



HoGent

Faculteit Bedrijf en Organisatie

Vergelijkende studie Android: Dalvik vs. Android Runtime (ART)

Tristan Mariën

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Jens Buysse

Instelling: —

Academiejaar: 2015-2016

Derde examenperiode

Faculteit Bedrijf en Organisatie

Vergelijkende studie Android: Dalvik vs. Android Runtime (ART)

Tristan Mariën

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Jens Buysse

Instelling: —

Academiejaar: 2015-2016

Derde examenperiode

Samenvatting

Deze bachelorproef is een vergelijkende studie tussen Android Runtime (ART) en zijn voorloper Dalvik. Dit onderzoek bevestigt dat ART wel degelijk meer performant is dan Dalvik bij het uitvoeren van de meeste applicaties.

Dit is van belang voor gebruikers die nog een versie van Android hebben die Dalvik gebruikt. Het gaat hier over ongeveer de helft van alle Android gebruikers. Deze studie zal deze mensen duidelijke resultaten aanbieden zodat ze een goede beslissing kunnen nemen om wel of niet over te schakelen naar een nieuwere versie. De resultaten zijn dus ook belangrijk voor Google, smartphone producenten en meeste ontwikkelaars van Android Applicaties.

Uit de hier uitgevoerde literatuurstudie blijkt dat theoretisch gezien ART meer performant zou moeten zijn dan Dalvik bij het uitvoeren van applicaties. Anderzijds zouden applicaties bij Dalvik sneller geïnstalleerd kunnen worden en zouden ze ook minder plaats innemen in de opslagruimte van de smartphone. Een blik op studies die vergelijkbaar zijn met voorliggend onderzoek, wijst er inderdaad op dat deze vaststellingen in de praktijk duidelijk waarneembaar zijn.

Om deze bevindingen dan in de praktijk uit te testen werden voor dit onderzoek twee Android applicaties ontwikkeld die ontworpen zijn om de performantie van beide runtimes te meten. Eén om de processor te testen en één om de snelheid van de Input/Output (IO) na te gaan. Deze applicaties werden ook gebruikt om de tijd die nodig is om een applicatie te installeren te meten, om na te gaan hoeveel opslaggeheugen er nodig is om de applicaties te installeren en om te bestuderen of er een verschil is in opstartsnelheid tussen de twee runtimes.

In deze bachelorproef worden duidelijke resultaten aangebracht die aanduiden dat ART

inderdaad de meeste applicaties vlotter kan uitvoeren dan Dalvik. Dalvik heeft vooral meer moeite met applicaties die veel berekeningen vergen en dus belastend zijn voor de processor. Anderzijds wordt ook bevestigd dat applicaties bij ART meer opslagruimte innemen en dat het installatieproces sneller gaat bij Dalvik.

Opmerkelijk is wel dat tijdens het onderzoek geen eenduidig antwoord kon gevonden worden op de vraag of de opstarttijd van een applicatie wel degelijk sneller is bij ART dan bij Dalvik. Uit de literatuurstudie komt nochtans duidelijk naar voor dat dit wel het geval zou moeten zijn. Dit zou verder kunnen onderzocht worden tijdens een volgend onderzoek.

De conclusie van dit onderzoek is dat de gebruikers met nog een oudere versie van Android gerust kunnen overschakelen naar een versie die ART gebruikt, tenzij ze veel belang hechten aan het beschikken over veel opslagruimte. Android laat echter tegenwoordig toe om de opslagruimte zodanig uit te breiden dat dit niet meer van zo groot belang is als vroeger.

Voorwoord

Met trots presenteer ik u mijn bachelor scriptie: 'Vergelijkende studie Android: Dalvik vs. Android Runtime (ART)'. Deze studie is uitgevoerd in het kader van mijn opleiding toegepaste informatica aan Hogeschool Gent. Ze is geheel individueel uitgevoerd en ik heb er veel van bijgeleerd, vooral over technische zaken, maar ook over hoe een groot project gepland en uitgevoerd wordt.

Ik heb dit onderwerp gekozen omdat het mij direct aansprak toen ik erover las; ik wou sowieso iets doen rond applicaties en/of Android en dit leek me perfect. Het schrijven van de applicaties en scripts voor het uitvoeren van het onderzoek vond ik dan ook het tofste deel.

Belangrijk is wel dat ik zelf niet op het onderwerp ben gekomen, daarvoor wil ik mijn promotor Jens Buysse bedanken. Samen met hem heb ik ook de onderzoeksvragen en een methodologie uitgewerkt. Hij heeft mij ook geholpen bij het kiezen van welke software ik het best gebruik (zoals TeXnicCenter en JabRef).

Ik wil voornamelijk ook mijn broer bedanken voor het helpen bij de statistische delen en voor het nalezen. Uiteraard zijn mijn ouders ook heel erg bedankt voor de steun en hulp gedurende de hele periode. Verder wil ik Bert Van Vreckem ook bedanken voor het beschikbaar maken van een zeer handig latex sjabloon dat veel heeft geholpen tijdens het schrijven van deze paper.

Met deze scriptie hoop ik een duidelijk antwoord te formuleren op de onderzoeksvragen. Ik wens u als lezer nog veel leesplezier toe.

Tristan Mariën
Dendermonde, 26 augustus 2016

Inhoudsopgave

1	Inleiding	11
1.1	Introductie tot Dalvik en ART	11
1.1.1	Wat is een runtime systeem	12
1.1.2	Verschillende compilatietechnieken	12
1.2	Stand van zaken	15
1.3	Probleemstelling en Onderzoeksvragen	16
1.4	Opzet van deze bachelorproef	17
2	Methodologie	19
3	De testen ontwerpen	21
3.1	De applicaties schrijven	22
3.1.1	CPU applicatie	22
3.1.2	IO applicatie	25

3.2	Installatietijd	25
3.3	Opstarttijd	26
3.4	Tasker	26
3.5	Resultaten analyseren	26
3.6	Gebruikte software en hardware	27
4	Praktijkonderzoek	29
4.1	CPU performantie	29
4.1.1	Binary trees	30
4.1.2	Chameneos Redux	30
4.1.3	Fannkuch Redux	30
4.1.4	Meteor puzzle	31
4.1.5	N body	31
4.1.6	Pidigits	31
4.1.7	Spectral Norm	31
4.1.8	Conclusie	32
4.2	IO performantie	33
4.3	Installatietijden	34
4.4	Opstarttijden	37
4.5	Gebruikte opslagruimte	38
5	Conclusie	41
	Bibliografie	44

Glossarium

De termen die hieronder uitgelegd worden, zijn in deze scriptie onderlijnd.

1. **Bytecode:** Dit is de term die gebruikt wordt als er gepraat wordt over een tussen-taal tussen source code en machine code (of bit code). Deze code wordt meestal uitgevoerd door een virtuele machine.
2. **Debugging:** Dit betekent de fouten uit een programma zoeken en oplossen. Hoewel dit voor een groot deel door de ontwikkelaar moet gebeuren zijn er veel compilers en IDEs (Integrated Development Enviroment) die hierbij kunnen helpen.
3. **Drivers:** Software op een computer die dient om een bepaald hardware component te doen werken, enkele voorbeelden zijn de netwerkdriever voor internet en de display driver om beeld te kunnen tonen.
4. **Emulator:** Een emulator maakt het mogelijk om een fysieke computer na te bootsen als een virtuele machine. Deze virtuele machines zijn niet vergelijkbaar met diegene die in de rest van het onderzoek worden besproken, ze zijn veel uitgebreider en kunnen een besturingssysteem en andere programma's draaien.
5. **Firmware:** Software die in een hardware component geprogrammeerd zit, bijvoorbeeld de BIOS van een pc die ervoor zorgt dat de computer opstart wanneer er op de "aan" knop gedrukt wordt. Ook een goed voorbeeld hiervan is de software in een afstandsbediening van de televisie.
6. **Garbage Collection:** Tijdens het uitvoeren van een programma worden er vaak variabelen of objecten aangemaakt om even te gebruiken en daarna vergeten te worden. Om ervoor te zorgen dat het RAM-geheugen niet vol raakt kan een Garbage Collector zoeken naar zulke "dode" objecten en ze verwijderen.
7. **Hardware:** De fysieke componenten van een computer zoals de processor of een harde schijf.
8. **Input/Output (IO):** IO is een term die gebruikt wordt om alle data te omschrijven

die een programma ontvangt of verstuurt naar de gebruiker of het geheugen van de computer.

9. **Jar files:** Staat voor Java Archive, dit is meestal een verzameling van Java bestanden die door een extern programma kunnen gebruikt worden.
10. **Library:** Een verzameling van bronnen zoals Jar files die kunnen aangeroepen worden door meerdere programma's.
11. **Machine code:** Deze code bestaan enkel uit enen en nullen in een specifieke structuur staan. Hierdoor kan deze code direct uitgevoerd worden door een processor die deze structuur kan lezen.
12. **Native Android applicatie:** Een applicatie die specifiek geschreven is voor Android. Wordt meestal via Google Play Store geïnstalleerd.
13. **Processor (CPU):** Dit is het meest belangrijke deel van elke computer, elk programma bestaat uit enen en nullen die de processor leest en op basis daarvan taken uitvoert zoals een plaats in het geheugen uitlezen of data naartoe schrijven.
14. **Random Access Memory (RAM):** Heel snel toegankelijk geheugen dat gebruikt wordt om data tijdelijk bij te houden. Hier wordt bijvoorbeeld de code van programma's bijgehouden die aan het lopen zijn.
15. **Software:** Alles dat digitaal wordt opgeslagen is software, dit kan bijvoorbeeld een programma zijn of gewoon data.
16. **Source code:** De code van een programma zoals het geschreven is door de ontwikkelaar in een programmeertaal zoals bijvoorbeeld Java. Source code is meestal de meest leesbare versie van de code maar moet vaak eerst gecompileerd worden voor het uitgevoerd kan worden.
17. **Thread:** Programma's kunnen opgesplitst worden in verschillende threads die samen uitgevoerd worden door een processor. De threads van een programma draaien apart, alsof ze elk een verschillend programma zijn, maar ze kunnen wel nog met elkaar communiceren.
18. **Virtuele machine:** De virtuele machines waarover gepraat wordt in dit onderzoek zijn vergelijkbaar met een runtime systeem. Ze zijn voornamelijk verantwoordelijk voor het uitvoeren van code, dit kan machine code of bytecode zijn.

1. Inleiding

Het onderzoek zal bestaan uit een vergelijkende studie tussen de nieuwe Android Runtime (ART) en zijn voorloper Dalvik. Er zal onderzocht worden wat de voordelen en beperkingen van beide runtimes zijn en wat de gevolgen voor de gebruikers ervan zijn.

Hieronder bij hoofdstuk 1.1 is alle informatie te vinden die nodig is om dit onderzoek te begrijpen. Daarna, bij de stand van zaken, zal er gesproken worden over de conclusies van vorige vergelijkbare onderzoeken.

Na de stand van zaken wordt de probleemstelling en de onderzoeksvragen omschreven. Vervolgens wordt er besproken hoe deze bachelorproef opgebouwd is.

1.1 Introductie tot Dalvik en ART

Dit hoofdstuk dient als een introductie tot hoe Android zijn applicaties compileert en uitvoert. De verschillen tussen Dalvik en ART worden hier van uit een theoretisch standpunt uitgelegd. Er wordt ook genoeg informatie verschaft over belangrijke begrippen zodat iedereen die deze bachelorproef wilt lezen alles kan begrijpen. Indien er technische begrippen zijn die niet aan bod komen in dit hoofdstuk kan het glossarium geraadpleegd worden.

De volgende tekst werd samengesteld uit volgende bronnen: [Google, 2014] [Google, 2010] [Mark Stoodley, 2007] [Marshall, 2015] [study.com, 2013]

1.1.1 Wat is een runtime systeem

Een runtime systeem (of runtime environment) kan omschreven worden als een verzameling van software en soms hardware componenten die het mogelijk maken om programma's op een toestel uit te voeren. Dit kan bijvoorbeeld een virtuele machine zijn zoals de Java-Virtual-Machine (JVM) voor Java. Een runtime zal de onderliggende systemen zoals drivers abstracter maken zodat libraries of applicaties ermee kunnen werken.

1.1.2 Verschillende compilatietechnieken

Voor compilers of interpreters bestonden werden programma's direct in machine code (of bitcode) geschreven, dit is de enige taal die een processor (CPU) kan begrijpen en is de meest low level taal die er bestaat. Het is gewoon een verzameling van enen en nullen, samen vormen deze bits een instructie die een zeer specifieke taak uitvoert zoals een adres in het geheugen laden. Machine code is voor een mens heel onleesbaar en op deze manier applicaties ontwikkelen was een moeilijk proces waarbij makkelijk fouten werden gemaakt. Om op een duidelijkere manier te kunnen programmeren werden rond 1950 de eerste compilers gemaakt.

Compiler

Een compiler vertaalt een gegeven programmeertaal naar machine code of naar bytecode zodat deze nadien snel kan uitgevoerd worden. Machine code kan direct door de processor van een systeem uitgevoerd worden, dit is de snelste manier waarop een programma kan draaien. Bytecode moet achteraf nog eens gecompileerd worden naar machine code, of door een interpreter uitgevoerd worden om het programma op een doelapparaat te doen draaien.

De eerste soort compiler was Assembler, een relatief simpele compiler die alleen Assembler kan omzetten naar machine code. Assembler is een low level code taal waarvan elk commando overeen komt met een commando in machine code. Het grootste verschil is dat Assembler leesbaar is door mensen. Een grote applicatie ontwerpen en onderhouden blijft wel moeilijk in een low level taal, daarom zijn er nadien meer complexe compilers gemaakt die high level talen konden omzetten naar machine code. High level programmeertalen zijn leesbaar door mensen en zijn makkelijk te begrijpen, bovendien hebben ze de mogelijkheid om extra functies aan te bieden zoals garbage collection en debugging.

Tegenwoordig wordt er bijna uitsluitend in high level talen geprogrammeerd. Alleen bij systemen waarbij hoge preformantie noodzakelijk is of als er op heel low level met de hardware componenten moet worden gecommuniceerd, wordt er in een low level taal geprogrammeerd. Voorbeelden hiervan zijn drivers en firmware in besturingssystemen.

Interpreter

Een interpreter is een virtuele machine die draait tijdens het uitvoeren van code die niet voordien is gecompileerd naar machine code. De code wordt door een virtuele processor lijn per lijn gelezen en uitgevoerd. Dit is zeer traag omdat de interpreter elke lijn code opnieuw moet analyseren, ook al zou diezelfde lijn meerdere keren kunnen voorkomen. Meestal zal de source code voordien door de ontwikkelaar gecompileerd worden naar bytecode, een tussentaal die sneller is om uit te voeren dan source code. Deze is niet zo snel als machine code maar is meestal wel overdraagbaar tussen verschillende systemen, dit is de belangrijkste reden waarom interpreters populair zijn geworden. Een nadeel van interpreters was vroeger dat ze meer RAM-geheugen in gebruik nemen omdat de hele virtuele machine in het geheugen geladen moet zijn, maar tegenwoordig hebben meeste systemen hier meer dan genoeg geheugen voor.

Statische Compilatie

Bij statische compilatie wordt er een *static build* gemaakt door een compiler. Deze build bestaat uit machine code en symbolen die aan objecten uit een library gelinkt zijn. Een library is een verzameling van veelgebruikte functies of methodes die op voorhand geschreven zijn, een ontwikkelaar kan deze objecten aanroepen bij het schrijven van zijn applicatie. Het linken van deze symbolen aan de bijhorende objecten in de libraries gebeurt bij het compileren van de source code. Na het linken zullen de nodige objecten uit de libraries in het programma zitten, dit noemt statisch linken. Op deze manier heeft het programma alles wat het nodig heeft om uitgevoerd te kunnen worden bij zich en is het toch compact. Men heeft dus enkel de executable (het uitvoerbaar bestand) nodig. Het uitvoeren van een static build is enorm snel omdat het uit machine code bestaat en meteen kan uitgevoerd worden door de processor.

Het grootste probleem bij deze compilatietechniek is dat een statisch gecompileerd programma maar op één soort hardware architectuur kan draaien, bijvoorbeeld maar op één type smartphone. Om een programma op verschillende systemen te doen draaien moet het voor elk type systeem opnieuw gecompileerd worden. Het uitvoerbaar bestand zal hierdoor veel groter worden of er moeten meerdere bestanden gemaakt worden.

Een voorbeeld van statische compilatie zijn meeste iOS applicaties. Er zijn maar een klein aantal apparaten die deze applicaties moeten kunnen uitvoeren dus Apple heeft maar weinig nadeel bij deze compilatietechniek.

Dynamisch Linken

Bij dynamisch linken worden de symbolen pas tijdens de looptijd van het programma gelinkt aan de bijhorende objecten in de libraries. Hierbij zitten de libraries die nodig zijn voor het uitvoeren van het programma in aparte bestanden. Deze bestanden kunnen gedeeld worden door meerdere applicaties, op deze manier moet veel voorkomende code maar één keer opgeslagen zijn op het systeem. Dit kan het updaten van foutieve code ook gemakkelijker maken, enkel het bestand met de library moet vernieuwd worden zonder

dat het programma opnieuw moet gecompileerd worden. Dit zorgt er wel voor dat de applicatie minder betrouwbaar is, deze externe bestanden kunnen onverwacht aangepast worden en er voor zorgen dat het programma niet goed meer werkt. Dit soort programma's uitvoeren is soms ook minder eenvoudig omdat er vaak extra pakketten moeten gedownload of geïnstalleerd worden als deze nog niet op de computer staan.

Een voorbeeld van dynamisch linken is dat programma's die in Java geschreven zijn de Java Runtime Environment (JRE) nodig hebben om uitgevoerd te kunnen worden. Aan de andere kant kan dan ook elk Java programma uitgevoerd worden op elke machine die de Java Runtime Environment ondersteunt.

Just-In-Time

Just-In-Time (JIT) compilatie is een vorm van dynamisch compileren. Dat wilt zeggen dat de source code niet wordt gecompileerd naar machine code door de ontwikkelaar maar nadien pas, bij het laden of het uitvoeren van het programma. Vaak wordt de source code wel eerst door de ontwikkelaar naar bytecode omgezet.

Meestal is JIT compilatie een combinatie van Ahead-Of-Time (AOT) compilatie en een interpreter. Om ervoor te zorgen dat het programma blijft draaien zal er een interpreter de source code of bytecode uitvoeren en ondertussen zal de runtime de applicatie overlopen om te zien welke code veel gebruikt wordt. Deze code wordt dan op voorhand al gecompileerd naar machine code. Als de JIT compiler goed kan voorspellen welke code al gecompileerd moet worden kan het zijn dat de interpreter zelfs niet nodig is.

Op deze manier kan JIT compilatie vergelijkbare performantie behalen als statische compilatie. Dit wordt door voorstanders van statische compilatie bestreden omdat bij JIT compilatie het programma zowel moet uitgevoerd als gecompileerd worden door eenzelfde processor. In de praktijk komt het er vaak op neer dat een JIT gecompileerd programma eerst wat traag verloopt om nadien ongeveer even snel te gaan als een statisch gecompileerd programma. Er is ook meer RAM-geheugen in gebruik bij JIT compilatie omdat zowel de interpreter, de originele source code of de bytecode en de JIT gecompileerde machinecode moeten bijgehouden worden.

Android compilatie

Android is van in het begin ontworpen om op heel veel verschillende apparaten te draaien. Om te vermijden dat elke applicatie voor elke smartphone op voorhand opnieuw gecompileerd moet worden is er voor gekozen om native Android applicaties in Java te schrijven.

Java source code wordt op voorhand gecompileerd in Java bytecode die kan gelezen worden door een Java-Virtual-Machine (JVM). Een JVM gebruikt vaak JIT compilatie om Java code uit te voeren maar er zijn verschillende implementaties die het anders aanpakken zoals ART en de oude versie van Dalvik. Android gebruikt niet de klassieke Oracle JVM maar hun eigen implementatie, de Java bytecode wordt eerst nog omgezet naar Dalvik bytecode in een dex bestand (dex staat voor Dalvik Executable). Alle Java-Archive (JAR) bestanden

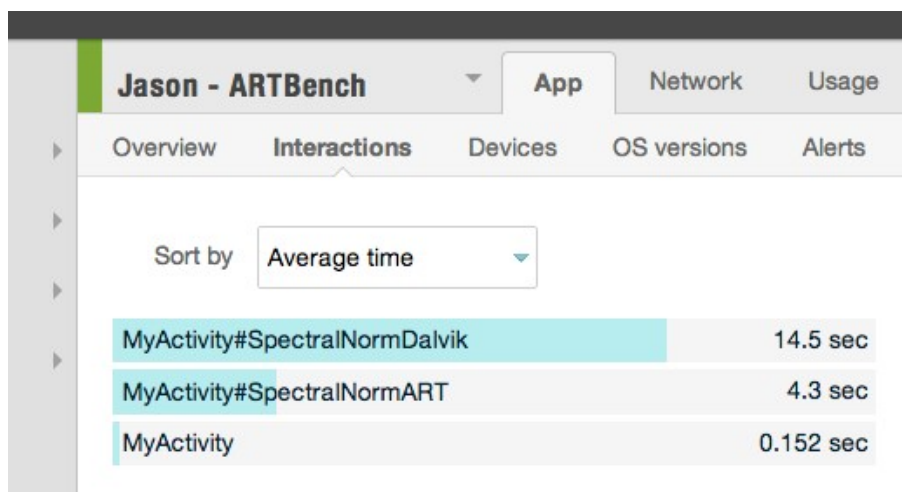
die het programma nodig hebben zitten ook in dit dex bestand. Het dex bestand wordt dan samen met de resource bestanden (zoals afbeeldingen) in een Android-Application-Package (APK) gestoken. Dit is een type van zip bestand, waarin de data gecomprimeerd worden om plaats te besparen. Een APK is klaar om geïnstalleerd en uitgevoerd te worden op een Android toestel, hoe dit precies gebeurt hangt af van de Android versie.

Vóór Android 2.2 Froyo gebruikte Dalvik enkel een interpreter om de Dalvik bytecode uit te voeren. Om de performantie te verbeteren werd er vanaf Froyo gebruik gemaakt van JIT compilatie.

Met Android 5.0 Lollipop werd Dalvik vervangen door Android Runtime (ART). ART gebruikt een vorm van AOT compilatie, hij zal namelijk de bytecode al compileren tijdens de installatie van de applicatie. Op deze manier verhoogt de performantie tijdens het uitvoeren van de applicatie omdat de processor enkel machine code moet uitvoeren. Twee opvallende nadelen hiervan zijn dat de installatie langer duurt en dat de applicatie meer geheugen in beslag zal nemen omdat de gecompileerde code ook zal moeten opgeslagen worden. Mogelijk heeft Google deze compilatietechniek gekozen omdat het beschikbare geheugen op smartphones nu groot genoeg is dat men zich hier niet zoveel zorgen over moet maken als vroeger.

1.2 Stand van zaken

Er zijn al een paar studies vergelijkbaar met deze te vinden, bijvoorbeeld dit artikel van New Relic door Jason [Snell, 2014]. Om een goede test applicatie te schrijven haalt hij code van dezelfde bron als omschreven in hoofdstuk 3.1.1: namelijk van de website van [Gouy, 2016]. Hij gebruikt echter maar één test, de Spectral Norm en hij laat deze test maar tien keer lopen.



The screenshot shows the 'Jason - ARTBench' application interface. It has a top navigation bar with 'App', 'Network', and 'Usage' tabs. Below this is a sub-navigation bar with 'Overview', 'Interactions', 'Devices', 'OS versions', and 'Alerts'. The 'Interactions' tab is selected. A 'Sort by' dropdown menu is set to 'Average time'. The main content area displays a table with three rows of performance data.

Test Name	Average time
MyActivity#SpectralNormDalvik	14.5 sec
MyActivity#SpectralNormART	4.3 sec
MyActivity	0.152 sec

Figuur 1.1: Resultaat van New Relic artikel [Snell, 2014]

Net zoals in deze scriptie wordt er een grote performantie-winst gevonden voor ART

bij het uitvoeren van de Spectral Norm test. Dit kan wel aan de lezers een misleidend beeld geven, zoals de Jason zelf zegt, wordt er enkel één test uitgevoerd op de processor. Maar bovendien, als er gekeken wordt naar de resultaten in hoofdstuk 4.1 waar zeven verschillende testen op de processor uitgevoerd worden, schiet het performantie verschil tussen ART en Dalvik net bij de Spectral Norm test er ver bovenuit. Andere testen op de processor tonen een veel kleiner (hoewel toch nog substantieel) performantie verschil. De conclusie in het artikel van Jason is dus niet foutief, maar zoals eerder vermeld, kunnen de lezers mogelijk een vervormd beeld krijgen van het verschil tussen ART en Dalvik.

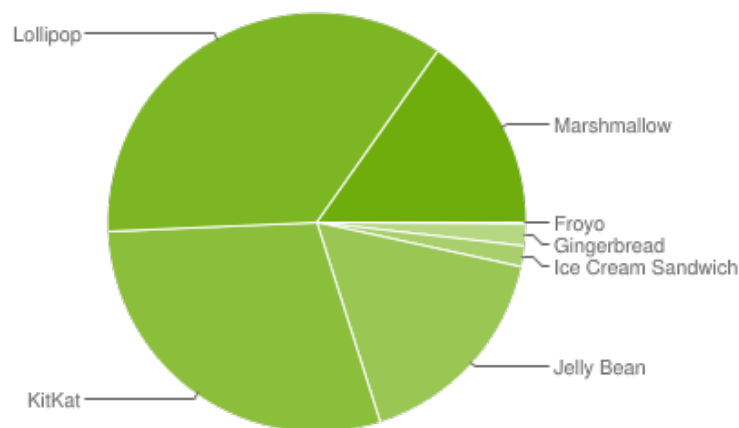
In een ander onderzoek door Tobias [Konradsson, 2015] worden er drie dingen getest: opslagruimte, RAM-verbruik en algemene performantie. De resultaten voor hoeveel opslagruimte een applicatie inneemt is volgens Tobias ongeveer gelijk met de resultaten in hoofdstuk 4.5 van deze scriptie. Bij beide wordt er een relatie gevonden tussen hoeveel groter de applicatie is op ART dan op Dalvik en hoeveel code ze bevat. Applicaties met veel data zoals ruwe tekst of afbeeldingen tonen een veel kleiner verschil in grootte.

Om de algemene performantie bij het uitvoeren van een applicatie te meten, wordt door Tobias echter benchmark applicaties gebruikt die te vinden zijn op de Google Play Store. Deze applicaties worden vaak gebruikt bij het vergelijken van verschillende smartphones. Er wordt bij het gebruiken van deze applicaties echter vertrouwd op de kennis van degene die deze applicatie geschreven heeft, er is geen manier om te weten wat of hoe er precies getest wordt.

De resultaten van Tobias zijn opnieuw vergelijkbaar met de resultaten omschreven in hoofdstuk 4.1 van deze scriptie. Al werd er een minder groot performantie verschil gevonden bij Tobias. Opmerkelijk is wel dat volgens Tobias Dalvik beter is dan ART als het over Input/Output (IO) gaat. Bij hoofdstuk 4.2 van deze scriptie is hier echter geen bewijs van te zien, misschien zelfs het tegenovergestelde. Tobias omschrijft de IO taken als: het bestandssysteem aanspreken en database operaties uitvoeren. Zoals eerder vermeld kan er helaas niet met zekerheid gezegd worden wat soort tests er precies uitgevoerd zijn.

1.3 Probleemstelling en Onderzoeksvragen

In Android 5.0 Lollipop werd er een nieuwe runtime geïntroduceerd die een veel hogere performantie zou geven dan zijn voorloper Dalvik. Nieuwe smartphones zijn echter vaak duur, dus de beslissing om een nieuwe te kopen is niet vanzelfsprekend. Momenteel gebruikt ongeveer de helft (49,3% [Google, 2016]) van de Android gebruikers nog een versie die Dalvik gebruikt. Deze studie zal deze mensen duidelijke resultaten aanbieden zodat ze een goede beslissing kunnen nemen om wel of niet over te schakelen naar een nieuwere versie. Aangezien sommige fabrikanten geen of weinig updates van Android aanbieden betekent dit in de praktijk wel vaak de aanschaf van een nieuwe smartphone.



Figuur 1.2: Grafiek van het aantal gebruikers per Android versie [Google, 2016]

Het is voor ontwikkelaars van Android applicaties van belang dat gebruikers zo snel mogelijk een nieuwere versie van Android aanschaffen. Want, als ze beslissen om een applicatie te maken die alleen op Android 5.0 of hoger werkt dan kan momenteel bijna de helft van de gebruikers deze nieuwe applicatie niet eens openen. Uiteraard hebben de producenten van Android smartphones en Google zelf er ook voordeel bij wanneer mensen een nieuwe Android smartphone kopen.

De onderzoeksvragen die zullen beantwoord worden in dit onderzoek zijn:

“Biedt Android Runtime (ART) een hogere performantie bij het uitvoeren van applicaties dan Dalvik?”

“Wat zijn de voor- en nadelen van AOT (Ahead-Of-Time) compilatie tegenover JIT (Just-In-Time) compilatie?”

1.4 Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:

In Hoofdstuk 2 wordt de methodologie toegelicht en worden de gebruikte onderzoekstechnieken besproken om een antwoord te kunnen formuleren op de onderzoeksvragen.

In Hoofdstuk 3 wordt in detail uitgelegd hoe en waarom de testen en applicaties ontworpen werden. Er wordt ook overlopen welke software en hardware er gebruikt werd tijdens het onderzoek.

In Hoofdstuk 4 worden de resultaten van de testen geanalyseerd en geïnterpreteerd om een goed besluit te kunnen vormen.

In Hoofdstuk 5, tenslotte, wordt de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvragen. Daarbij wordt ook een aanzet gegeven voor toekomstig onderzoek

binnen dit domein.

2. Methodologie

Er werd begonnen met een onderzoeksfase waarin beschikbare literatuur die relevant is voor de onderzoeksvragen onderzocht werd. Op basis van documentatie van Google zelf en verschillende onderzoeken werd een antwoord gevormd op de onderzoeksvragen.

Om deze bevindingen dan in de praktijk uit te testen werden een aantal Android applicaties ontwikkeld. Met deze applicaties werden een aantal testen uitgevoerd op beide runtimes. Door de tijden bij te houden die de runtimes nodig hadden om deze testen uit te voeren werden de data verzameld waarop dit onderzoek gebaseerd is. Deze tijden werden geanalyseerd om tot een conclusie te komen over de voor- en nadelen van beide runtimes.

Naast het meten van de tijden werd er ook gekeken naar hoeveel ruimte de applicaties innemen. Tot slot werden de behaalde resultaten vergeleken met de verwachtingen uit het literatuuronderzoek en de conclusies van andere onderzoeken.

Er werd voor dit onderzoek gekozen om zelf applicaties te schrijven in plaats van reeds bestaande benchmark applicaties te gebruiken. Op deze manier was er absolute controle over wat er precies getest werd en hoe dit gebeurde. Bij een andere applicatie zou er vertrouwd moeten worden op de ontwikkelaar ervan. Bovendien kan iedereen zo een applicatie downloaden en uitvoeren. De resultaten van dit onderzoek kunnen niet nagebootst worden zonder bijna identiek dezelfde applicaties te schrijven.

In hoofdstuk 3 wordt in detail uitgelegd hoe de applicaties geschreven werden, alsook hoe en waarom de testen ontworpen zijn.

3. De testen ontwerpen

De applicaties die geschreven zijn om de verschillen in performantie tussen ART en Dalvik in kaart te brengen moeten getest worden in zo identiek mogelijke omstandigheden. In Android KitKat (versie 4.4) heeft de gebruiker de kans om te wisselen tussen Dalvik en ART. Dit is helaas maar een bètaversie van ART waar de focus ligt op het zo weinig mogelijk bevatten van bugs en de gebruikers een kans te geven ART eens uit te proberen en feedback te geven. [Google, 2014] Deze versie van de runtime is nog niet volledig geoptimaliseerd naar performantie toe en is dus niet interessant voor deze studie. Pas in Android Lollipop (versie 5.0) werd ART de standaard runtime.

De ideale situatie zou zijn dat de applicaties op twee exact dezelfde smartphones uitgevoerd worden waarbij één smartphone op Android KitKat draait en de andere op Android Lollipop. Smartphones die Android Lollipop hebben zijn echter niet goedkoop. Daarnaast is het niet vanzelfsprekend om twee verschillende Android versies op twee dezelfde soorten smartphone te hebben. Er is namelijk geen officiële manier om een smartphone te downgraden naar een lagere Android versie. Tijdens het verloop van deze studie moeten de smartphones ook enkel voor deze studie gebruikt worden, er moeten zo weinig mogelijk extra applicaties geïnstalleerd zijn zodat de testresultaten niet beïnvloed worden door achtergrond processen.

Om deze redenen is er voor deze studie gebruik gemaakt van twee emulators om de applicaties uit te voeren. Emulators bootsen de hardware en software van fysieke toestellen na. Op deze manier moet er geen rekening gehouden worden met eventueel verschillende hardware, de emulators kunnen gewoon door dezelfde pc uitgevoerd worden. Er is gebruik gemaakt van de virtual mobile device emulator die standaard bij de Android SDK zit en beide emulators bootsen de hardware van een LG Nexus 5 na, één met Android api 4.4.4 (KitKat) en de andere met Android api 5.1.1 (Lollipop).

3.1 De applicaties schrijven

Bij het ontwerpen van de applicaties werd er besloten om twee applicaties te schrijven: de CPU applicatie om de processor te testen en de IO applicatie die meet hoe snel er data kan ingelezen/weggeschreven worden (Input/Output). Elke test wordt vijftig keer uitgevoerd zodat er genoeg data is om statistisch gezien een juiste conclusie te kunnen trekken.

3.1.1 CPU applicatie

Om de algemene performantie van de processor te meten werd een applicatie geschreven die een aantal algoritmes uitvoert. Deze algoritmes zijn gemaakt om programmeertalen te benchmarken en belasten de processor door lange wiskundige berekeningen uit te voeren. De applicatie voert de algoritmes na elkaar uit en houdt bij hoeveel tijd er nodig was om elk algoritme uit te voeren.

De algoritmen gebruikt in dit onderzoek zijn afkomstig van de website behorende tot [Gouy, 2016]. Deze site biedt een aantal kleine programma's aan die ontworpen zijn om de performantie van verschillende programmeertalen te benchmarken. Om zo goed mogelijke metingen te kunnen maken werd voor dit onderzoek een aantal interessante programma's in een applicatie verzameld. Deze programma's werden voor dit onderzoek enkel gewijzigd om ze te doen werken in een Android applicatie.

Hieronder volgt een omschrijving van de zeven gebruikte programma's in de CPU applicatie.

Binary trees

De code van dit deel van de applicatie is bijgedragen door Heikki Salokanto en is aangepast door Chandra Sekar en Mike Krüger. Deze code is in functie van dit onderzoek enkel aangepast om te werken in een Android applicatie.

Deze test maakt een groot aantal binaire bomen aan, controleert of ze effectief bestaan door het onderste element op te vragen en verwijdert ze dan. In het begin wordt er ook één grote binaire boom aangemaakt die in de loop van de test in het geheugen blijft. Op het einde van de test wordt ook deze boom gecontroleerd. De parameter die wordt meegegeven bij het starten van de test bepaalt de maximale diepte van de binaire bomen.

Chameneos Redux

De code van dit deel van de applicatie is bijgedragen door Michael Barker en is aangepast door Daryl Griffith. Deze code is in functie van dit onderzoek enkel aangepast om te werken in een Android applicatie.

Deze test simuleert een aantal dieren, “chameneos”, die drie verschillende kleuren kunnen hebben. Deze dieren gaan herhaaldelijk naar een gemeenschappelijke ontmoetingsplaats

om daar elkaar te ontmoeten, of om te wachten tot er iemand is om te ontmoeten. Bij een ontmoeting veranderen de dieren van kleur afhankelijk van de kleur van de chameneos die hij ontmoet. Elk dier is een aparte thread, per test wordt dit scenario een keer uitgevoerd met drie dieren (één van elke kleur) en een keer met tien dieren. De parameter die wordt meegegeven bij het starten van de test bepaalt hoeveel ontmoetingen er gebeuren elke keer dat het scenario wordt gesimuleerd.

Fannkuch Redux

De code van dit deel van de applicatie is bijgedragen door Isaac Gouy en is aangepast door Oleg Mazurov. Deze code is in functie van dit onderzoek enkel aangepast om te werken in een Android applicatie.

Bij deze test wordt er een reeks getallen genomen van één tot en met het getal dat als parameter aan de test werd meegegeven 1, ..., n. Voor elke mogelijke volgorde van die getallen wordt er berekend hoeveel stappen er nodig zijn om te eindigen met een één vooraan als het volgende proces gevolgd wordt: er wordt naar het eerste getal gekeken, als dit bijvoorbeeld een drie is, wissel dan de eerste drie getallen om van plaats.

Bijvoorbeeld uit [Gouy, 2016]:

Als de test wordt uitgevoerd met als parameter vijf: {1, 2, 3, 4, 5}.

Een mogelijke volgorde is dan: {4, 2, 1, 5, 3}

Dit begint met een vier dus worden de eerste vier getallen omgewisseld: {5, 1, 2, 4, 3}

Dit wordt herhaald tot er een één vooraan staat: {3, 4, 2, 1, 5} -> {2, 4, 3, 1, 5}

-> {4, 2, 3, 1, 5} -> {1, 3, 2, 4, 5}

Voor deze volgorde waren er dus vijf stappen nodig.

Meteor puzzle

De code van dit deel van de applicatie is bijgedragen door Amir K aka Razii en is gemaakt op basis van code door Ben St. John en Michael Deardeuff. Deze code is in functie van dit onderzoek enkel aangepast om te werken in een Android applicatie.

De meteor puzzle bestaat uit een veld van vijf op tien hexadiagonale velden. Het programma moet dit veld proberen opvullen met tien puzzelstukken van vijf velden groot, de puzzelstukken mogen gelijk welke vorm hebben.



Figuur 3.1: Mogelijke oplossing voor de meteor puzzle [Erwin Vervae, 2002]

De parameter die wordt meegegeven bij het starten van de test bepaalt hoeveel oplossingen er gezocht zullen worden.

N body

De code van dit deel van de applicatie is bijgedragen door Mark C. Lewis en is aangepast door Chad Whipkey. Deze code is in functie van dit onderzoek enkel aangepast om te werken in een Android applicatie.

Bij deze test wordt de baan van de vier reuzenplaneten in ons zonnestelsel gevolgd: Jupiter, Saturnus, Uranus en Neptunus. De parameter die wordt meegegeven bij het starten van de test bepaalt hoelang de planeten gevolgd moeten worden.

Pidigits

De code van dit deel van de applicatie is bijgedragen door Isaac Gouy. Deze code is in functie van dit onderzoek enkel aangepast om te werken in een Android applicatie.

Bij deze test worden de cijfers na de komma van pi berekend, het aantal berekende cijfers hangt af van de parameter die is meegegeven bij het starten van de test.

Spectral Norm

De code van dit deel van de applicatie is bijgedragen door Jarkko Miettinen en is aangepast door Isaac Gouy. Deze code is in functie van dit onderzoek enkel aangepast om te werken in een Android applicatie.

Deze test zal de spectral norm van een oneindige matrix berekenen als deze wordt toegepast op een vector. De parameter die wordt meegegeven aan deze test bepaalt hoe lang de vector is.

De oneindige matrix ziet er als volgt uit:

$a_{11}=1$, $a_{12}=1/2$, $a_{21}=1/3$, $a_{13}=1/4$, $a_{22}=1/5$, $a_{31}=1/6$, etc

De vector is: $[1, 1, \dots, 1]$

Een matrix is een manier om lineaire transformaties uit te voeren op datasets. De spectral norm is in essentie de grootste factor in zo een transformatie. Om de spectral norm beter te kunnen uitleggen wordt het volgende voorbeeld genomen. Als dataset wordt er een afbeelding genomen, als deze afbeelding zouden gedraaid, uitgerekt of ingekort zou worden, dan wordt er een lineaire transformatie op uitgevoerd. De spectral norm is dus de grootste factor die op de afbeelding inspeelt. Als de afbeelding bijvoorbeeld twintig graden wordt gekanteld, in de breedte tot dubbel zo breed wordt uitgerekt, in de lengte een tiende kleiner wordt en nog eens veertig graden wordt gekanteld, dan zal de spectral norm gelijk zijn aan twee omdat hij in de breedte met een factor van twee werd uitgerekt.

3.1.2 IO applicatie

Om te meten hoe snel de runtime data kan inlezen en naar het schrijfgeheugen wegschrijven werd de IO (Input/Output) applicatie gemaakt die een grote tekst moet wegschrijven naar een nieuw bestand. De applicatie heeft als resource een tekstbestand met ongeveer 2,31MB (2 422 846 bytes) aan willekeurig gegenereerde Lorem Ipsum tekst. De applicatie maakt een bestand aan in het schrijfgeheugen en kopieert daar de tekst lijn per lijn naartoe. Dit gebeurt 50 keer. Net voordat het bestand wordt aangemaakt houdt de applicatie de huidige tijd in milliseconden bij als de begintijd. Nadat de tekst 50 keer in het bestand gekopieerd is wordt het bestand verwijderd, dan wordt opnieuw de huidige tijd bijgehouden, deze keer als eindtijd. Het verschil tussen de eindtijd en de begintijd wordt beschouwd als de tijd die het programma nodig had om de IO test te voltooien.

3.2 Installatietijd

Om de installatietijd te meten werd er voor zowel de IO en de CPU applicatie een klein script geschreven dat automatisch 50 keer een applicatie op de gestarte emulator zal installeren en dan terug verwijderen. De scripts houden ook de tijd bij die nodig was voor beide runtimes, in een formaat dat kan verwerkt worden door het Java programma dat alle resultaten analyseert.

3.3 Opstarttijd

De tijd die de applicaties nodig hebben om op te starten wordt gemeten door de applicatie de tijd in milliseconden te laten bijhouden bij het afsluiten. De volgende instantie van de applicatie zal die tijd dan aftrekken van de huidige tijd om een idee te krijgen over hoelang de applicatie nodig heeft om opnieuw op te starten. Hierbij valt wel op te merken dat deze tijd de som is van de tijd die nodig om de applicatie eerst af te sluiten en de tijd om hem terug op te starten. Omdat de eerste keer dat de applicaties worden opgestart er geen afsluittijd is om mee te werken werd deze test in praktijk maar 49 keer uitgevoerd in plaats van 50.

3.4 Tasker

Tasker is een Android applicatie die de gebruiker toelaat om taken te starten bij bepaalde events zoals bijvoorbeeld wanneer er een nieuwe notificatie is of wanneer de smartphone opstart. In dit onderzoek wordt Tasker gebruikt om de test applicaties opnieuw op te starten wanneer ze zichzelf afsluiten. Er wordt in Tasker een teller bijgehouden die ervoor zorgt dat na 50 keer dit proces stopt zodat er geen oneindige cyclus ontstaat.

3.5 Resultaten analyseren

Tijdens het uitvoeren van de applicaties worden de resultaten in de Logcat console geprint bij de Android Monitor van Android Studio. Deze resultaten staan in een bepaalde structuur: ze beginnen met “Result” om ze makkelijk te kunnen filteren in Logcat, dan komt de naam van de test zoals “Install” of “BinaryTrees 17” met als slot het aantal milliseconden waarin de test volbracht werd. Alle testresultaten zijn te vinden in de bijlage van deze scriptie.

Om de resultaten in deze vorm te kunnen analyseren is er een Java programma ontwikkeld dat de output van de applicaties leest en verwerkt. Alle resultaten worden in een categorie geplaatst per test die uitgevoerd werd. Nadat ze zo gegroepeerd zijn worden de volgende statistisch significante gegevens berekend:

- n: het aantal keer dat deze test is uitgevoerd.
- average: het gemiddelde.
- standard deviation: de standaard deviatie, dit bepaalt hoe ver de meeste resultaten verspreid liggen tegenover het gemiddelde. Als de standaard deviatie laag is dan zijn de resultaten dicht bij het gemiddelde gegroepeerd.
- standard error of the mean: De “juistheid” van het gemiddelde, als het laag is dan wilt dit zeggen dat het gemiddelde waarschijnlijk dicht bij het “ware” gemiddelde van de resultaten.
- range: het verschil tussen het hoogste resultaat en het kleinste. Zoals de standaard deviatie bepaalt dit hoever de resultaten zijn verspreid, maar dit is veel minder betrouwbaar omdat het teveel beïnvloed wordt door mogelijke uitschieters.
- median: de mediaan.

- lower quartile: het laagste kwartiel.
- upper quartile: het bovenste kwartiel.
- interquartile range: de interkwartielafstand, het verschil tussen het bovenste kwartiel en het laagste kwartiel. Dit wordt meestal gebruikt om een box plot grafiek te maken.
- variance: de variatie, dit heeft hetzelfde nut als de standaard deviatie. In dit onderzoek wordt het enkel gebruikt om de standaard deviatie uit te berekenen omdat die dezelfde eenheden als de resultaten gebruikt, in dit geval milliseconden.

Met deze gegevens kan per test bestudeerd worden of er een statistisch significant verschil is tussen Dalvik en ART bij de uitvoering van de testen. Zo zal er een duidelijke conclusie getrokken worden op het einde van dit onderzoek. Het verder analyseren van de testresultaten gebeurt in hoofdstuk 4.

3.6 Gebruikte software en hardware

De voornaamste software die gebruikt werd tijdens dit onderzoek:

- Voor het schrijven van de applicaties en het uitvoeren van de emulators werd Android Studio 1.5.1 gebruikt. De emulators liepen op een Windows 10 (pro) pc met een Intel(R) Core(TM) i5-4460 CPU @ 3.20GHz en 8GB ram.
- Er is gebruik gemaakt van de virtual mobile device emulator die standaard bij de Android SDK zit en beide emulators bootsen de hardware van een LG Nexus 5 na, één met Android api 4.4.4 (KitKat) en de andere met Android api 5.1.1 (Lollipop).
- Tasker (versie 4.7m) wordt gebruikt om de applicaties meerdere keren na elkaar te lanceren op de emulators.
- Voor het programmeren van het Java programma dat de resultaten analyseert werd NetBeans IDE 8.1 gebruikt.
- Het schrijven van deze paper gebeurde in TeXnicCenter versie: 2.02 Stable (32 bit) dankzij het aanraden van de promotor Jens Buysse.
- Om de gebruikte bronnen bij te houden en te organiseren werd JabRef 2.10 gebruikt, ook dankzij Jens Buysse.

4. Praktijkonderzoek

Het vergelijken van de testresultaten zal telkens vooraf gegaan worden door een Student's t -test die zal duidelijk maken of er een statistisch significant verschil tussen de meetresultaten bestaat. Hiervoor moet er een waarde α die de onbetrouwbaarheidsdrempel aanduidt gekozen worden. Voor dit onderzoek is er besloten te werken met een betrouwbaarheid van 95%, of een α van 0,05. De t -test zal een p -waarde als resultaat geven, als deze p -waarde kleiner is dan de α dan kunnen we met 95% zekerheid zeggen dat er een statistisch significant verschil te vinden is tussen de testresultaten.

Een probleem met deze werkwijze is dat in dit onderzoek vijftien verschillende t -testen uitgevoerd zullen worden. Met een betrouwbaarheid van 95% voor elke test wilt dit zeggen dat ongeveer één op de twintig testen foutief zullen zijn. Om dit tegen te gaan en er voor te zorgen dat er een echt zekerheid level van 95% kan behouden worden over het hele onderzoek: dit noemt men de Family Wise Error Rate (FWER). Om de FWER te beperken op 5%, zal er gebruik gemaakt worden van de Bonferroni correctie. Deze simpele maar effectieve procedure zegt enkel dat de α gedeeld moet worden door het aantal testen dat uitgevoerd zal worden. Dit betekent dat er zal gewerkt worden met een α van 0,00333...

De grafieken in dit hoofdstuk tonen de behaalde testresultaten,

4.1 CPU performantie

Om deze resultaten betrouwbaar te maken werd er voor gezorgd dat er zo weinig mogelijk achtergrondtoepassingen of processen aan het werken waren. Op de pc waarop de emulators draaien werden programma's zoals browsers (Firefox of Chrome) en antivirussen uitgeschakeld tijdens het uitvoeren van de applicaties. Dankzij de processen en de perfor-

mantie tabs in Windows taakbeheer kon er ook makkelijk in het oog gehouden worden dat de emulators genoeg toegang hadden tot de processor en het RAM-geheugen.

De Android machines die op de emulators draaien zijn net voor het uitvoeren van de tests geïnstalleerd en ook hier werd er zoveel mogelijk voor gezorgd dat er geen overbodige applicaties in de achtergrond draaien. De emulators hadden geen toegang tot wifi, gps of mobiele data.

4.1.1 Binary trees

Tabel 4.1: *t*-test voor resultaten Binary trees

runtime	Dalvik	ART
gemiddelde	11276,42ms	5802,46ms
standaard deviatie	123,13ms	205,33ms
aantal testen	50	50
p-waarde	0,0001	

4.1.2 Chameneos Redux

Tabel 4.2: *t*-test voor resultaten Chameneos Redux

runtime	Dalvik	ART
gemiddelde	9952,12ms	6342,34ms
standaard deviatie	189,72ms	221,72ms
aantal testen	50	50
p-waarde	0,0001	

4.1.3 Fannkuch Redux

Tabel 4.3: *t*-test voor resultaten Fannkuch Redux

runtime	Dalvik	ART
gemiddelde	8588,74ms	5926,04ms
standaard deviatie	47,21ms	332,94ms
aantal testen	50	50
p-waarde	0,0001	

4.1.4 Meteor puzzle

Tabel 4.4: *t*-test voor resultaten Meteor puzzle

runtime	Dalvik	ART
gemiddelde	6210,42ms	3414,06ms
standaard deviatie	94,53ms	37,41ms
aantal testen	50	50
p-waarde	0,0001	

4.1.5 N body

Tabel 4.5: *t*-test voor resultaten N body

runtime	Dalvik	ART
gemiddelde	3962,5ms	2468,8ms
standaard deviatie	30,54ms	30,41ms
aantal testen	50	50
p-waarde	0,0001	

4.1.6 Pidigits

Tabel 4.6: *t*-test voor resultaten Pidigits

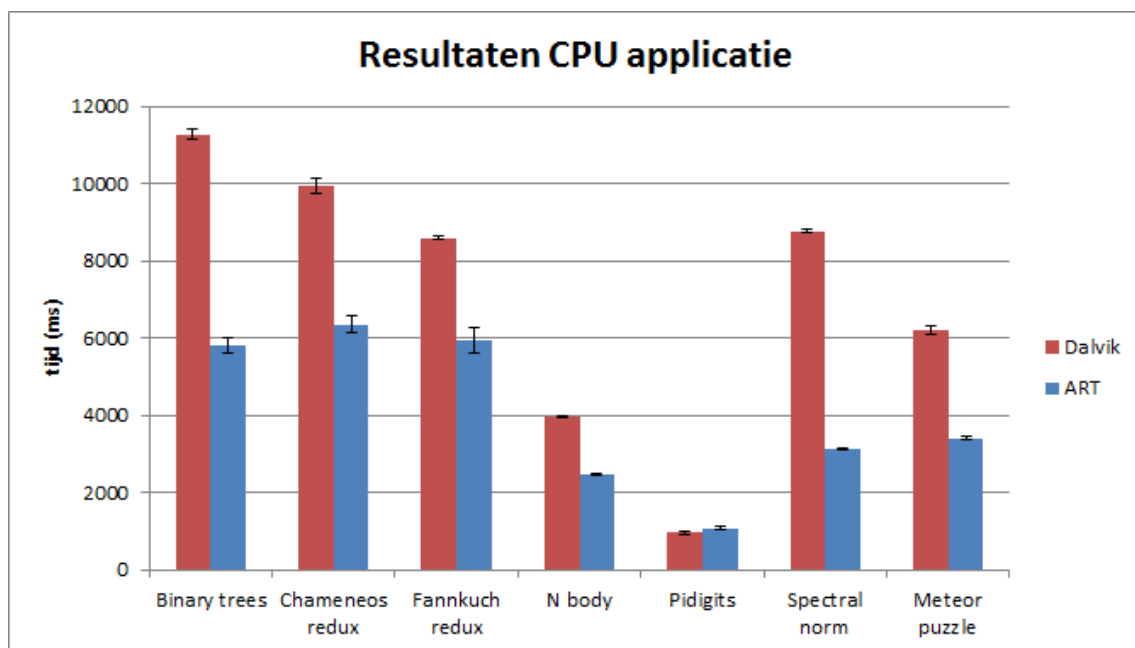
runtime	Dalvik	ART
gemiddelde	976,78ms	1080,1ms
standaard deviatie	42,68ms	44,45ms
aantal testen	50	50
p-waarde	0,0001	

4.1.7 Spectral Norm

Tabel 4.7: *t*-test voor resultaten Spectral Norm

runtime	Dalvik	ART
gemiddelde	8769,06ms	3128,36ms
standaard deviatie	49,42ms	35,04ms
aantal testen	50	50
p-waarde	0,0001	

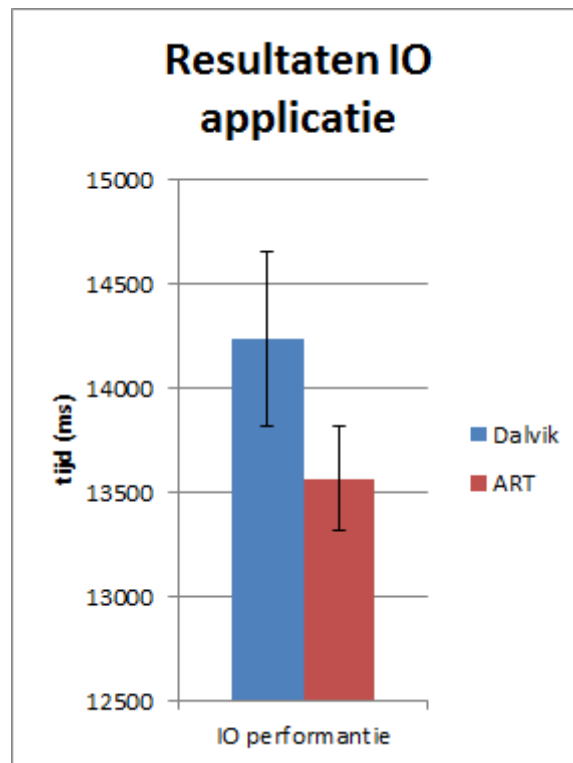
4.1.8 Conclusie



Figuur 4.1: Resultaten CPU applicatie

Elke p-waarde is kleiner dan de α , er is dus bij elk onderdeel van de CPU test een statistisch significant verschil gevonden. Buiten de Pidigits test waren alle tests in het voordeel van ART, ART presteerde gemiddeld ook 72,76% beter dan Dalvik. Het blijkt dat ART's Ahead-Of-Time compilatie effectief is om op veel vlakken zijn performantie omhoog te halen. Of toch wanneer het gaat over taken die belastend zijn voor de processor zoals zware berekeningen.

4.2 IO performantie



Figuur 4.2: Resultaten IO applicatie

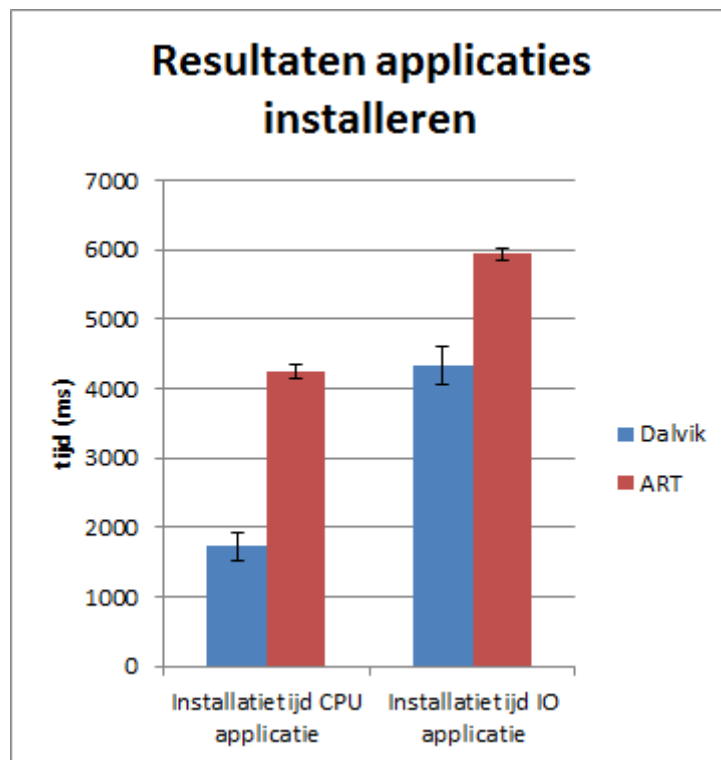
Tabel 4.8: *t*-test voor resultaten IO applicatie

runtime	Dalvik	ART
gemiddelde	14235,56ms	13567ms
standaard deviatie	414,42ms	247,36ms
aantal testen	50	50
p-waarde	0,0001	

De p-waarde is kleiner dan de α , er is dus wel degelijk een statistisch significant verschil tussen de uitvoering van de IO applicatie bij de twee runtimes. Het gaat echter maar over een klein verschil. ART voerde de IO test gemiddeld maar 4,93% sneller uit dan Dalvik. Dit komt waarschijnlijk niet door een verschil in performantie bij het wegschrijven en/of het lezen van data in het geheugen maar eerder door een imperfectie in de test. De IO applicatie is gebouwd voor het testen van hoe snel er een hoeveelheid ruwe data in tekst naar de schijf kan weggeschreven worden. De applicatie vertrouwt er op dat er een bottleneck zal ontstaan bij het schrijven naar het geheugen en dat de CPU meestal dus geen zwaar werk moet doen. Maar de CPU blijft een belangrijke rol spelen bij het uitvoeren van de applicatie. Dit wil zeggen dat de performantie van de CPU nog steeds een invloed heeft op de resultaten en voordien is er aangetoond dat ART in het algemeen een betere

performantie heeft dan Dalvik. Het relatief klein tijdsverschil tussen de twee runtimes bij het uitvoeren van deze test komt dus waarschijnlijk niet doordat ART beter is dan Dalvik in het wegschrijven of ophalen van data.

4.3 Installatietijden



Figuur 4.3: Resultaten applicaties installeren

Tabel 4.9: *t*-test voor installatietijd CPU applicatie

runtime	Dalvik	ART
gemiddelde	1724,7ms	4238,34ms
standaard deviatie	193,41ms	93,63ms
aantal testen	50	50
p-waarde	0,0001	

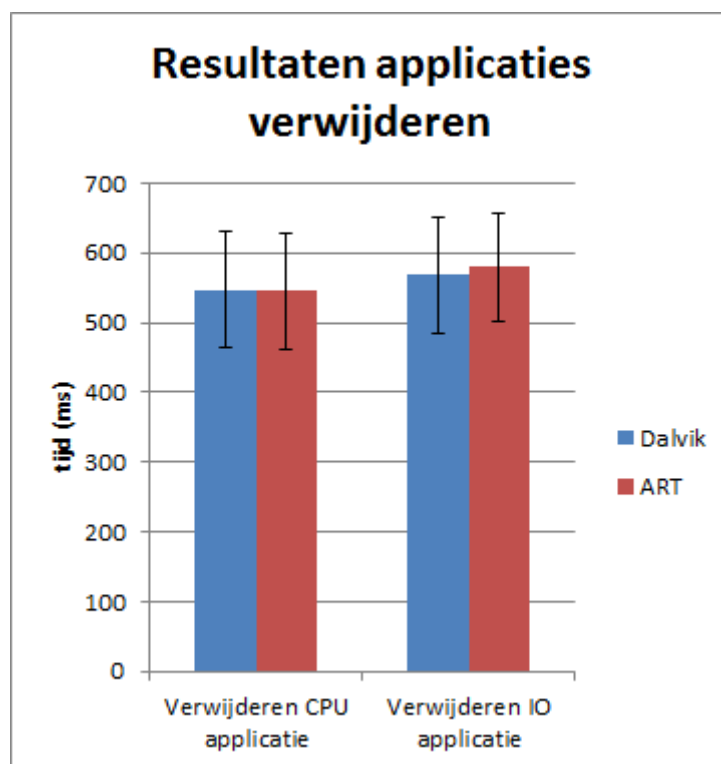
Tabel 4.10: *t*-test voor installatietijd IO applicatie

runtime	Dalvik	ART
gemiddelde	4332,98ms	5935,12ms
standaard deviatie	263,5ms	89,27ms
aantal testen	50	50
p-waarde	0,0001	

Beide p-waarden zijn lager dan de α , het installeren van de applicaties gaat dus zoals verwacht vlotter bij Dalvik. Waarschijnlijk komt dit doordat ART tijdens de installatie al de code gaat compileren tot machine code.

Er is wel een opmerkelijk verschil tussen de resultaten van de CPU applicatie en de IO applicatie. Bij de IO applicatie gaat het over een gemiddeld verschil van 36,98% en bij de CPU applicatie 145,74%. Dit verschil komt waarschijnlijk door dezelfde reden dat ART in dit geval trager is dan Dalvik. ART compileert de code van de applicaties al tijdens het installeren maar de IO applicatie bestaat grotendeels uit ruwe data. Deze data moeten niet gecompileerd worden maar enkel uitgepakt worden uit het APK bestand, dit zal vermoedelijk ongeveer evenveel tijd innemen bij beide runtimes.

Dit verschil tussen de twee applicaties toont wel aan dat de installatiesnelheid bij ART eerder afhangt van hoeveel code de applicatie bevat, niet alleen hoe groot die is. Dit wordt bevestigd door de grootte van de applicaties te vergelijken. De apk van de CPU applicatie (CPU_App.apk) is 1140KB en de IO applicatie (IO_App.apk) is 4496KB, bijna vier keer zo groot, of 294,39% groter om precies te zijn. Maar de gemiddelde installatietijd van de IO applicatie bij ART is maar 40,03% sneller dan bij Dalvik.



Figuur 4.4: Resultaten applicaties verwijderen

Tabel 4.11: *t*-test voor verwijderen CPU applicatie

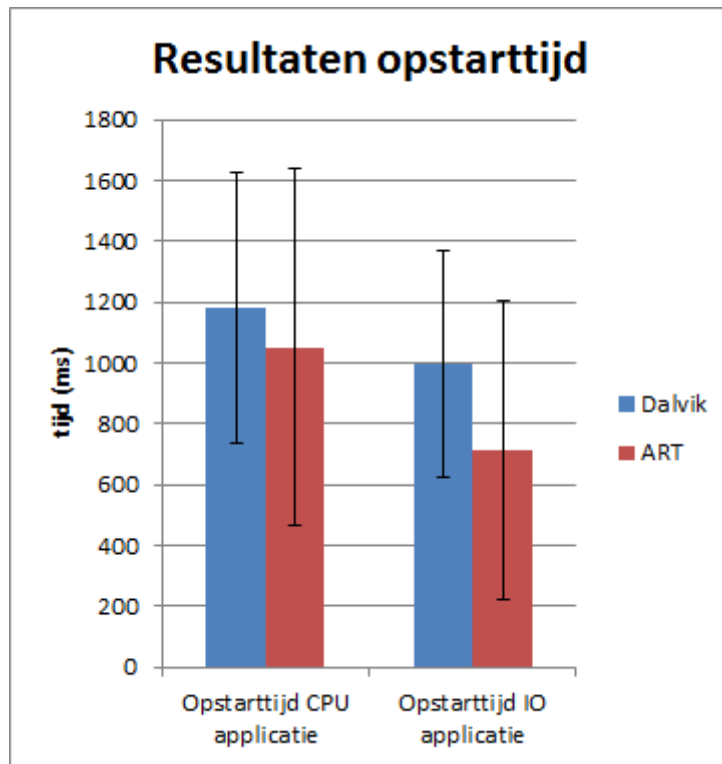
runtime	Dalvik	ART
gemiddelde	547,08ms	545,1ms
standaard deviatie	82,65ms	83,26ms
aantal testen	50	50
p-waarde	0,9052	

Tabel 4.12: *t*-test voor verwijderen IO applicatie

runtime	Dalvik	ART
gemiddelde	568,34ms	579,56ms
standaard deviatie	83,64ms	78,28ms
aantal testen	50	50
p-waarde	0,4902	

De gemiddelde tijd die beide runtimes nodig hebben om de applicatie te verwijderen verschilt niet veel. Deze p-waarden zijn hoger dan de α , er kan dus niet besloten worden dat ART sneller of trager is dan Dalvik in het verwijderen van een applicatie.

4.4 Opstarttijden



Figuur 4.5: Resultaten opstarttijd

Bij de metingen van de opstarttijden van de CPU applicatie valt er op dat er zowel bij Dalvik en ART een grote spreiding is van de resultaten. Met een standaard deviatie van respectievelijk 447,68ms en 583,89ms en gemiddelden die zo dicht bij elkaar liggen zal er waarschijnlijk geen statistisch significant verschil aangetoond kunnen worden. Door de resultaten op het oog te bestuderen lijkt het wel dat ART een iets minder voorspelbare opstarttijd heeft, in de zin dat ART af en toe iets sneller is. Dalvik heeft als snelste tijd 665ms waartegenover ART 16 van de 49 keer de applicatie sneller dan die tijd heeft kunnen opstarten. Beide runtimes konden buiten enkele uitschieters de applicatie telkens opstarten onder de 2 seconden.

Tabel 4.13: *t*-test voor opstarttijd CPU applicatie

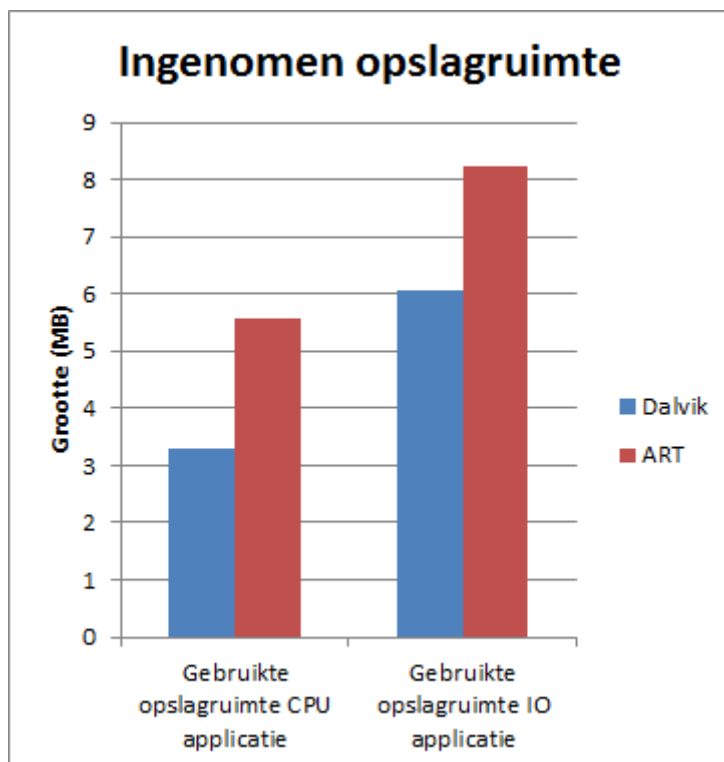
runtime	Dalvik	ART
gemiddelde	1181,33ms	1052,16ms
standaard deviatie	447,68ms	583,89ms
aantal testen	49	49
p-waarde	0,2221	

Tabel 4.14: *t*-test voor opstarttijd IO applicatie

runtime	Dalvik	ART
gemiddelde	994,41ms	715,49ms
standaard deviatie	371,66ms	491,97ms
aantal testen	49	49
p-waarde	0,0021	

Verrassend genoeg is er bij de IO applicatie dus wel een statistisch significant verschil. ART start de applicatie gemiddeld 38,98% sneller op dan Dalvik. Dit is het resultaat dat kon verwacht worden uit de literatuurstudie. Dalvik gaat met zijn Just-In-Time compilatie bij het opstarten van de applicatie proberen voorspellen welke delen van de code als volgende gaat uitgevoerd worden en deze dan compileren. ART heeft dit bij de installatie van de applicatie al gedaan en zou dus vooral bij het starten van de applicatie sneller moeten kunnen werken dan Dalvik. Er kon in deze studie niet verklaard worden waarom dit verschil alleen merkbaar was bij de IO applicatie, en niet bij de CPU applicatie.

4.5 Gebruikte opslagruimte



Figuur 4.6: Ingenomen opslagruimte

Tabel 4.15: Gebruikte opslagruimte

runtime	Dalvik	ART	ART/Dalvik
CPU applicatie	3,31MB	5,57MB	68,28%
IO applicatie	6,05MB	8,25MB	36,36%

Zoals voorspeld neemt een applicatie meer opslagruimte in bij ART dan bij Dalvik. Dit komt opnieuw doordat ART de applicatie tijdens de installatie al zal compileren en de machine code bijhouden. De CPU applicatie is 68,28% groter bij ART en de IO applicatie is maar 36,36% groter. Dit is logisch omdat de IO applicatie vooral uit ruwe data bestaat, deze data wordt niet gecompileerd naar machine code en moet dus niet bijgehouden worden. Net zoals bij de installatietijden is ook de opslagruimte nodig voor een applicatie dus voor een groot deel afhankelijk van hoeveel code de applicatie bevat.

5. Conclusie

Uit het onderzoek bleek dat in het algemeen Android Runtime (ART) wel degelijk een betere performantie biedt dan Dalvik bij het uitvoeren van applicaties. Vooral bij applicaties die vaak berekeningen moeten doen was er een groot verschil in snelheid terug te vinden. Applicaties die meer data ophalen en wegschrijven zonder er berekeningen op te doen zullen een minder grote performantie-winst ondervinden, bijvoorbeeld een file manager.

Deze hogere performantie bij het uitvoeren van een applicatie is ART's grootste troef, het is dan ook waar veel gebruikers zich op zullen focussen. Zoals verwacht werden er echter ook een paar nadelen van ART gevonden, zoals de tijd die nodig is om een applicatie te installeren en de hoeveelheid opslagruimte die de applicatie inneemt. Beide zijn een stuk beter bij Dalvik omdat Dalvik pas bij het uitvoeren van de applicatie begint te compileren. Er moet wel mee rekening gehouden worden dat een applicatie normaal maar één keer moet geïnstalleerd worden, een betere performantie tijdens het uitvoeren ervan is dus een stuk belangrijker. Tegenwoordig is de opslagruimte van een smartphone ook zodanig uitbreidbaar dat het niet meer zo erg is als voordien als een applicatie meer plaats inneemt.

Op basis van dit onderzoek kunnen de gebruikers die nog een oudere versie van Android gebruiken wel degelijk met een gerust hart overschakelen naar een versie die ART gebruikt. Zeker als er een geheugenkaartje wordt bijgekocht om de opslagruimte wat uit te breiden.

Waar het onderzoek geen duidelijk antwoord op heeft kunnen geven is of applicaties sneller opstarten in ART dan in Dalvik. Naast het feit dat er bij de IO applicatie een onverklaarbaar groter verschil zit dan bij de CPU applicatie, was er ook een te grote variatie in de testresultaten. Mogelijk zou er een vervolg onderzoek kunnen komen dat hier een beter antwoord op kan geven. De volgende aspecten van een runtime die in dit onderzoek niet aan bod zijn geweest zouden ook kunnen bestudeerd worden in een volgend

onderzoek:

- Biedt ART een betere performantie bij het uitvoeren van zware grafische taken?
- Heeft een smartphone die op ART draait een betere batterijduur dan bij Dalvik?
- Maakt ART efficiënter gebruik van het beschikbare RAM (geheugen)?

Bibliografie

- [Erwin Vervae, 2002] Erwin Vervae, M. D. C. (2002). Java optimization techniques. <http://www.dsc.ufcg.edu.br/jacques/cursos/2004.2/gr/recursos/j-javaopt.pdf>.
- [Google, 2010] Google (2010). Google i/o 2010 - a jit compiler for android's dalvik vm. Youtube. <https://www.youtube.com/watch?v=Ls0tM-c4Vfo>.
- [Google, 2014] Google (2014). Google i/o 2014 - the art runtime. Youtube. <https://www.youtube.com/watch?t=141&v=EBITzQsUoOw>.
- [Google, 2016] Google (2016). Dashboards | android developers. <https://developer.android.com/about/dashboards/index.html>.
- [Gouy, 2016] Gouy, I. (2016). The computer language benchmarks game. <http://benchmarksgame.alioth.debian.org/>.
- [Konradsson, 2015] Konradsson, T. (2015). Art and dalvik performance compared. <http://www.diva-portal.org/smash/get/diva2:852736/FULLTEXT01.pdf>.
- [Mark Stoodley, 2007] Mark Stoodley, Kenneth Ma, M. L. (2007). Real-time java, part 2: Comparing compilation techniques. <http://www.diva-portal.org/smash/get/diva2:852736/FULLTEXT01.pdf>.
- [Marshall, 2015] Marshall, S. (2015). Static and dynamic libraries. http://pewpewthespells.com/blog/static_and_dynamic_libraries.html.
- [Snell, 2014] Snell, J. (2014). Android runtime performance analysis using new relic: Art vs. dalvik. <https://blog.newrelic.com/2014/07/07/android-art-vs-dalvik/>.

[study.com, 2013] study.com (2013). Machine code and high-level languages: Using interpreters and compilers. <http://study.com/academy/lesson/machine-code-and-high-level-languages-using-interpreters-and-compilers.html>.

Lijst van figuren

1.1	Resultaat van New Relic artikel (Snell, 2014)	15
1.2	Grafiek van het aantal gebruikers per Android versie (Google, 2016)	17
3.1	Mogelijke oplossing voor de meteor puzzle (Erwin Vervaet, 2002)	24
4.1	Resultaten CPU applicatie	32
4.2	Resultaten IO applicatie	33
4.3	Resultaten applicaties installeren	34
4.4	Resultaten applicaties verwijderen	35
4.5	Resultaten opstarttijd	37
4.6	Ingenomen opslagruimte	38

Lijst van tabellen

4.1	<i>t</i> -test voor resultaten Binary trees	30
4.2	<i>t</i> -test voor resultaten Chameneos Redux	30
4.3	<i>t</i> -test voor resultaten Fannkuch Redux	30
4.4	<i>t</i> -test voor resultaten Meteor puzzle	31
4.5	<i>t</i> -test voor resultaten N body	31
4.6	<i>t</i> -test voor resultaten Pidigits	31
4.7	<i>t</i> -test voor resultaten Spectral Norm	31
4.8	<i>t</i> -test voor resultaten IO applicatie	33
4.9	<i>t</i> -test voor installatietijd CPU applicatie	34
4.10	<i>t</i> -test voor installatietijd IO applicatie	34
4.11	<i>t</i> -test voor verwijderen CPU applicatie	36
4.12	<i>t</i> -test voor verwijderen IO applicatie	36
4.13	<i>t</i> -test voor opstarttijd CPU applicatie	37
4.14	<i>t</i> -test voor opstarttijd IO applicatie	38
4.15	Gebruikte opslagruimte	39