå beständigt tillstånd genom att spara tillståndsdata till sekundärminnet som följer med Android som standard. Dessa metoder inkluderar XML, JSON, Java Serializable, Preferences och SQLite.

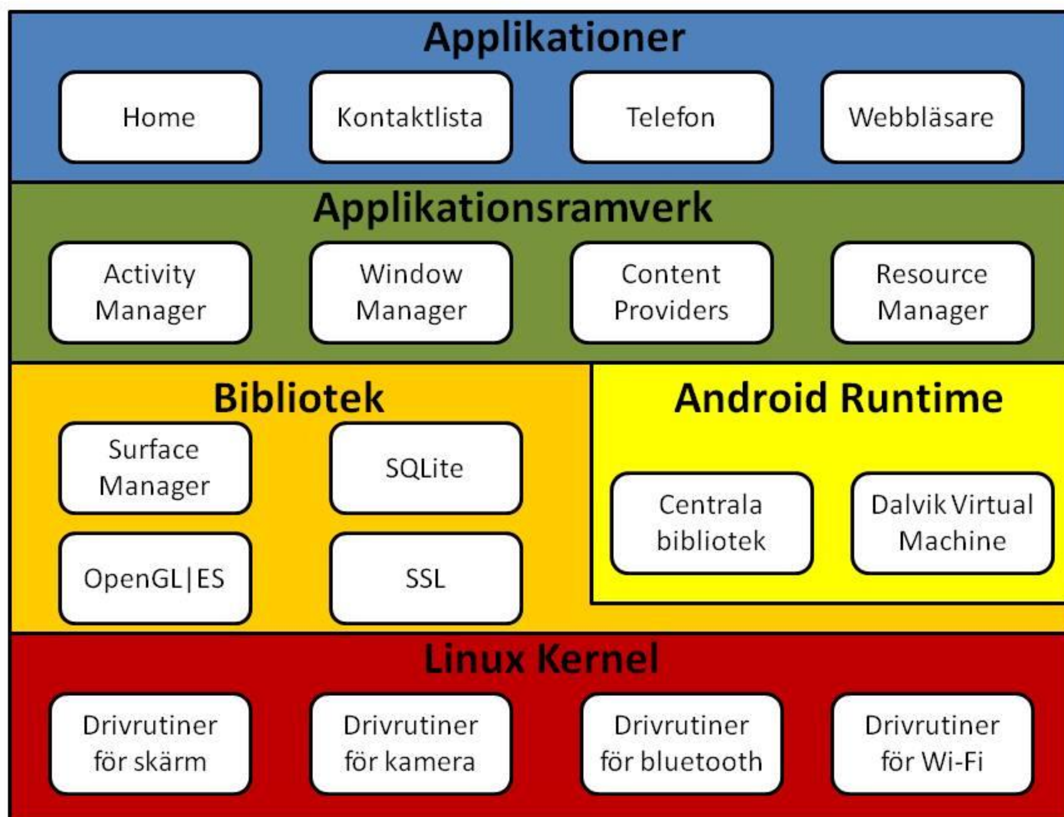
Rapporten är organiserad som följer. Kapitel 2 ger en bakgrund till arbetet genom att beskriva Android och de metoder som inkluderas i arbetet, samt aspekter som är relevanta för analysen av metoderna. Kapitel 3 beskriver tillståndsproblemet i detalj och den metod som används för att lösa det. Kapitel 4 ger en grundlig genomgång av implementationsprocessen och de resultat som erhållits presenteras. I kaptitel 5 presenteras en analys och diskussion av resultaten, samt möjliga framtida relaterade projekt presenteras.

2 Bakgrund

I detta kapitel presenteras relevant bakgrundsinformation för arbetet. Först beskrivs Androidplattformen. Därefter förklaras begreppet beständigt tillstånd och vad som måste utföras för att uppnå detta. Nästa del av kapitlet presenteras och förklarar de metoder som inkluderas i arbetet. I slutet av kapitlet förklaras olika faktorer som påverkar testningen av metoderna.

2.1 Android

Android är ett operativsystem för mobiler som är öppen källkod. Enligt Android Developers (2013a) är Android det mest använda mobila operativsystemet sett till antalet produkter som det är installerat på. I Sumaray & Kami Makki (2012) beskrivs Android som en stapel av mjukvara. I denna stapel inkluderas operativsystem, mellanprogramvara och ett antal applikationer. Nedan i figur 1 kan en bild ses som beskriver stapeln och dess fyra lager.



Figur 1: Androids mjukvarustapel (inspirerad från Sumaray & Kami Makki, 2012)

Sumaray & Kami Makki (2012) beskriver att Android använder en Linuxkärna som abstraktionslager mellan mjukvaran och hårdvaran. Linuxkärnan används för drivrutiner, minnes- och processhantering samt närverksfunktionalitet. Androidapplikationer gör dock inte anrop direkt till Linuxkärnan, utan alla applikationsprocesser använder sig av den Dalvik virtual machine instans som denna blivit tilldelad. Dalvik virtual machine kommer att presenteras mer utförligt senare i detta kapitel.

Hyeon-Ju (2012) beskriver de övre lagren i Androids mjukvarustapel i lite mer detalj. Dessa lager är designade för att förenkla utvecklingen av applikationer och innehåller bibliotek och funktionalitet, såsom exempelvis SQLite. Det översta lagret består av ett antal applikationer som följer med Androidsystemet som standard. Androids processkommunikations design gör att dessa applikationer kan användas för att förenkla implementationen av t.ex. e-postfunktionalitet i en utvecklares egna applikation. Detta görs genom att applikationer skicka jobbförfrågningar direkt till andra applikationer med den funktionalitet som behövs.

2.2 Beständigt tillstånd

Termen tillstånd (eng. ”state”) inom datalogi beskrivs av Harris & Harris (2007) som all information som en applikation håller vid en specifik tidpunkt. Tillstånd sparas främst i primärminnet, vilket innebär att det försvinner när primärminnet töms eller inte längre strömtillförs, som t.ex. när enheten stängs av. Ett tillstånd kan också sparas i ett permanent minne så som hårddisk eller flashminne och kan då existera längre än processen som skapat det. Ett sådant tillstånd kallas för beständigt (eng. ”persistent”) (Wikipedia, 2013). Med beständigt tillstånd (eng. ”persistent state”) kan en applikation alltid återställas till samma tillstånd som den befann sig i innan den t.ex. avslutades, kraschade eller tillfälligt avallokerades av operativsystemet.

I Android uppkommer frekvent situationer då tillståndet hos en applikation avallokeras. Ett exempel är när användaren vrider på Androidenheten för att ändra orienteringsläge. Ett annat exempel är när användaren tar emot ett samtal under pågående session. Situationer som dessa resulterar i att applikationens tillstånd avallokeras, vilket kan ge problem som exempelvis skärmfrysningar eller att applikationen kraschar. Vidare avallokerar Android också efter en kort tid applikationer som inte är aktiva, vilket innebär att en process som läggs i bakgrunden ofta snabbt får sin data avallokerad. Genom att implementera en metod för att uppnå beständigt tillstånd kan de problem som avallokering innebär avhjälpas. Viktig applikationsinformation finns då sparad i sekundärminnet och kan återställas när användaren återvänder till applikationen.

2.3 Androids standardmetoder för att uppnå beständigt tillstånd

I detta delkapitel beskrivs de metoder som Android inkluderar som standard för att uppnå beständigt tillstånd. Alla metoder är tillgängliga via Android 4.2 (Jelly Bean), som är den senaste Androidversionen i skrivande stund (2013-04-02). Av dessa metoder använder en Sqldatabas, tre textbaserad serialisering, en binärbaserad serialisering . Serialisering innebär i korthet en process som skriver ett objekts tillstånd till en ström och även använder denna ström för att senare hämta objektets tillstånd igen. Serialisering av denna typ kan användas både till att skicka serialiserad data över nätverk eller till att spara data till sekundärminnet (Sumaray & Kami Makki, 2012).

2.3.1 XML

Extensible Markup Language, förkortat XML, är enligt Bray m.fl. (2006) version av SGML där all data representeras i textformat i form av så kallade entiteter. Tanken bakom XML är bland annat att det ska vara lätt att använda och förstå samt användbart på internet.

Sumaray & Kami Makki (2012) menar att XML är ett av de seraliseringsformat som används mest idag, men att det har fått kritik för att vara för överflödigt i sitt sätt att spara data och därför kan vara olämpligt på mobila plattformar.

2.3.2 JSON

Java Script Object Notation, förkortat JSON, är enligt Nurseitov m.fl. (2009) designat för att vara läsbart för människor, samt att det ska vara lätt för datorer att behandla data. Nurseitov m.fl. (2009) nämner också att JSON kan vara så mycket som 100 gånger snabbare än XML när det används i java script, där JSON har direkt stöd. Sumaray & Kami Makki (2012) beskriver JSON som ett effektivare alternativ till XML och antyder att JSON är det språk som föredras till mobila plattformar.

2.3.3 Preferences

I Android finns stöd för något som kallas delade preferenser (eng. "shared preferences"). Enligt Android Developers (2013e) så är det möjligt att lägga till data som ska vara beständig i en delad preferens. Dessa preferenser är knutna till applikationen och datan som sparas i sekundärminnet kommer automatiskt att tas bort om applikationen avinstalleras. Datan sparas i en speciell intern mapp som är kopplad till applikationen som skapat den, så en utvecklare kan inte själv välja vart datan ska förvaras. Datan serialiseras och deserialiseras genom att använda XML. Delade preferenser har funnits tillgängligt i Android sedan första versionen och går att använda på alla Androidenheter.

2.3.4 Javas Serializable

Enligt Oracle (2013) är Serializable ett gränssnitt i Java som kan användas för att serialisera en klass binärt. Genom att implementera gränssnittet java.io.Serializable så kommer både klassen som implementerat gränssnittet och dess sub-klasser att vara serialiseringbara. Android innehåller bara ett sub-set av java, men serializable gränssnittet är en del av den.

Sumaray & Kami Makki (2012) nämner att även då XML och JSON fortfarande är de mest använda serialiseringsformaten idag så är binär serialisering på framfart. Binär serialisering är oftast både snabbare och tar mindre plats i sekundärminnet. Dock så är en metod för binär serialisering bunden till vissa programmeringspråk och är därför inte lika flexibel som exempelvis XML och JSON.

2.3.5 SQLite

SQL står för Structured Query Language och är ett språk designat för att hämta och modifiera data i relationsdatabaser. Enligt SQLite Home Page (2013) så är SQLite den versionen av SQL som är den mest använda i världen. Hyojun m.fl. (2012) benämner SQLite som en databasmotor som har förhållandevis låg komplexitet och använder förhållandevis liten mängd av primär- och sekundärminne, vilket gör SQLite populär att använda till mobila produkter och inbäddade system. Detta i kombination med att SQLite är öppen källkod är troliga anledningar till att Android valt att inkludera SQLite i sin plattform. I Android kan en eller flera databaser skapas per applikation och de kan sparas var än utvecklaren väljer att förvara databasen.

2.4 Dalvik virtual machine

Bornstein (2008) beskriver Dalvik som en registerbaserad virtuell maskin som är designad för att köras på system med långsam CPU och relativt lite primärminne. Dalvik är en del av

Androidplatformen men bedrivs som ett separat öppen källkods-projekt med Google i spetsen. Enligt Hyeon-Ju (2012) så tar Dalvik de genererade Java klassfilerna och kombinerar dem till en eller flera exekverbara filer, som identifieras som .dex (Dalvik exekverbara filer).

2.4.1 JIT

2010 lanserade Google JIT-stöd för DVM och presenterade denna nya funktionalitet på Google I/O konferensen 2010 via Cheng & Buzbee (2010). JIT står för Just In Time och är ett sätt för Dalvik att aktivt analysera applikationskoden för att översätta den till en så optimerad form som möjligt, vilket gör att applikationens kod kan utföras snabbare. Detta sker medan applikationen är igång, och kan därför leda till att applikationen utför en uppgift snabbare med tiden om applikationen spenderar mycket tid på denna specifika uppgift.

2.5 Flashdisk

Enligt Hyojun m.fl. (2012) så är de flesta mobila enheter idag utrustade med flashdiskar som sekundärminne. Flahdiskar anses vara både snabbare och energisnålare än vanliga mekaniska hårdiskar. I Hyojun m.fl. (2011) och Hyojun m.fl. (2012) nämns att beroende på hur flashdisken och dess kontrollerkort behandlar ingående och utgående information (I/O), så kan applikationers reaktionstider variera kraftigt.

3 Problemformulering

Detta kapitel ger en övergripande beskrivning av problemområdet och en specifikation av det problem som studeras i arbetet. Vidare presenteras arbetets syfte och en metodbeskrivning ges.

3.1 Problemspecifikation och syfte

Som tidigare beskrivits inkluderar Android flera olika metoder för att uppnå beständigt tillstånd genom att spara data till sekundärminnet. En omfattande litteraturgenomgång indikerar att det saknas studier som jämför dessa olika metoder, dvs, det är inte klarlagt vilken metod som är mest effektiv. Det existerar ett antal studier som behandlar jämförelser av tidseffektivitet (se t.ex. Eriksson & Hallberg, 2011; Hericko m.fl, 2003; Maeda, 2012 och Sumaray & Kami Makki, 2012), men dessa studier är inte uteslutande fokuserade på Android och behandlar dessutom endast delmängder av de metoder som Android inkluderar. En omfattande jämförelse av serialiseringsmetoder har också utförts av öppen källkodsprojektet thrift-protobuf-compare project (2013). Dock så inkluderar inte heller denna jämförelse samtliga Androids standardmetoder för att uppnå beständigt tillstånd och testerna är inte heller utförda på en Androidenhet.

Avsaknaden av en kartläggning av effektiviteten hos metoder i Android för att uppnå beständigt tillstånd innebär en betydande risk att den tillståndshantering som implementeras i ett spel inte är den mest effektiva i förhållande till tillståndets storlek, vilket leder till en sämre användarupplevelse.

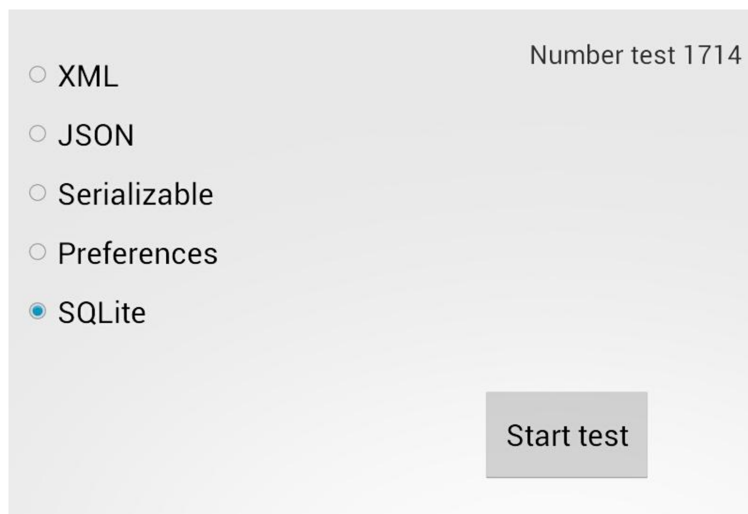
I detta arbete undersöks effektiviteten hos de metoder som inkluderas i Android för att uppnå beständigt tillstånd genom att spara tillståndsdata till sekundärminnet, vilket inkluderar XML serialisering, JSON serialisering, Javas Serializable, Preferences, och SQLite databas. Med effektivitet avses primärt tidseffektivitet, men även storlek av den sparade datan samt hur lätt metoden är att använda kommer att beaktas i undersökningen. Syftet med undersökningen är att identifiera vilka metoder som är mest effektiva och därmed bör väljas i första hand vid implementation av Androidspel. Genom att klarlägga detta finns potential till en förbättrad spelimplementation och därmed också en förbättrad användarupplevelse.

3.2 Metodbeskrivning

En undersökning av metodernas effektivitet kan göras antingen teoretiskt eller praktiskt. Då en teorietisk studie ansågs ge mycket begränsade resultat, främst för att publikationer inom området som skulle kunna utgöra underlag i stor utsträckning saknas, så har en praktisk undersökning valts. Denna praktiska undersökning baseras på en implementation av de olika metoderna. Metoderna implementeras enligt de rekommendationer som finns att tillgå via Android Developers (2013a). Då tillförlitligheten hos arbetets resultat i hög utsträckning beror på korrektheten hos implementation så läggs stor vikt vid att verifiera att koden är korrekt och stämmer med referensimplementationerna.

För att strukturera testerna och underlätta insamlingen av resultat ansågs användandet av ett testverktyg nödvändigt används därför i arbetet. Testverktyget implementerades och används på en fysisk Androidenheter. Testverktyget använder sig av Android standard

development kit (sdk) för att implementera de olika metoderna för att uppnå beständigt tillstånd som undersöks i arbetet. I verktyget finns det möjlighet att välja vilken metod man vill testa genom att markera ett av de givna alternativen. Testet startas därefter genom att trycka på startknappen. Verktygets design är inspirerat av Sumaray & Kami Makki (2012) och en konceptbild återfinns i figur 2 nedan.



Figur 2: Exempelbild på det färdiga testverktyget.

En testiteration består av tre faser, där den första fasen syftar till att mäta den tid (mätt i millisekunder) det tar för de olika metoderna att spara ner testdata till enhetens flashdisk. Vid nedsparningen skrivs all tillståndsdata vid ett och samma tillfälle för att efterlikna en verklig situation där spelprocessen sparar undan all tillståndsdata innan systemet deallokerar datan. Sparas inte all data samtidigt så kommer inte det senast giltiga tillståndet att kunna återskapas efter deallokeringen. Inspirerat av den studie som Sumaray & Kami Makki (2012) genomfört så testas nedsparning av två olika typer av dataobjekt, ett som innehåller text och ett som innehåller nummer. Anledningen till detta är för att hitta eventuella prestationsskillnader som beror på vilken sorts data som behandlas.

I fas två av testiterationen så noteras hur mycket utrymme som det sparade tillståndet tar upp på flashdisken (mätt i byte). Hur mycket utrymme som tas upp är intressant att mäta för att kunna jämföra hur mycket lagringsutrymme de olika metoderna behöver för att representera samma dataobjekt. Utrymme på disk är också något som noterats som viktigt att mäta i tidigare, liknande studier (se exempelvis Eriksson & Hallberg, 2011; Hericko et al, 2003; Maeda 2012 samt Sumaray & Kami Makki, 2012).

I fas tre av testiterationen så mäts den tid det tar att läsa in testdatan och återställa testobjektet. Resultaten från varje testiteration sparas i en textfil för att sedan kunna bearbetas och presenteras i tabeller och grafer som tydliggör resultaten.

Liksom i tidigare, liknande, studier (se t.ex. Hericko et al, 2003 och Maeda 2012) så replikeras både skrivning och läsning ett stort antal gånger (1000 replikeringar per skrivning/läsning) och medelvärdet från replikeringarna används när resultaten analyseras. Anledningen till att använda medelvärde är för att minska eventuella felfaktorer i tidtagningen. Vidare används i tidigare studier (Hericko et al, 2003 och Maeda 2012) en uppvärmningsfas med skrivningar/läsningar innan testiterationerna som tidtags påbörjas, vilket också görs i detta arbete. Anledningen till att använda en uppvärmningsfas är för att

minska felmarginalen som de eventuellt långsammare skrivningarna/läsningarna i början av testningen kan orsaka. Att läsning/skrivning kan gå långsammare i början beror på att DVM:ens JIT-funktionalitet ännu inte fullt hunnit optimera den körbara koden. För att eliminera detta problem så kommer mätningar för arbetet att påbörjas först efter att 1000 uppvärmningstester har körts.

Tester med olika stor mängd testdata kommer att utföras för att få en klar bild över hur bra metodernas tidseffektivitet skalas i förhållande till testdatans storlek. I detta arbete kommer tester att utföras med testobjekt innehållande 100, 250, 500, 1000, 2500, 5000 och 10000 nummer/textvärden i varje lista i testobjektet. Liknande testupplägg har bl.a. utförts av (Hericko et al, 2003 och Maeda 2012) i deras arbeten.

4 Genomförande

Detta kapitel kommer att beskriva implementationen av testverktyget som används för att utföra prestandatesterna samt beskriva de designval som gjort och varför. Efter implementationsdelen kommer resultaten av de genomförda mätningarna att presenteras och förklaras.

4.1 Testmiljö

Testerna kommer att utföras på en Asus Transformer Pad TF300T. Denna platta har valts för att den är en av de vanligaste plattorna på marknaden just nu. För mer information om produktens specifikation se ASUS (2013).

4.2 Testverktygets Implementation

Precis som i Sumaray & Kami Makki (2012) arbete så har testerna utförts i ett egenutvecklat testprogram som kommer att köras på en fysisk Android platta. Genom att undvika emuleringstestning så är förhoppningen att efterlika en så realistisk testmiljö som möjligt för att få mer rättvisande mätningar. Inga extra applikationer kommer vara installerade på plattan, det vill säga att plattan kommer bibehåller det tillstånd som de blev skickade i från fabrik. Testverktyget utvecklas mot Android version 4.0 (Ice Cream Sandwich).

I detta arbete har 5 olika metoder implementerats för att kunna fastställa vilken metod som är tidseffektivas att använda om man vill uppnå beständigt tillstånd i Androidspel, där hela tillståndet sparas samtidigt. Alla implementerade tester har använt samma testdata, som blivit designad för att testa metodernas förmåga att spara både texttunga och nummertunga tillstånd. Tester kommer utföras där listorna i testobjekten innehåller 100, 250, 500, 1000, 2500, 5000 och 10000 nummer eller text variabler. Testverktyget har implementerats i java användandes Eclipse (2013) som utvecklingsmiljö. Två olika tidtagningsmetoder har också använts för att kunna mäta tiden med och utan metoderna hjälptrådar.

4.2.1 Tidtagning

Det först problemet som stöttes på under implementationens gång var att försöka få en så rättvisande tidtagning som möjligt till testerna. Då Android bara innehåller ett eget subset av java så var det inte till en början helt klart vilken tidtagningsmetod som skulle användas för att ge ett så rättvisande resultat som möjligt.

I implementationens början användes först System.currentTimeMillis(). Enligt Android Developers. (2013)g så ger currentTimeMillis() funktionen antalet millisekunder (ms) som passerat sedan epoken. Genom att spara undan detta värde innan och efter ett test så kunde tiden som ett test tog räknas ut. Nedan kan man se hur tidtagningen behandlades i början av implementationen.

```
serializationStartTime = SystemClock.currentTimeMillis();

//Test

serializationEndTime = SystemClock.currentTimeMillis();

serializationResult = serializationEndTime - serializationStartTime;
```

Då testdatan var betydligt mindre i början av implementationen så tog det inte mycket längre än 1-2 ms för många av testerna att slutföras. Därför ansågs det inte tillräckligt noggrant att använda ms i testerna, utan currentTimeMillis() funktionen blev utbyt mot System.nanoTime(). Enligt Android Developers. (2013)g så har nanoTime() liknande funktionalitet som currentTimeMillis() förutom att resultatet ges i nanosekunder (ns) istället för ms. nanoTime() gav den noggrannhet som behövdes för testerna, dock så var denna typ av tidtagning inte tillräklig för att producera ett rättvist testresultat.

Båda funktionerna currentTimeMillis() och nanoTime() hade samma problem, och det är att de mäter den verkliga tiden och inte den tiden som det faktiskt tagit för plattan att utföra det givna testet. Under tidtagningen finns det fortfarande möjlighet för andra processer, och andra trådar i testapplikationens egen process att ta processortid, vilket leder till att beroende på hur CPU:n schemaläggs så kan tiden ett test tar variera väldigt mycket. Detta lede till att tidtagningsmetod bytes ännu en gång, denna gång till Android.os.Process.getElapsedCpuTime(). Enligt Android Developers. (2013)e så mäter getElapsedCpuTime() den totala tiden som en process fått tillgång till CPU:n i ms. Detta medför att tidtagningen nu är mer rättvisande då tiden som andra processer tar på CPU:n under ett test inte räknas med i tidtagningen. Dock så återintroducera denna funktion ett gammalt problem, vilket var att ms inte var tillräkligt noggrann tidsmätning. Detta löstes genom att öka mängden testdata och ansågs acceptabelt för mättningar ner till 100 variabler i testobjektens listor, vilket gav en tid runt 2-3ms.

Vid denna punkt i implementeringen var felkällorna för tidtagningen ganska minimerade men ändå inte helt åtgärdade. Även om det bara är den egna processens tid på CPU:n som räknas så kan även detta vara missvisande, då det kan finnas många trådar i en process. I en process i Android finns det extra trådar som jobbar i bakgrunden utöver de trådar som man själv har implementerat. För att eliminera denna felfaktor så implementerades metoden SystemClock.currentThreadTimeMillis(), som mäter tiden i ms som en specifik tråd fått på CPU:n. Med denna slutgilltiga mätmetod borde de enda variationerna i tidtagningen komma ifrån skillnader i minnesallokeringstid och att tråden eventuellt får vänta på input/output när denna kommunicerar med flashdiskens kontrollerkort. Mer information kan fås om currentThreadTimeMillis() på denna sida Android Developers. (2013)h. Dock så implementerades både currentThreadTimeMillis() och getElapsedCPUTime() i slutändan, då visa metoder använder sig av hjälptrådar så var det även intressant att kunna mäta tiden dessa behövde på CPU:n. Nedan kan man se det slutgiltiga resultatet av tidtagningsimplementationen.

```
serializationStartTime = SystemClock.currentThreadTimeMillis();

//Test

serializationEndTime = SystemClock. currentThreadTimeMillis ();

serializationResult = serializationEndTime - serializationStartTime;
```

4.2.2 Testdata

I början av implementationen så hölls testdatan relativt liten, då detta både skulle underlätta implementationen samt vara mer representativt för ett Androidspel. Testdatan inspirerades även av hur testdatan såg ut i liknande tester så som i Maeda (2012), Hericko m.fl. (2003) och Sumaray & Kami Makki (2012) arbeten, och designades därför för att testa metodernas

förmåga att hantera texttunga och nummertunga tillstånd. Två klasser implementerades, en för att representera texttillståndet och ett för att representera nummertillståndet. Både namngivna variabler och behållare i form av listor har implementerats i klasserna, då de olika metoderna kan hantera namngiven och icke namngiven data olika effektivt. Nedan kan man se hur de två olika klasserna är strukturerade.

```
class WordBattleGameState implements Serializable
{
	public String playerName;
	public String opponentName;

	public ArrayList<String> playerWords;
	public ArrayList<String> opponentWords;

	public WordBattleGameState()
	{
		playerName = "default";
		opponentName = "default";

		playerWords = new ArrayList<String>();
		opponentWords = new ArrayList<String>();
	}
}

class PhysicsPuzzleGameState implements Serializable
{
	public Integer playerScore;
	public Integer currentLevel;

	public ArrayList<Integer> entitiesSpeed;
	public ArrayList<Integer> entitiesDirection;

	public PhysicsPuzzleGameState()
	{
		playerScore = 0;
		currentLevel = 0;

		entitiesSpeed = new ArrayList<Integer>();
		entitiesDirection = new ArrayList<Integer>();
	}
}
```

När testapplikationen startas så instansieras och fylls dessa två klasser med relevant testdata. Som nämnt ovan då implementationena av tidtagningen förklarades så var det tvunget att öka storleken på testdatan för att få ett mer rättvisande resultat användandes millisekunder som tidsenhet. Då en del av metoderna som skulle testas krävde lågnivåhantering av namngivna variabler så ansågs det inte realistiskt att öka antalen av dessa allt för mycket, istället lass mer värden in i listorna. En naktdel är att testdatan nu inte längre representerar ett lika realistiskt tillstånd för ett Androidspel. Storleken kan nu uppkomma till 0.1 MB och börjar därför närma sig mer storleken som man förväntar sig att tillstånd i PC-spel att har. Dock var detta en nödvändig ändring för att få ett mer rättvisande resultat av de olika metodernas prestationer, vilket är huvudfokus i detta arbete. I slutändan utfördes tester testobjekt innehållande 100, 250, 500, 1000, 2500, 5000 och 10000 variabler för att få en klar överblick av hur metoderna effektivitet påverkades beroende på

datamängden. 100 variabler ansågs som det lägsta acceptabla mängden för tidsmätningar i ms.

4.2.3 Anpassa koden för DVM:en

Efter att ha läst arbeten som Maeda (2012) och Sumaray & Kami Makki (2012) så klarstod det att man var tvungen att designa sitt test så att den virtuella maskinens JIT funktionalitet inte påverkar testmätningarna. I Maeda (2012) och Sumaray & Kami Makki (2012) arbeten har dem löst problemet genom att köra ett högt antal uppvärmningsiterationer av testerna innan iterationer med faktisk tidtagning körs. På grund av detta valdes en liknande design till detta arbetes implementation, där 1000 uppvärmningsiterationer kördes innan 1000 tidtagningsiterationer utfördes. Den inledande implementationen kan ses nedan.

```
for(int i = 0; i < 1000; i++)
{
	//Uppvärmingstester
}

for(int i = 0; i < 1000; i++)
{
	//Tidtagningstester
}
```

Den ovan visade implementationen fick senare ändras då DVM:en behandla de båda for-looparna som separata kodstycken, och också optimerar dem separat under körning. För att få DVM:en att optimera testkoden innan faktisk tidtagning påbörjas måste uppvärmningsiterationerna och tidtagningsiterationerna utföras av samma for-loop. Nedan kan man se den slutgiltiga implementationen.

```
for(int i = 0; i < numberOfRuns; i++)
{
	if(i >= numberOfWarmups)
		serializationStartTime    =    SystemClock.currentThreadTimeMillis();


	//Test


	if(i >= numberOfWarmups)
	{
		serializationEndTime = SystemClock.currentThreadTimeMillis();

		serializationResult = serializationEndTime - serializationStartTime;
	}
}
```

Genom att sätta numberOfWarmups till 1000 så hinner koden köras så pass länge innan tidtagningsiterationerna påbörjas så att JIT funktionaliteten hinner optimera den körbara koden fullt ut, och påverkar därför inte tidtagningen.

4.2.4 Generella designval och implementation

Det absolut största problemet i detta arbete har varit att få en så rättviss implementering av varje metod som möjligt. Då några av metoderna kräver implementation på ganska låg nivå så finns det en väldigt stor chans för missvisande resultat på grund av att metoder som blivit

ineffektivt implementerade. För att minimeras denna felfaktor så har exempelimplementationer från Android Developers. (2013)a försökts följas till så hög grad som möjligt. Dock så har i vissa fall ändringar behövt göras på grund av att exempelimplementationerna varit för inriktade på minneseffektivitet, då vi i detta arbete vill fokusera på tidseffektivitet.

En generell tumregel som använts genom alla implementationerna har varit att allokeringar av resurser som behövs vid sparandet och läsandet av tillstånden räknas med i tidtagning. Ett exempel på en sådan resurs kan t.ex. vara StringBuilder till XML eller JSON metoderna eller ByteArrayOutputStream för Serializable. Då återanvändandet av visa resurser i visa fall varit väldigt omständigt eller rent ut sagt omöjligt så har denna tummregel införts och har gällt vid implementeringen av alla metoder.

4.2.5 Metodspecifika designval och implementation

De metoder som implementerats är XML, JSON, javas Serializable, Preferences och SQLite databas. XML, JSON och Preferences är text baserad serialisering och javas Serializable är binär baserad serialisering. Android innehåller även en metod kallad Parcelable för binär baserad serialisering, men då denna metod är designad för processkommunikation som huvudfokus så kommer denna metod inte att vara med i analysen. Anledningen till detta är att Parcelable inte är designat för att spara ner data permanent. Google kan när som helst göra ändringar i hur Parcelable hanterar och strukturerar data vid en eventuell uppdatering av Android. Enligt Android Developers (2013)c kan detta leda till att data som blivit sparad på disk innan en uppdatering inte längre går att använda efter att en uppdatering utförts.

Den första implementationen som utfördes var XML testet. I början av denna implementering så upptäcktes ganska snabbt att det fanns mer än ett sätt att använda XML serialisering i Android. XmlPullParser valdes på grund av rekommendationer ifrån Android Developers (2013)d där metoden beskrevs som enkel att underhålla och effektivt. Den implementerade XML serialiseringen skriver först hela sin layout till primärminnet innan den sedan sparar datan till flashdisken. Detta görs för att det går märkbart snabbare att göra en stor skrivning till flashdisken än många små skrivningar. Med den funktionalitet som finns för XML serialisering i Android så kommer man inte ifrån att det blir mycket låg nivå hantering av tillståndets data.

I implementationen av JSON testerna så används Androids JsonWritter klassen, vilket verkar vara det enda sättet att serialisera till JSON formatet på Android. Precis som i XML implementationen så sparas hela stringlayouten i primmärminnet innan det sparas ner till flashdisken, vilket är mer tidseffektivt än flera små skrivningar. Implementationen har gjorts i förhållande till Android Developers. (2013)b rekommendationer, och har därför inte använt JsonTokener. Att använda JSONTokener skulle underlätta implementationen då man inte behöver behandla informationen på samma låga nivå. Dock så är användandet av JSONTokener långsamare än den nuvarande implementationen och därför används den befintliga. I början av implementationen så lästes hela strängen in för att konverteras direkt till ett JSONObject som visas nedan.

```
String jsonString = reader.readLine();
stream = new FileInputStream(file);
FileChannel fc = stream.getChannel();

MappedByteBuffer bb = fc.map(FileChannel.MapMode.READ_ONLY, 0, fc.size());
```

```
jString = Charset.defaultCharset().decode(bb).toString();
stream.close();

json = new JSONObject(jString);
```

Dock visade sig detta vara ett ineffektivt sätt att återställa tillståndet på, därför ändrades det senare till att använda sig av StringBuilder klassen, som mycket effektivare kunde återställa tillståndet. Nedan kan man se den slutgilltiga koden.

```
BufferedReader reader = new BufferedReader(new InputStreamReader(new
FileInputStream(file), "UTF-8"));

StringBuilder builder = new StringBuilder();

for(String line = null; (line = reader.readLine()) != null;)
{
	builder.append(line).append("\n");
}

String jsonString = builder.toString();

jsonObject = new JSONObject(jsonString);
```

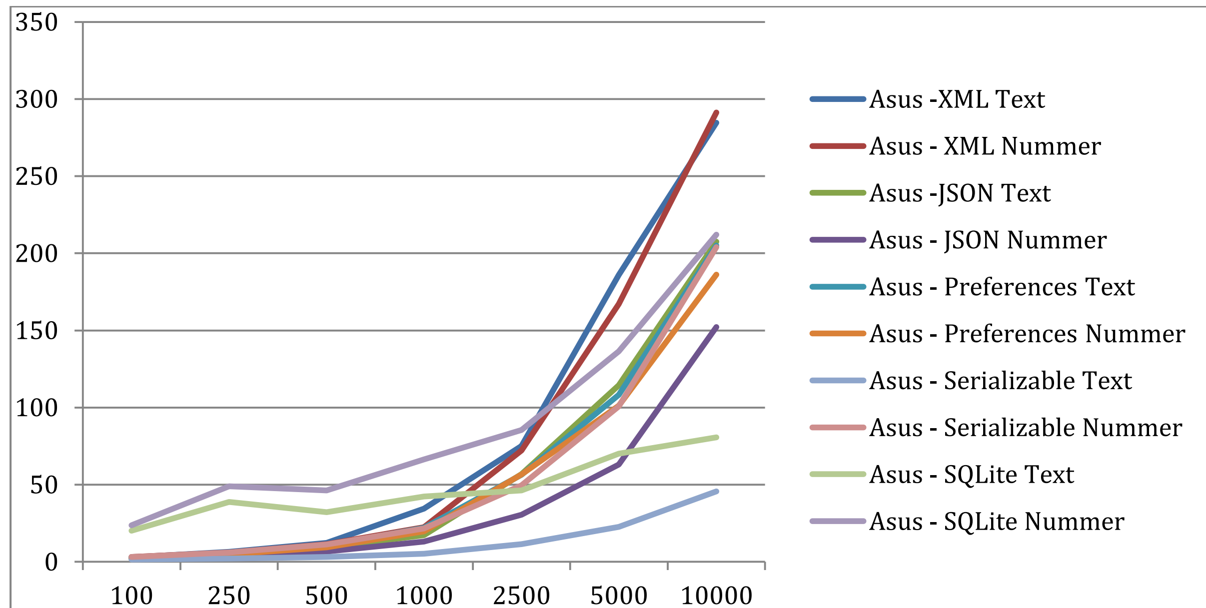
Vid implementationen av SQLite testet så behandlas listorna i testklasserna som binära objekt när de lagras i databasen, vilket var det mest tidseffektiva sättet att behandla så stora listor. Två separata hjälpklasser, en för varje testobjekt, har implementerats för att behandla sparning och läsning till de två olika databaserna som representerar de två olika tillstånden. Nedan kan man se en del av den slutgilltiga koden som använts i hjälpklassen för att behandla texttillståndet.

```
public class TextDBAdapter
{
	public static final String KEY_ROWID = "_id";
	public static final String KEY_PLAYER_NAME = "player_name";
	public static final String KEY_OPPONENT_NAME = "opponent_name";
	public static final String KEY_PLAYER_WORDS = "player_words";
	public static final String KEY_OPPONENT_WORDS = "opponent_words";
	public static final String TAG = "TextDBAdapter";

	private static final String DATABASE_NAME = "sql_text_test";
	private static final String DATABASE_TABLE = "text_state";
	private static final int DATABASE_VERSION = 1;

	private static final String DATABASE_CREATE_TABLE =
			"create table text_state (_id integer primary key
			autoincrement, "
			+ "player_name text not null, "
			+ "opponent_name text not null, "
			+ "player_words BLOB not null, "
			+ "opponent_words BLOB not null);";

	private final Context context;

	private DatabaseHelper DBHelper;
	private SQLiteDatabase db;

	public TextDBAdapter(Context context)
```

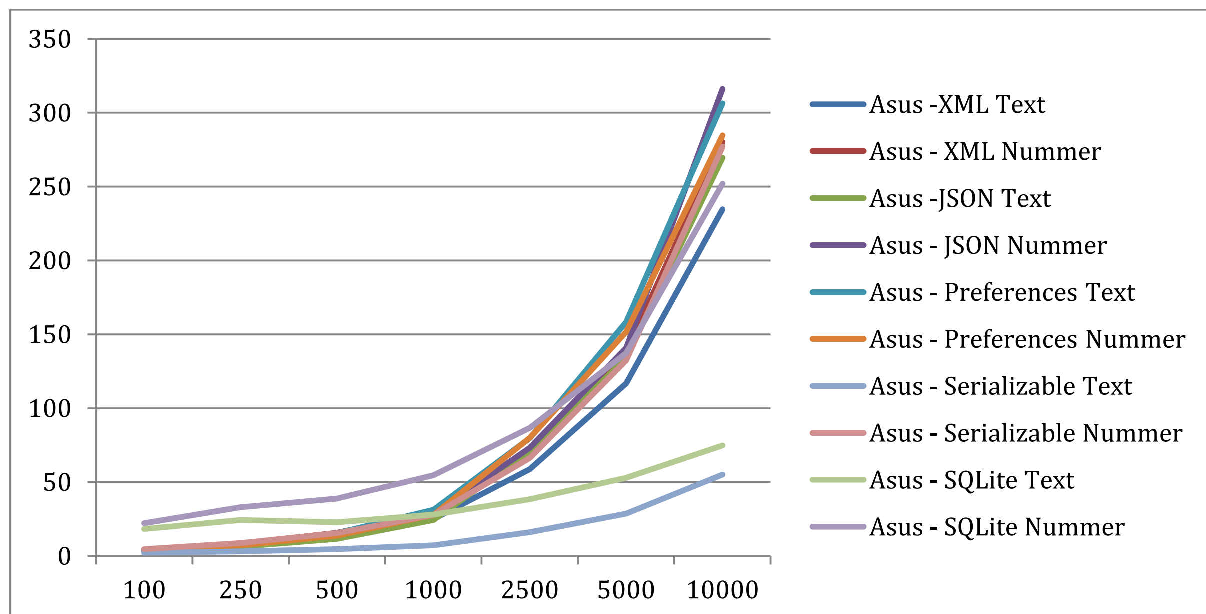
```
	{
		this.context = context;
		DBHelper = new DatabaseHelper(context);
	}

	private static class DatabaseHelper extends SQLiteOpenHelper
	{
		DatabaseHelper(Context context)
		{
			super(context, DATABASE_NAME, null, DATABASE_VERSION);
		}

		@Override
		public void onCreate(SQLiteDatabase db)
		{
			db.execSQL(DATABASE_CREATE_TABLE);
		}

		@Override
		public void onUpgrade(SQLiteDatabase db, int oldVersion, int
		newVersion)
		{
			Log.w(TAG, "Upgrading database from version " + oldVersion + "
			to " + newVersion + ", which will destroy all old data");
			db.execSQL("DROP TABLE IF EXISTS titles");
			onCreate(db);
		}
	}

	//--- Insert state ---
	public long insertState(String player_name, String opponent_name, byte[]
	player_words, byte[] opponent_words)
	{
		ContentValues initialValues = new ContentValues();
		initialValues.put(KEY_PLAYER_NAME, player_name);
		initialValues.put(KEY_OPPONENT_NAME, opponent_name);
		initialValues.put(KEY_PLAYER_WORDS, player_words);
		initialValues.put(KEY_OPPONENT_WORDS, opponent_words);
		return db.insert(DATABASE_TABLE, null, initialValues);
	}

	…


	//Resten av klassen


	…

}
```

4.3 Analys

I figur 3 till 9 nedan presenteras testresultatet från undersökningens tester.



Figur 3: Resultat från nedsparning med trådtidtagning. X-axeln visar antalet dataobjekt i testobjektet, y-axeln visar tiden i millisekunder.

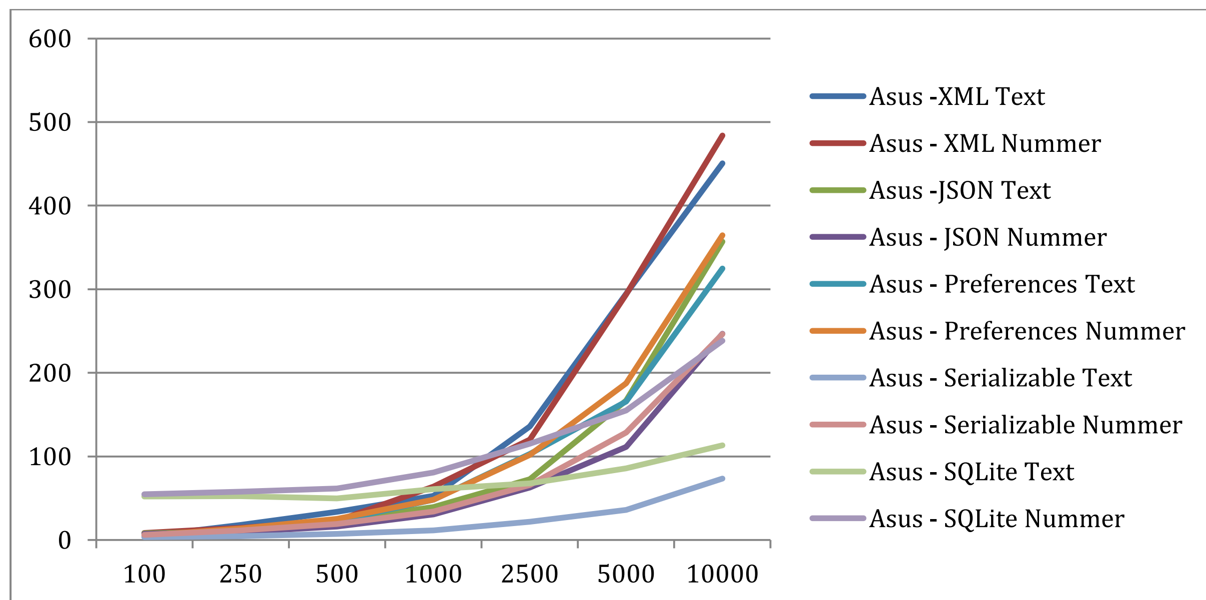


Figur 4: Resultat från inläsning med trådtidtagning. X-axeln visar antalet dataobjekt i testobjektet, y-axeln visar tiden i millisekunder.

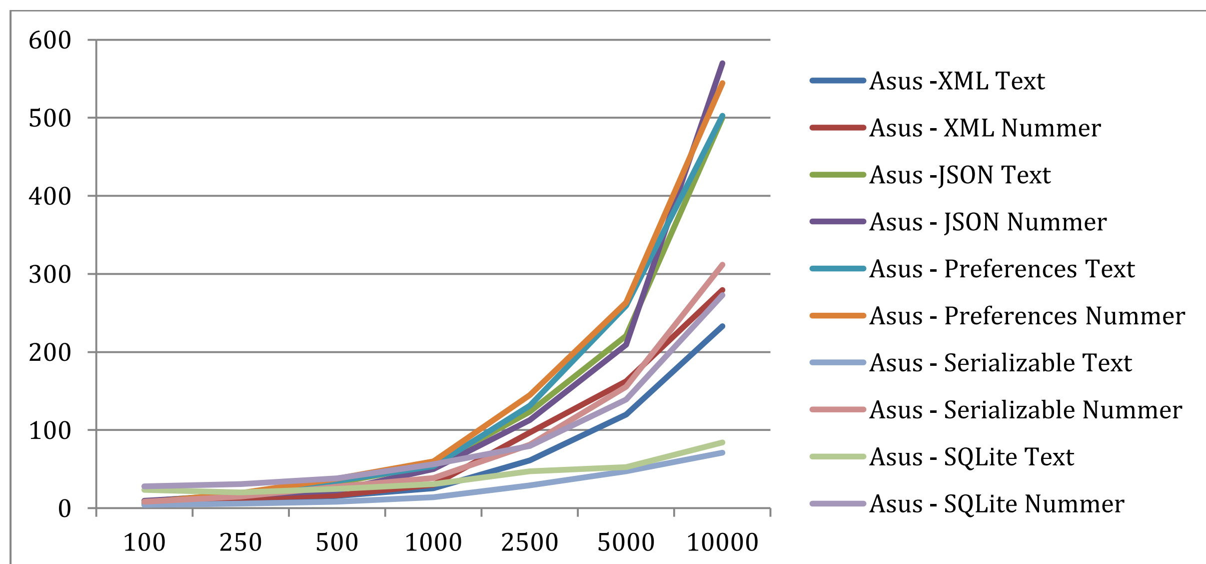
Som framgår av figur 3 och 4 så indikerar resultaten att de flesta metoder presterar relativt lika vid hantering av små mängder data. En metod som presterar betydligt sämre än de andra metoderna med små mängder data är SQLite, dock är denna metod samtidigt en av de bättre vid hantering av stora datamängder. Anledningen till att SQLite är sämre vid små datamängder beror på att databasen måste skapas i varje test, vilket är tidskrävande. Vidare indikerar testresultaten att den metod som presterar bäst vid både små och stora datamängder vid nedsparning är Javas Serializable. Java Serializable är också den metod

som presterar bäst vid inläsning av text, medan JSON är den metod som är mest tidseffektiv vid inläsning av nummer.

I figur 5 och figur 6 nedan presenteras testresultatet för nedsparning och inläsning mätt i hur mycket tid processen har haft på CPU:n. Som nämnts tidigare så används två olika typer av tidtagningar i testerna för att kunna se ifall någon av metoderna använder sig av extra bakgrundstrådar.



Figur 5: Resultaten från nedsparning med processtidtagning. X-axeln visar antalet dataobjekt i testobjektet, y-axeln visar tiden i millisekunder.

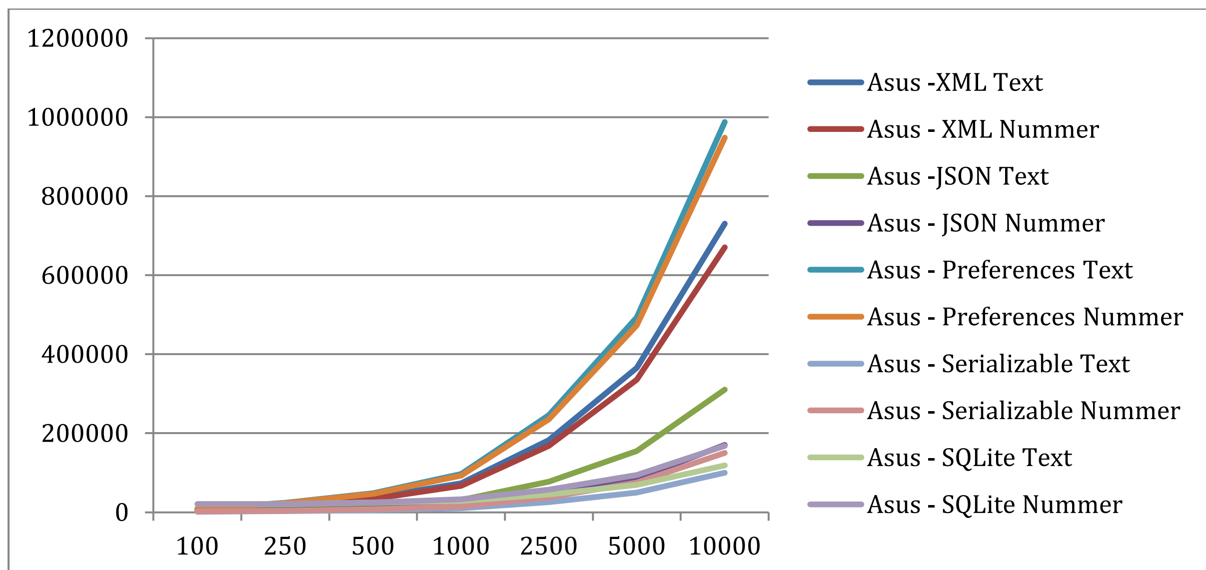


Figur 6: Resultat från inläsning med processtidtagning. X-axeln visar antalet dataobjekt i testobjektet, y-axeln visar tiden i millisekunder.

I figur 5 och figur 6 framgår att samtliga metoder är långsammare med processtidtagningen i jämförelse med trådtidtagningen. I relation till varandra liknar dock resultaten för de olika metoderna i processtidtagning de från trådtidtagningen, med undantag för JSON och Preferences som erhåller ett sämre resultat med processtidtagning. Detta beror på att dessa

metoder använder egna bakgrundstrådar och i processtidtagningen, till skillnad från trådtidtagningen, så tillkommer tiden som dessa trådar spenderar på CPU:n.

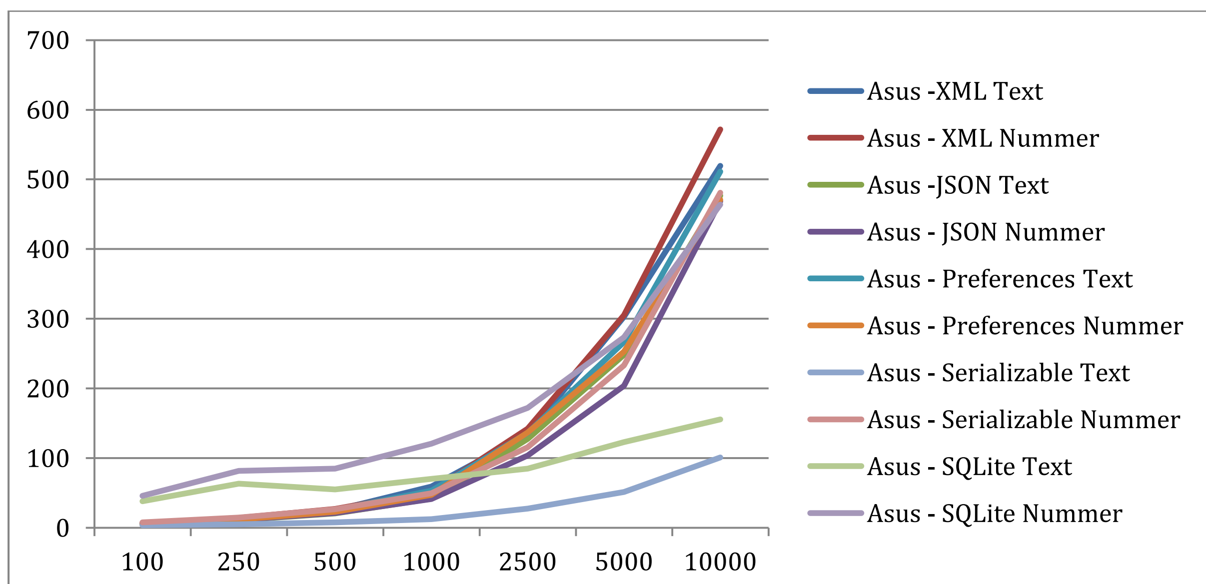
I figur 7 nedan visas storleken på de nedsparade filerna för de olika metoderna.



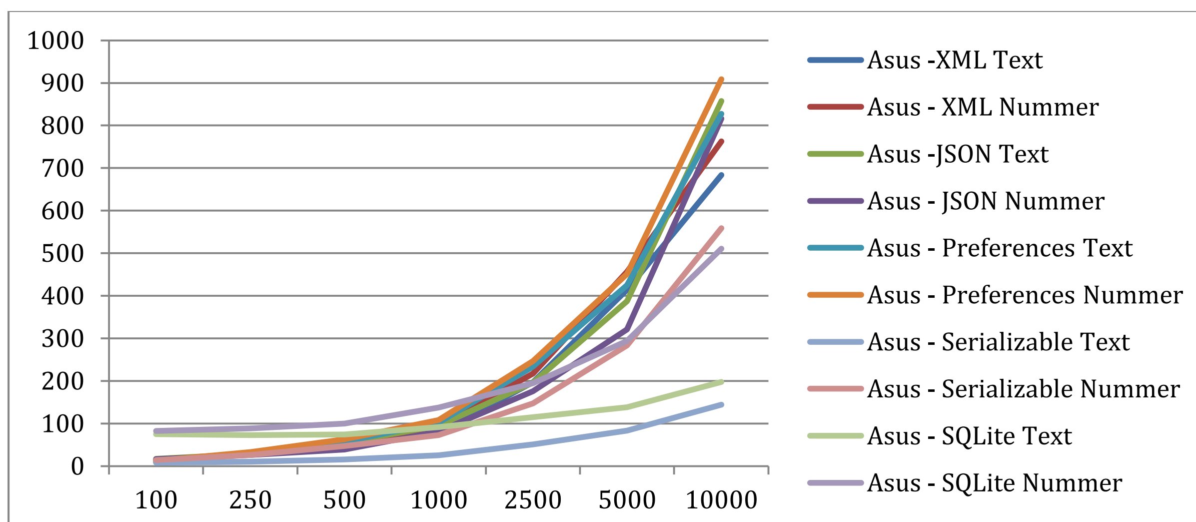
Figur 7: Storleken på de sparade filerna. X-axeln visar antalet dataobjekt i testobjektet, y-axeln visar storleken i bytes.

Resultaten av storleksmättningarna visar att SQLite och Javas Serializable är de metoder som är mest effektivt reducerar storleken på den sparade datan i förhållande till testdatans storlek.

I figur 8 och figur 9 nedan presenteras den sammanlagda behandlingstiden (dvs. tiden det tar att spara hela tillståndet och sedan läsa in det igen) för de olika metoderna med trådtidtagning respektive processtidtagning.

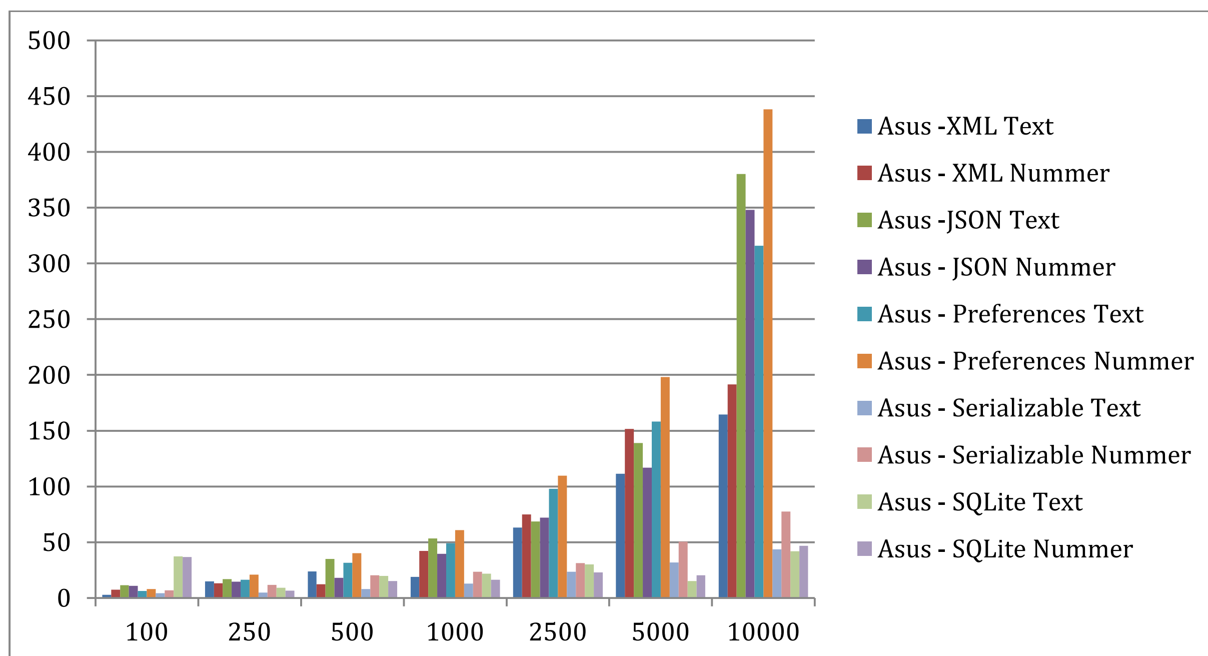


Figur 8: Sammanlagd behandlingstid med trådtidtagning. X-axeln visar antalet dataobjekt i testobjektet, y-axeln visar tiden i millisekunder.



Figur 9: Sammanlagd behandlingstid med processtidtagning. X-axeln visar antalet dataobjekt i testobjektet, y-axeln visar tiden i millisekunder.

Figur 9 visar, i likhet med tidigare resultat, att de flesta metoder presterar relativt likvärdigt vid små mängder data. SQLite är långsammare vid små mängder data, vilket som nämnts tidigare beror på att SQLite måste skapa databasen i varje test. Vid mellanstora och stora datamängder så presterar SQLite och Javas Serializable bäst, för både nummer och text.



Figur 10: Skillnaden i sammanlagd behandligstid mellan trådtidtagning och processtidtagning för de olika metoderna. X-axeln visar antalet dataobjekt i testobjektet, y-axeln visar tiden i millisekunder.

I Figur 10 ovan framgår tydligt att JSON och Preferences är de metoder som har störst skillnad mellan de olika tidtagningarna. Detta antyder att dessa metoders hjälpklasser använder sig av hjälptrådar. Arbetet som dessa trådar utför på CPU:n kommer inte med i trådtidtagningen, dock så kommer de med i processtidtagningen. Detta leder till att dessa metoder har en ganska stor differens mellan trådtidtagningen och processtidtagningen.

5 Slutsats

Detta kapitel börjar med att presentera de viktigaste resultaten från arbetet i relation till arbetets mål. Därefter diskuteras intressanta reflektioner från arbetet. Kapitlet avslutas med en presentation av möjliga framtida utökningar av arbetet.

5.1 Sammanfattning

I Android uppkommer frekvent situationer då tillståndsdatan i ett spels process avallokeras vilket kan ge problem som exempelvis skärmfrysningar. Ett sätt att hantera detta problem är att använda beständigt tillstånd, vilket innebär att ett spels tillstånd sparas till sekundärminnet. Android inkluderar flera olika metoder att uppnå beständigt tillstånd, men en omfattande litteraturgenomgång indikerar att det saknas studier som jämför effektiviteten hos dessa olika metoder. Avsaknaden av en kartläggning av effektiviteten hos metoder i Android för att uppnå beständigt tillstånd innebär en betydande risk att den tillståndshantering som implementeras i ett spel inte är den mest effektiva i förhållande till tillståndets storlek, vilket leder till en sämre användarupplevelse. Detta arbete har haft som syfte att identifiera vilka metoder som är mest effektiva och därmed bör väljas i första hand vid implementation av Androidspel.

I arbetet har inkluderats de standardmetoder som finns i Android för att uppnå beständigt tillstånd, vilket inkluderar XML serialisering, JSON serialisering, Javas Serializable, Preferences, och SQLite databas. För att utvärdera de olika metoderna har ett testverktyg implementerats och de olika metodernas effektivitet har mätts utifrån tidseffektivitet och storlek på den sparade datan. Mätningarna från testverktyget har sammanställts i grafer för att tydligt kunna jämföra de olika metodernas effektivitet. Sammanfattningsvis visar resultaten från arbetet att alla metoder är ungefär lika effektiva vid små datamängder, utom SQLite som presterar sämre än de andra. Vid större datamängder är dock SQLite, tillsammans med Java Serializable, mest effektiv. SQLLite och Java Serializable är också de två metoder som använder minst utrymme i sekundärminnet för att spara tillståndet.

5.2 Diskussion

Resultaten från arbetet visar att ingen av Androids metoder för att spara tillståndsdata till sekundärminnet är entydigt bäst, utan vilken metod som är mest effektiv beror på hur stor datamängd som tillståndet utgör i det specifika fallet. Effektiviteten hos de olika metoderna är också beroende på om det är text eller nummer som sparas ner. Resultaten från arbetet visar att effektiviteten hos en viss metod kan skilja sig väsentligt beroende på om det är text eller nummer som hanteras. Generellt är nedsparning av nummer långsammare jämfört med motsvarande datamängd i text för de metoder som använder sig av textserialisering, vilket beror på att dessa metoder behöver konvertera nummer till text innan nedsparning. Mot bakgrund av detta rekommenderas att vid implementation av Androidspel analysera både tillståndets ungefärliga storlek och dess innehåll (mestadels text eller mestadels nummer) innan metod för att spara tillståndsdata väljs. I tabell 1 nedan ges en guide till val av metod ur ett effektivitetsperspektiv baserat på datamängd och innehåll.

	Liten datamängd 100 - 250	Mellanstor datamängd 250 - 2500	Stor datamängd 2500 - 10000
Mestadels text	Alla utom SQLite	Serializable	Serializable
Mestadels nummer	Alla utom SQLite	JSON eller Serializable	SQLite, Serializable eller JSON

Tabell 1: Guide till val av metod för nedsparning av tillståndsdata i Androidspel för androidenheter med flerkärning processor. Enda skillnaden vid val till enkärniga enheter är att JSON inte skulle finnas med i tabellen.

Det bör nämnas att testresultaten för SQLite har påverkats av det generella desingbeslut som togs vid implementationen av testverktyget med avseende på att allokeringen av alla resurser som metoderna använde sig av skulle vara en del av tidtagningen. Detta måste anses vara till nackdel för SQLite som måste skapa om databasen i början av varje test. Om databasen istället varit förallokerad så skulle resultaten för SQLite förmodligen vara i nivå med de andra metodernas vid små datamängder.

Utöver effektivitet är en väsentlig aspekt att väga in vid valet av metod för att spara tillståndsdata hur enkel metoden är att implementera. Denna aspekt har inte utvärderas explicit som en del av arbetet, men författarens subjektiva åsikt är att Java Serializable och Preferences är de två metoder som är enklast att förstå och använda. Den metod som av författaren anses svårast att implementera var SQLite. Anledningen till att denna metod anses svårast är att den kräver en del lågnivå implementation, samt att man behöver kunna skriva sql frågor (queries) för att manipulera data.

De resultat som presenterats i arbetet är användbara inte bara för Androidspel, utan kan tillämpas generellt för Androidapplikationer. Alla Androidapplikationer har nytta av en effektiv tillståndshantering eftersom detta leder till en bättre användarupplevelse, men framförallt värdefulla är arbetets resultat för de applikationer i vilka användaren, liksom i spel, förväntar sig korta reaktionstider. Ett exempel på en sådan applikation är navigeringsmjukvara där en skärmfrysning kan innebära en felkörning eller i värsta fall en trafikfara.

Vidare är det rimligt att anta att arbetets resultat också är användbara för implementation av kommunikationsrelaterad serialisering. Med stor sannolikhet så skulle de metoder som presterar bäst i denna undersökning även vara de metoder som är effektivast för användning vid process- och webbkommunikation i Androidapplikationer. Detta gäller då specifikt serialiseringsmetoderna som även kan användas till den typen av funktionalitet, så som XML, JSON och java Serializable.

Det är också möjligt att vissa av de metoder som utvärderats i detta arbete skulle uppvisa liknande resultat även på andra plattformar än Android. Metoderna Serializable och SQLite är metoder vars standard är den samma på många olika plattformar och det är därför inte orimligt att anta att dessa borde prestera ungefär lika bra i förhållande till varandra oavsett plattform. Implementationerna av XML, JSON och parceble har i denna undersökning dock varit så hårt knutna till Android, så resultaten för dessa metoder är inte direkt relevanta för andra plattformar.

Ur ett större perspektiv så är den generella nyttan med arbetets resultat att de kan leda till att både applikationer och spel uppvisar en bättre användarupplevelse. Att implementera så effektiva lösningar som möjligt är också något som alltid är eftersträvansvärt inom mjukvaruutveckling. Detta gäller inte minst mobila enheter eftersom ju effektivare mjukvaran exekveras desto mindre ström drar enheten och desto längre varar då batteriet.

Till sist bör nämnas att den undersöknings som genomförts i arbetet kan vara svår att återskapa. Det största hindret är att små ändringar i implementationen kan leda till stora skillnader i testresultat. För att få exakt samma resultat som i arbetet så krävs därför en ganska exakt kopia av den koden som användes vid undersökningen. Ett annat problem kan vara bakåtkompatibilitet. När det gäller Androids API är det i nuläget (juni 2013) inga problem då ingen funktionalitet för de gamla versionerna tas bort eller ändras när en ny version släpp. Sannolikheten är därför stor att det kommer att gå att implementera ett liknande test till samma Androidversion som denna undersökning använt sig av. Dock så kan det finnas svårigheter att i framtiden hitta Androidenheter som faktiskt använder just denna version (som ju då är gammal), vilket leder till att det blir svårt att återskapa exakt samma test eftersom ändringar i metodernas funktionalitet kan ha skett i nyare Androidversioner.

5.3 Framtida arbeten

De metoder som undersökts i detta arbete är de som följer med Android som standard. Utöver dessa metoder så finns ytterligare möjligheter att uppnå beständigt tillstånd genom att använda externa bibliotek. Hur effektiva Androids standardmetoder är i jämförelse med de metoder som tillhandahålls i externa bibliotek är inte klarlagt, och det skulle därför vara intressant att i framtiden utöka studien till att inkludera också externa bibliotek.

Beständigt tillstånd kan även uppnås genom att lagra data globalt snarare än lokalt, t.ex. via en molntjänst eller andra nätverksmetoder. En viktig fördel med metoder som lagrar data globalt är att de inte är låsta till en specifik Androidenhet, vilket innebär att användaren kan fortsätta sin spelsession på en annan enhet. I en framtida undersökning skulle det vara intressant att jämföra effektiviteten hos globala metoder med de metoder som sparar ner tillståndet lokalt.

Ytterligare en intressant studie för framtiden är en prestandajämförelse på andra Androidenheter än den som används i denna undersökning för att se hur mycket hårdvaran påverkar testresultaten. Exempelvis är det möjligt at metoder är väldigt olika effektiva beroende på systemets hårdvara. Det är också möjligt att olika hantering av in- och utdata för olika flashdiskars kontrollerkort har påverkan på metodernas prestanda.

Även om man kan göra antaganden baserat på resultaten från denna undersökning för att få en bra bild över hur dessa metoder presterar vid användning vid processkommunikation och webbkommunikation i andra Androidapplikationer så skulle ytterligare tester behöva utföras för att säkerställa att dessa antaganden verkligen stämmer. Testverktyg för dessa tester skulle kunna implementeras som en applikation innehållande ett antal processer för processkommunikations relaterade tester, eller ett webbgränssnitt (webbsida) för tester relaterade till webbkommunikation.

Det skulle även vara intressant att göra liknande undersökningar på andra plattformar för att få en lika användbar kartläggning mellan metoderna på plattformen i fråga. Man kan i

visa fall använda resultaten från denna undersökning för att få en uppfattning hur bra en metod är i förhållande till en annan även på andra plattformar. Dock så krävs det en undersökning för att få detaljerade resultat mellan alla metoder.

Referenser

Android Developers. (2013)a *Android, the world's most popular mobile platform | Android Developers*. [Online] Available at: <http://developer.android.com/about/index.html> [Accessed 21 March 2013].

Android Developers. (2013)b *JsonWriter | Android Developers*. [Online] Available at: <http://developer.android.com/reference/android/util/JsonWriter.html> [Accessed 28 March 2013].

Android Developers. (2013)c *Parcel | Android Developers*. [Online] Available at: <http://developer.android.com/reference/android/os/Parcel.html> [Accessed 03 April 2013].

Android Developers. (2013)d *Parsing XML Data | Android Developers*. [Online] Available at: <http://developer.android.com/training/basics/network-ops/xml.html> [Accessed 28 March 2013].

Android Developers. (2013)e *Process | Android Developers*. [Online] Available at: <http://developer.android.com/reference/android/os/Process.html#getElapsedCpuTime()> [Accessed 28 March 2013].

Android Developers. (2013)f *Storage Options | Android Developers*. [Online] Available at: <http://developer.android.com/guide/topics/data/data-storage.html> [Accessed 21 March 2013].

Android Developers. (2013)g *System | Android Developers*. [Online] Available at: <http://developer.android.com/reference/java/lang/System.html> [Accessed 28 March 2013].

Android Developers. (2013)h *SystemClock | Android Developers*. [Online] Available at: <http://developer.android.com/reference/android/os/SystemClock.html#currentThreadTimeMillis()> [Accessed 28 March 2013].

ASUS. (2013) ASUS - - ASUS ASUS Transformer Pad TF300T. [Online] Available at: <http://www.asus.com/Tablet_Mobile/ASUS_Transformer_Pad_TF300T/#specifications> [Accessed 20 February 2013].

Bornstein, D. (2008) *Dalvik VM Internals*. Google I/O. [session] 2008. Available at: <https://sites.google.com/site/io/dalvik-vm-internals> [Accessed 21 March 2013].

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F & Cowan, J. (2006) *Extensible Markup Language (XML) 1.1 (Second Edition)*. [Online] W3C Available at: <http://www.w3pdf.com/W3cSpec/XML/2/REC-xml11-20060816.pdf> [Accessed 21 March 2013].

Cheng, B. & Buzbee, B. (2010) *A JIT Compiler for Android's Dalvik VM*. Google I/O. [session] May 19-20. Available at: <http://www.google.com/events/io/2010/sessions/jit-compiler-androids-dalvik-vm.html> [Accessed 21 March 2013].

Eclipse. (2013) *Eclipse - The Eclipse Foundation open source community website*. [Online] Available at: <http://www.eclipse.org/> [Accessed 31 March 2013].

Eriksson, M & Hallberg, V. (2011) Comparison between JSON and YAML for data serialization. [Online] Royal Institute of Technology Available at: <http://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand11/Group2Mads/Rapport_Malin_Eriksson_Viktor_Hallberg.pdf> [Accessed 11 Maj 2013].

Harris, D. M. & Harris, S. L. (2007) *Digital Design and Computer Architecture*. USA: Morgan Kaufmann. s. 103. ISBN 0123704979.

Hericko, M., Juric, M.B., Rozman, I., Beloglavec, S. & Zivkovic, A. (2003) *Object serialization analysis and comparison in Java and .NET*. ACM SIGPLAN Notices, [Newsletter] 38(8). s. 44–54.

Hyojun, K., Agrawal, N. & Ungureanu, C. (2011) *Examining storage performance on mobile devices*. MobiHeld '11 Proceeding of the 3rd ACM SOSP Workshop on Networking, Systems, and Applications on Mobile Handhelds, [Proceeding] Article No. 6.

Hyojun, K., Agrawal, N, & Ungureanu, C. (2012) *Revisiting Storage for Smartphones*. ACM Transactions on Storage (TOS), [Journal] 8(4). Article No. 14.

Maeda, K. (2012) *Performance evaluation of object serialization libraries in XML, JSON and binary formats*. Digital Information and Communication Technology and it's Applications (DICTAP), 2012 second International Conference [Conference Publication] s. 177–182.

Nurseitov, N., Paulson, M., Reynolds, T. & Izurieta, C. (2009) *Comparison of JSON and XML Data Interchange Formats: A Case Study*. Proceedings of CAINE (2009) s. 157–162.

Oracle. (2013) Serializable (Java Platform SE 6). [Online] Oracle Available at: <http://docs.oracle.com/javase/6/docs/api/java/io/Serializable.html> [Accessed 21 March 2013].

SQLite. (2013) *SQLite Home Page*. [Online] Available at: <http://www.sqlite.org/> [Accessed 18 February 2013].

Sumaray, A. & Kami Makki, S. (2012) *A comparison of data serialization formats for optimal efficiency on a mobile platform*. Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication. Article No. 48.

thrift-protobuf-compare project. (2013) Home – eishay/jvm-serializers Wiki – GitHub. [Online] Available at: <https://github.com/eishay/jvm-serializers/wiki> [Accessed 11 Maj 2013].

Wikipedia (2013) *Persistence (computer science) - Wikipedia, the free encyclopedia*. [Online] Available at: <http://en.wikipedia.org/wiki/Persistence_(computer_science)> [Accessed 06 Juni 2013].