Performance-Analyse von nativen und plattformübergreifenden Entwicklungsansätzen bei typischen UI-Interaktionen

Masterarbeit

zur Erlangung des akademischen Grades

Master of Science in Engineering (M.Sc.)

Eingereicht bei:

Fachhochschule Kufstein Tirol Bildungs GmbH

Web Communication & Information Systems

Verfasser/in:

Paula Engelberg, BSc

1810738255

Erstgutachter : Stefan Huber, MA

Zweitgutachter : Prof. (FH) Dr. Johannes Lüthi

Abgabedatum:

6. November 2020

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich die vorliegende Masterarbeit selbständig und ohne fremde Hilfe verfasst und in der Bearbeitung und Abfassung keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe. Die vorliegende Masterarbeit wurde noch nicht anderweitig für Prüfungszwecke vorgelegt.

Wien, 6. November 2020

Paula Engelberg, BSc

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

API Application Programming Interface

CPU Central Processing Unit

CSS Cascading Style Sheets

DSL Domain Specific Language

FFI Foreign Function Interface

GPS Global Positioning System

GPU Graphics Processing Unit

HDD Hard Disk Drive

HTML HyperText Markup Language

HTTPS HyperText Transfer Protocol Secure

IDE integrierte Entwicklungsumgebung

integrierte Entwicklungsumgebungen

JS JavaScript

M2M Model-to-Model

MB-UID Model-Based User Interface Development

MDA Model-Driven Architekture

MDD Model-Driven Development

MDE Model-Driven Engineering

MDSD Model-Driven Software Development

MD-UID Model-driven User Interface Development

MiB Mebibyte

OMG Object Management Group

PIM Platform Independent Model

PSM Platform Specific Model

PWA Progressive Web App

RAM Random Access Memory

SDK Software Development Kit

SSD Solid State Drive

UI User Interface

URL Uniform Resource Locator

VM Virtual Machine

FH Kufstein Tirol
Web Communication & Information Systems

Abstract of the thesis: **Performance-Analyse von nativen und plattformübergreifenden Entwicklungsansätzen bei typischen UI-Interaktionen**

Author: Paula Engelberg, BSc
First reviewer: Stefan Huber, MA
Second reviewer: Prof. (FH) Dr. Johannes Lüthi

This master thesis deals with the topic of cross-platform development approaches for mobile applications for the two largest mobile platforms Android and iOS as well as performance analysis of typical UI interactions. The following two research questions are addressed: *How can the performance of cross-platform development approaches be measured uniformly* and *How does the performance of cross-platform development approaches differ when considering typical UI interactions?* To answer the research questions, a comprehensive theoretical introduction was given at the beginning to provide the necessary basic knowledge for the practical part. For the practical part five mobile applications were implemented. These included three cross-platform applications developed with React Native, Flutter and Ionic/Capacitor and additionally, two native applications for Android and iOS. Afterwards, the CPU usage and memory consumption was measured during the execution of three typical UI interactions on both Android and iOS devices. The analyzed UI interactions include the *opening and closing of the Navigation Drawer*, the *transition between two screens*, and the *scrolling through a virtual list*. The research showed that there is no framework or tool that allows the measurement of resources on both Android and iOS devices in a consistent manner. The test runs were therefore performed using two different tools. The results show that the cross-platform applications consistently consume more CPU than their native counterparts. The memory consumption on Android depends on the interaction and on iOS the Ionic/Capacitor applica-

tion requires the same amount of memory as the native application. The React Native application consumes significantly more.

6. November 2020

FH Kufstein Tirol
Web Communication & Information Systems

Kurzfassung der Masterarbeit: **Performance-Analyse von nativen und plattformübergreifenden Entwicklungsansätzen bei typischen UI-Interaktionen**

Verfasser: Paula Engelberg, BSc
Erstgutachter: Stefan Huber, MA
Zweitgutachter: Prof. (FH) Dr. Johannes Lüthi

Die vorliegende Masterarbeit befasst sich mit dem Thema der plattformübergreifenden Entwicklungsansätze für mobile Anwendung für die beiden größten mobilen Plattformen Android und iOS sowie der Performanz-Analyse typischer UI-Interaktionen. Dazu werden die folgenden zwei Forschungsfragen bearbeitet: *Wie kann die Performanz von plattformübergreifenden Entwicklungsansätzen einheitlich gemessen werden?* sowie *Wie unterscheidet sich die Performanz von plattformübergreifenden Entwicklungsansätzen unter der Nutzung typischer UI-Interaktionen?* Um die Forschungsfragen zu beantworten, wurde am Anfang eine umfassende theoretische Einführung geben, um das notwendige Basiswissen für den praktischen Teil zu erarbeiten. Für den praktischen Teil wurden fünf mobile Anwendungen implementiert. Diese beinhalteten drei plattformübergreifende Anwendungen, welche mit React Native, Flutter und Ionic/-Capacitor entwickelt wurden und zusätzlich, zwei native Anwendungen für Android und iOS. Danach wurde der Ressourcenverbrauch (CPU und RAM) während der Ausführung von drei typischen UI-Interaktionen sowohl auf einem Android- als auch einem iOS-Gerät gemessen. Zu den untersuchten UI-Interaktionen zählen das *Öffnen und Schließen des Navigation Drawers*, der *Übergang zwischen zwei Bildschirmen* sowie das *Scrollen durch eine virtuelle Liste*. Die Recherche zeigte, dass es noch kein Framework oder Tool gibt, welches Ressourcenmessungen sowohl auf einem Android- als auch einem iOS-Gerät einheitlich zulässt. Die Testdurchläufe wurden daher mit zwei unterschiedlichen

Tools durchgeführt. Die Ergebnisse zeigen, dass die plattformübergreifend programmierten Anwendungen durchwegs mehr CPU in Anspruch nehmen, als die nativen Gegenstücke. Der Arbeitsspeicherverbrauch ist auf Android von der Interaktion abhängig und auf iOS benötigt die Ionic/Capacitor Anwendung gleich viel Speicherplatz wie die native Anwendung. Die React Native Anwendung verbraucht deutlich mehr.

6. November 2020

1. Einführung

Im Jahr 2019 besaßen rund 3,2 Milliarden Menschen ein Smartphone (Newzoo, 2019) und mobile Anwendungen sind mittlerweile aus unserem Leben, ob privat oder beruflich, kaum mehr wegzudenken. Mit der Zeit haben sich zwei mobile Betriebssysteme aus dem Angebot herauskristallisiert und sich deutlich von den Restlichen abgesetzt. Google Android und Apple iOS kommen gemeinsam auf einen Marktanteil von 99,42 % (StatCounter, 2020). Beide Betriebssysteme sind von Grund auf verschieden und Anwendungen, welche für eines der beiden Betriebssysteme entwickelt wurden, sind auf dem jeweils anderem nicht ohne Weiteres Zutun lauffähig. Die Entwicklung für ein bestimmtes Betriebssystem wird native Entwicklung genannt. Da es jedoch in vielen Fällen zu zeitaufwändig und kostspielig ist, eine Anwendung nativ für beide Betriebssysteme zu entwickeln, sind Technologien entstanden, welche eine plattformübergreifende Entwicklung ermöglichen. Diese Technologien ermöglichen die Entwicklung einer Anwendung unter Verwendung einer einzelnen Codebasis, aus welcher eine lauffähige Anwendung für beide Betriebssysteme erzeugt werden kann. Mittlerweile gibt es jedoch eine nahezu unüberschaubare Anzahl an Technologien und Frameworks, die eine plattformübergreifende Entwicklung mit unterschiedlichen Entwicklungsansätzen ermöglichen. Diese beruhen auf unterschiedlichen Ansätzen und bringen dementsprechend unterschiedliche Vor- und Nachteile mit sich. Sich im Vorfeld der Entwicklung für das passende Framework zu entscheiden, kann mitunter ein Problem darstellen, wenn nicht genügend Wissen über die Performanz des Frameworks zur

Verfügung steht. An diesem Punkt setzt diese Masterarbeit an und untersucht den Ressourcenverbrauch (Central Processing Unit (CPU) und Random Access Memory (RAM)) von drei typischen User Interface (UI)-Interaktionen von mobilen Anwendungen, welche mit drei unterschiedlichen Frameworks beziehungsweise Entwicklungsansätzen implementiert wurden. Die Ergebnisse wurden in weiterer Folge mit dem Ressourcenverbrauch der jeweiligen nativen Implementierung verglichen. Alle Untersuchungen wurden sowohl auf einem Android- als auch iOS-Gerät durchgeführt. Das erste Kapitel dieser Masterarbeit zeigt die Probleme auf, welche sich im Zuge der Entwicklung einer mobilen Anwendung ergeben. In diesem Zusammenhang werden sowohl die Zielsetzung als auch die Forschungsfrage dargestellt. Zum Abschluss wird die Methodik der Arbeit erläutert.

1.1 Problemstellung

Statistiken zufolge werden im Jahr 2020 etwa 3,5 Milliarden Menschen ein Smartphone besitzen und für das Jahr 2021 ist ein weiterer Anstieg von 300 Millionen prognostiziert (Newzoo, 2019). Dies führt auch zu einem Anstieg der angebotenen Anwendungen in den beiden führenden App Stores. Im dritten Quartal 2020 können Android-User aus 2,87 Millionen Anwendungen und iOS-User aus 1,96 Millionen auswählen (Appfigures and VentureBeat, 2020a). 2019 wurden aus beiden App Stores etwa 116 Milliarden Anwendungen heruntergeladen und im Jahr 2024 sollen es voraussichtlich etwa 184 Milliarden sein (Tower and TechCrunch, 2020). Betrachtet man zusätzlich noch die weltweiten Einnahmen aus mobilen Anwendungen, welche sich im Jahr 2018 auf 365,2 Milliarden US-Dollar belaufen, so wird klar, wie wichtig Anwendungen für Unternehmen sind. In den nächsten drei Jahren sollen die Einnahmen sogar noch auf das über 2,5-fache von 2018 ansteigen (iResearch, 2019).

Unternehmen, die eine Anwendung entwickeln lassen wollen, können heute

zwischen mehreren Entwicklungsansätzen wählen. Einerseits können Anwendungen nativ entwickelt werden was jedoch bedeutet, wenn die beiden größten mobilen Plattformen bedient werden sollen, dass sowohl eine Anwendung für Android als auch eine für iOS separat entwickelt werden muss. Da in diesem Fall zwei Codebasen gepflegt werden müssen, bringt dies dementsprechend einen deutlich höheren Aufwand mit sich. Um dieses Problem der doppelten Codebasen zu umgehen, gibt es heute eine Fülle von unterschiedlichen plattformübergreifenden Entwicklungsansätzen, welche es ermöglichen aus einer einzigen Codebasis mehrere mobile Anwendungen für mehrere mobile Plattformen bereitzustellen. Das Hauptkonzept plattformübergreifender Entwicklung besteht also darin, die Anwendung einmal zu entwickeln und sie überall ausführen zu können. Hinter plattformübergreifenden Entwicklungsansätzen verbergen sich unterschiedliche Technologien, welche weiter klassifiziert werden können. In dieser Masterarbeit wird die Klassifizierung nach Biørn-Hansen et al. (2018) verwendet. Biørn-Hansen et al. (2018) unterscheidet fünf Kategorien plattformübergreifender Entwicklungsansätze: hybride, interpretierte, cross-kompilierte, Modell-getriebene Entwicklung und Progressive Web Apps. All diese Ansätze sowie der native Entwicklungsansatz verwenden unterschiedliche Techniken und haben auch unterschiedliche Anforderungen an die mobilen Geräte, auf welchen sie aufgeführt werden. In der Literatur herrscht die Meinung vor, dass Anwendungen, welche mit plattformübergreifenden Entwicklungsansätzen implementiert wurden, im Gegensatz zu nativen Implementierungen mehr Ressourcen verbrauchen. Dabei wurden aber zumeist rechenintensive Aufgaben betrachtet und nicht UI-Interaktionen. Diese Interaktionen wie beispielsweise der Seitenwechsel oder das Swipen durch Listen, kommen hauptsächlich bei der Bedienung von Anwendungen zum Einsatz.

Auf die aktuelle Studienlage wird im Kapitel 2.3 genauer eingegangen. UI-Interaktionen sind ein wichtiger Teil der Anwendungsnutzung und gehören daher ebenso auf Ressourcenverbrauch untersucht. Diese Arbeit widmet sich

der Performanceanalyse von UI-Interaktionen von plattformübergreifend entwickelten Anwendungen. Studien von Huber and Demetz (2019) und Huber et al. (2020) auf Android zeigen, dass die getesteten plattformübergreifenden Anwendungen mehr Ressourcen beim Ausführen von UI-Interaktionen verbrauchen, als die nativ implementierte Android-Anwendung. Diese Masterarbeit widmet sich der Untersuchung des Ressourcenverbrauchs (CPU und RAM) von drei UI-Interaktionen sowohl auf einem Android- als auch einem iOS-Gerät. Die Resultate der beiden nativen Anwendungen dienen als Bezugspunkt für die Ergebnisse der plattformübergreifenden Anwendungen.

1.2 Zielsetzung und Forschungsfragen

Um den Ressoucenverbrauch der UI-Interaktionen testen zu können, wurden fünf Anwendungen entwickelt, welche die gleichen drei Interaktionen beinhalten. Das Ziel dieser Arbeit ist es herauszufinden, wie sich die einzelnen Entwicklungsansätze unter der Nutzung typischer UI-Interaktionen unterscheiden und wie der Ressourcenverbrauch dieser Entwicklungsansätze gemessen und in weiterer Folge verglichen werden kann. Daraus ergeben sich zwei Forschungsfragen:

Forschungsfrage 1: *Wie kann die Performanz von plattformübergreifenden Entwicklungsansätzen einheitlich gemessen werden?*

Forschungsfrage 2: *Wie unterscheidet sich die Performanz von plattformübergreifenden Entwicklungsansätzen unter der Nutzung typischer UI-Interaktionen?*

1.3 Aufbau der Arbeit

In dieser Masterarbeit wird das Thema des Ressourcenverbrauchs der plattformübergreifenden Entwicklungsansätze für mobile Anwendungen, sowohl auf der Android- als auch iOS-Plattform, betrachtet. Dazu werden im theoretischen Teil die unterschiedlichen Entwicklungsansätze vorgestellt, um ein Verständnis für den praktischen Teil der Arbeit zu vermitteln. Danach werden die verwandten Forschungsarbeiten zum Thema der Ressourcenmessung aufgearbeitet. Im praktischen Teil der Arbeit wird die technische Umsetzung der Studie vorgestellt. Dabei wird zuerst auf die Auswahl der Entwicklungsansätze eingegangen, gefolgt von der Vorstellung der ausgewählten Frameworks, welche für diese Masterarbeit ausgesucht wurden. Anschließen werden die drei ausgewählten UI-Interaktionen beschrieben und die Implementierungsdetails für jede Anwendung vorgestellt. Darauf folgt die Beschreibung der automatisierten Testfälle sowie die Bewertungskriterien, Messwerkzeuge und Testgeräte. Im vierten Kapitel werden die Testergebnisse der Messungen ausführlich dargestellt. Im fünften Kapitel werden die Ergebnisse diskutiert und die Einschränkungen besprochen. Den Abschluss dieser Masterarbeit bietet das Fazit und der Ausblick für zukünftige Arbeiten.

2. Grundlagen

Das zweite Kapitel dieser Masterarbeit gibt eine Einführung in die Entwicklung mobiler Anwendungen mit dem Ziel, Verständnis für den praktischen Teil dieser Arbeit zu vermitteln. Den Beginn macht hierbei eine Einführung in die native Entwicklungsvariante, gefolgt von der plattformübergreifenden Entwicklung mobiler Anwendungen. Ziel ist es, dem Leser die Einzelheiten und Unterschiede der beiden Entwicklungsansätze aufzuzeigen. Anschließend wird ein Überblick über das Thema der Ressourcenmessung bei mobilen Anwendungen gegeben und schließlich wird auf die Interaktionen mit mobilen Anwendungen eingegangen.

2.1 Native Entwicklung

Native Anwendungen werden für ein konkretes mobiles Betriebssystem entwickelt. Sie bauen auf Programmiersprachen auf, die nur von einem bestimmten mobilen Betriebssystem unterstützt werden. Dieser Entwicklungsansatz verwendet die Programmiersprachen, Werkzeuge und integrierte Entwicklungsumgebung (IDE) (Xanthopoulos and Xinogalos, 2013), welche speziell für die Programmierung von Anwendungen für ein konkretes Betriebssystem entwickelt wurden. Das bedeutet, dass Anwendungen, die für Android entwickelt wurden, auf Geräten unterschiedlicher Hersteller - welche das Android Betriebssystem nutzen - funktionieren. Anwendungen, welche für iOS entwickelt

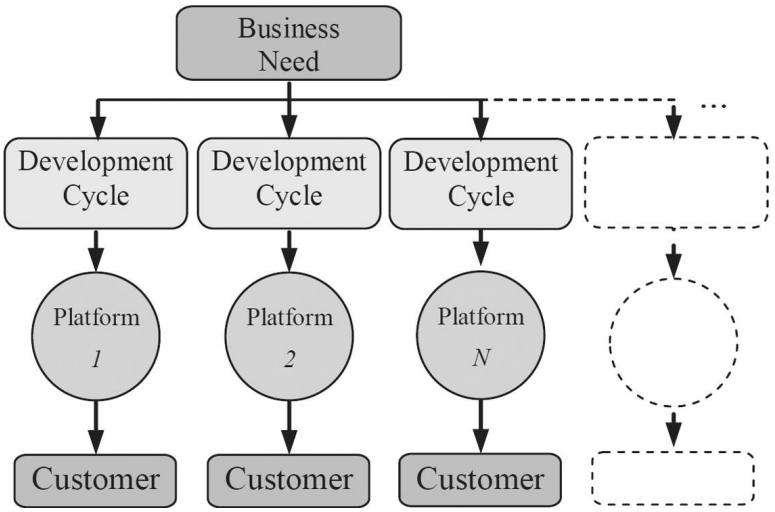

Abbildung 1: Traditionelles Entwicklungsmodell[1]

wurden, hingegen nur auf iOS-Geräten ausführbar sind (Hall and Anderson, 2009). In Abbildung 1 wird schematisch der native Entwicklungsprozess dargestellt. Möchte man eine Anwendung für beide Plattformen entwickeln, muss der gesamte Entwicklungsprozess für jede Plattform wiederholt und angepasst werden.

Die Anwendungen für Android werden mit der Programmiersprache Java und/oder Kotlin vorzugsweise in der IDE Android Studio und dem Build-Tool Gradle programmiert. Für iOS wird Objective-C (Smutny, 2012; Heitkötter and Majchrzak, 2013), oder Swift 3.0 in der integrierten Entwicklungsumgebung Xcode verwendet (siehe Tabelle 1).

Native mobile Anwendungen können nach ihrer Entwicklung und Veröffentlichung direkt aus dem entsprechenden App Store heruntergeladen und auf dem Smartphone installiert werden. Die Anwendung wird in weiterer Folge direkt von der Laufzeitumgebung des mobilen Betriebssystems ausgeführt.

[1] Darstellung entnommen aus Corral et al. (2012)

Betriebssystem	Programmiersprache	Entwicklungsumgebung	App Store
Android	Java, Kotlin	Android Studio	Google Play
iOS	Objective-C, Swift	Xcode	App Store

Tabelle 1: Unterschiede zwischen den beiden mobilen Betriebssystemen aus Sicht der Entwicklung

Vorteile nativer Programmierung

Anwendungen, welche nativ entwickelt wurden, haben Zugriff auf alle Hardwareressourcen und gerätespezifischen Sensoren. Aus diesem Grund sind native Anwendungen performanter und ressourcensparender als andere Entwicklungsansätze. Dementsprechend sind sie eine gute Wahl für komplexe, rechenintensive und performante mobile Anwendungen mit hochwertiger User Experience (Vollmer, 2017).

Als Vorteil kann vor allem die effiziente Ausführung des Quellcodes und die daraus resultierende reichhaltigste Benutzererfahrung (Xanthopoulos and Xinogalos, 2013) genannt werden. Dadurch, dass jede Plattform ihre einzigartigen Elemente und eine eigene Designsprache besitzt, ist es möglich, den Benutzerinnen und Benutzern das Gefühl der Einheit zu geben. Dies bedeutet, dass die Designelemente mit welchen der Benutzer interagiert, sauber aufeinander abgestimmt sind, was in weiterer Folge die intuitive Navigation und Bewegung durch die Anwendung erleichtert.

Nachteile nativer Programmierung

Wenn ein Unternehmen eine Anwendung für mehr als ein Betriebssystem entwickeln möchte, dann müssen dementsprechend separate Anwendungen entwickelt werden. Alle Schritte, die für die Entstehung einer Anwendung durchlaufen werden, müssen für alle Versionen der Anwendung eingehal-

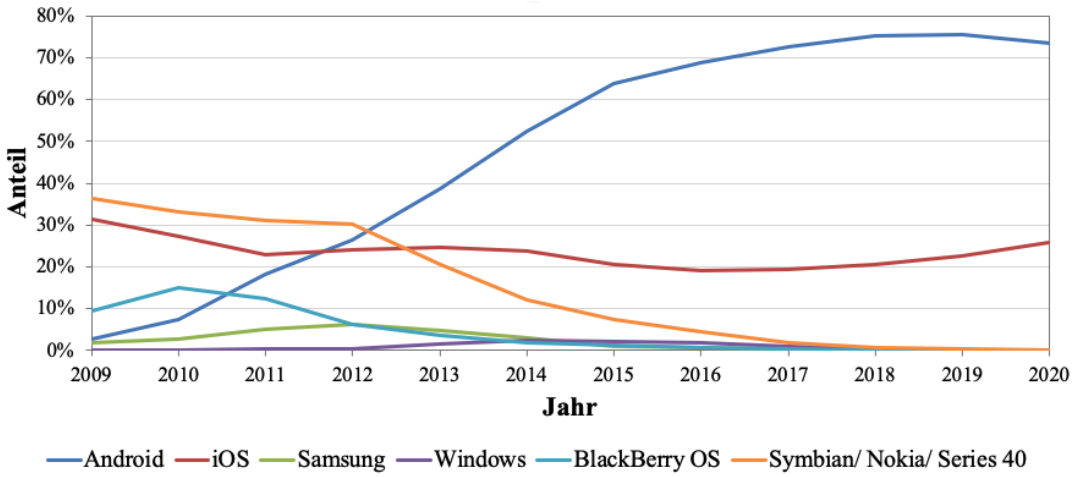

Abbildung 2: Marktanteile der mobilen Betriebssysteme an der Internetnutzung mit Mobiltelefonen weltweit zwischen 09.2009 und 03.2020 (Jahresdurchschnitte)[3]

ten werden (vgl. Abbildung 1). Dies erfordert deutlich mehr Ressourcen und bedeutet mehr Zeit- und Wartungsaufwand, was schlussendlich zu höheren Kosten führt.

2.1.1 Verteilung zwischen Android und iOS

In den letzten vier Jahren haben sich zwei große mobile Betriebssysteme durchgesetzt. Android von Google und iOS von Apple. Beide haben gemeinsam einen weltweiten Marktanteil von 99,42 %, wobei Android auf 74,6 % und iOS auf 24,82 % kommt (StatCounter, 2020). Den verschwindend geringen Teil von 0,58 % teilen sich alle anderen mobilen Betriebssysteme untereinander auf. Dazu gehören Samsung, Symbian, BlackBerry OS und Windows Mobile. Da der weltweite Markt von Android und iOS dominiert wird, werden die anderen Betriebssysteme in dieser Arbeit nicht betrachtet.

[3]Darstellung entnommen aus StatCounter (2020)

Android

Die Erfolgsgeschichte von Android began 2005 als Google Android kaufte und das Betriebssystem für jeden Entwickler frei zur Verfügung stellte (Hall and Anderson, 2009). Im November 2007 gründete Google die Open Handset Alliance, welche sich unter anderem Softwareunternehmen, Netzbetreiber und Geräte-Hersteller (darunter zum Beispiel HTC, LG Electronics, Samsung Electronics, Motorola, etc.) anschlossen (OHA, 2007). Damit konnten Anwendungen, die für Android programmiert wurden, auf unterschiedlichen Geräten von verschiedenen Herstellern ausgeführt werden. Anwendungen für Android Betriebssysteme können auf jeder Hardware genutzt werden, welche dieses Betriebssystem nutzen. Dies umfasst mittlerweile weit mehr als nur Smartphones oder Tablets. Um eine Anwendung über den Google Play Store anbieten zu können, wird ein kostenloser Google Entwickleraccount benötigt.

iOS

Im Januar 2007 stellte Apple das erste iPhone vor. Dieses fand in der großen Masse sofort Anklang (Hall and Anderson, 2009). Seither hält Apple an deren eigenem Betriebssystem fest. Um eine native Anwendungen für iOS zu entwickeln, müssen einige Voraussetzungen erfüllt werden. iOS-Anwendungen können ausschließlich auf einem Mac von Apple entwickelt werden. Damit die entwickelten Anwendungen deployed und im App Store angeboten werden können, muss eine gebührenpflichtige einjährige Mitgliedschaft im Apple Developer Programm abgeschlossen werden.

Bereits 2009 haben Hall and Anderson (2009) einen großen Vorteil von Android gegenüber iOS beschrieben. Die Autoren nannten in diesem Hinblick die einfachere Entwicklung für Android. Unter einfach verstanden die Autoren unter anderem, dass im Gegenzug zu dem proprietären, geschlossenem System von Apple (iOS), das Android Betriebssystem auf einem Open-Source

Linux-Betriebssystem basiert und somit der Quellcode für alle Entwickler offen einsehbar ist. Weiters beschrieben sie, dass für die Entwicklung von iOS Anwendungen ein MacBook angeschafft werden muss, wohingegen Android Anwendungen auf jedem Computer programmiert werden können.

2.2 Plattformübergreifende Entwicklungsansätze

Neben der nativen Entwicklung gibt es die betriebssystemübergreifende beziehungsweise plattformübergreifende Entwicklung. Hier wird eine einzige Codebasis entwickelt (siehe Abbildung 3), welche für die verschiedenen Betriebssysteme angepasst werden kann. Diese kann von jedem mobilen Gerät gelesen und ausgeführt werden (Corral et al., 2012). Auf diese Weise reduziert sich die Entwicklungszeit sowie der Entwicklungsaufwand. Der Schwerpunkt liegt bei diesem Entwicklungsansatz auf der Optimierung des Kosten-Nutzen-Verhältnisses. Diese Optimierung kommt durch die Nutzung einer Codebasis für die verschiedenen Betriebssysteme zustande (Delia et al., 2015).

Die plattformübergreifenden Entwicklungsansätze basieren größtenteils auf Webtechnologien. Zu diesen zählen unter anderem HyperText Markup Language (HTML), Cascading Style Sheets (CSS) und JavaScript (JS). Diese Sprachen sind unter Webentwicklern sehr bekannt, was bedeutet, dass von den Entwicklern keine zusätzlichen spezifischen oder plattformabhängigen Programmiersprachen erlernt werden müssen.

Die drei Grundkategorien der Entwicklungsansätze *nativ*, *hybrid* und *web* werden in der Literatur akzeptiert, wobei manchmal weitere Kategorien und Unterteilungen hinzugefügt werden. Native Anwendungen wurden bereits im vorherigen Kapitel 2.1 beschrieben. Die plattformübergreifenden Entwicklungsansätze werden in den nachfolgenden Unterkapiteln beschrieben.

[4]Darstellung entnommen aus Corral et al. (2012)

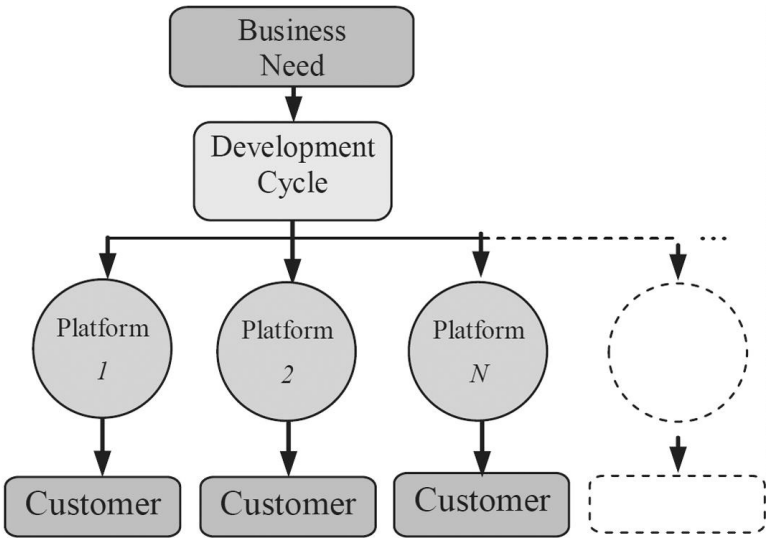

Abbildung 3: Multiplatform Entwicklungsmodell[4]

Für die Unterteilungen gibt es in der Literatur eine Vielzahl an Modellen, welche in weiterer Folge chronologisch, beginnend bei 2010 bis heute, nachstehend erläutert werden.

2010

Behrens (2010a) beschrieb in seinem Vortrag auf der MobileTechCon 2010 fünf Arten von mobilen Entwicklungsansätzen. Zu den drei zuvor genannten (nativ, hybrid und web) fügt der Behrens noch *interpretierte* und *generierte* Anwendungen hinzu. Als interpretierte Anwendungen beschreibt Behrens solche, die plattformspezifische native UI-Elemente zur Interaktion mit dem Benutzer verwenden, während die Anwendungslogik plattformunabhängig geschrieben ist. Als generierte Anwendungen beschreibt der Autor Anwendungen, die aus einer einzigen Codebasis native Anwendungen für jede Zielplattform in der jeweiligen Programmiersprache generieren (Behrens, 2010b).

2012

Ohrt and Turau (2012) listen zwei Kategorien für mobile Anwendungen in ihrer Arbeit auf. Sie unterscheiden *integrierte* und *nicht integrierte* Anwendungen. Integrierte Anwendungen werden nach der Installation in das System integriert und können auf die Systemfunktionalitäten zugreifen, wohingegen nicht integrierte Anwendungen zumeist nicht auf die Systemfunktionalitäten zugreifen können. Die nicht integrierten Anwendungen werden meist über separate Tools, wie einen Webbrowser aufgerufen.

Raj and Tolety (2012) klassifizieren die plattformübergreifenden mobilen Entwicklungsansätze nach vier Kategorien: *Web-Ansatz*, *Hybrid-Ansatz*, *interpretierter Ansatz* und *Cross-Kompilierter-Ansatz*. In ihrem Paper beschreiben die Autoren die oben genannten Ansätze mit deren Vorteilen und Herausforderungen.

Ribeiro and da Silva (2012) verwenden in ihrer Studie die folgende Kategorisierung für plattformübergreifende Entwicklungsansätze: *Laufzeit*, *Web-to-native-Wrapper*, *App Factory* und *Domain Specific Language (DSL)*. Dazu analysieren und beschreiben sie sechs plattformübergreifende Frameworks und Tools (Rhodos, PhoneGap, DragonRAD, Appcelerator Titanium, mobl und Canappi mdsl) und listen deren Vor- und Nachteile auf. Die Autoren vergleichen die Frameworks auf Grund folgender Faktoren: technologischer Ansatz, unterstützte Plattformen, Entwicklungsumgebung, resultierender Typ der Anwendung und die Unterstützung der Geräte-Application Programming Interface (API).

Smutny (2012) beschreibt in seinem Paper mobile Web-Anwendungen und native Anwendungen. Der Autor ordnet sie in die vier Kategorien ein: *native* Anwendungen, *hybride* Anwendungen, *dedizierte Web*-Anwendungen und *generische mobile* Anwendungen. Eine dedizierte Web-Anwendung ist eine mobile Webseite, die auf eine bestimmte Plattform oder einen bestimmten Formfaktor zugeschnitten ist, wohingegen generische mobile Anwendungen, mobile

Webseiten sind, die auf jedem internetfähigen Smartphone oder Tablet laufen.

2013

Perchat et al. (2013) klassifizieren plattformübergreifende Lösungen in ihrer Arbeit wie folgt: Anwendungen die auf *Cross-Compilern basieren*, Lösungen, die auf *modellgetriebenem Engineering basieren* und die *Interpreter*, welche mit einer speziellen Engine den Quellcode in Echtzeit in eine ausführbare Anwendung übersetzen. Den Quellcode-Interpreter-Ansatz unterteilen die Autoren weiters in *virtuelle Maschinen* (VMs) und *webbasierte Lösungen*.

Xanthopoulos and Xinogalos (2013) klassifizieren plattformübergreifende Entwicklungsansätze nach vier Kategorien: *Web*, *hybride*, *interpretierte* und *generierte* Anwendungen. Die Autoren geben einen Überblick und führen eine vergleichende Analyse aktueller plattformübergreifender Ansätze durch. Dabei heben sie die Vor- und Nachteile der einzelnen Entwicklungstypen hervor und kommen zu dem Schluss, dass die Implementierung von interpretierten Entwicklungsansätzen eine vielversprechende plattformübergreifende Entwicklungslösung darstellt.

2015

2015 haben El-Kassas et al. (2017) in ihrem Paper die folgende Taxonomie für Entwicklungsansätze von plattformübergreifenden mobilen Anwendungen vorgeschlagen: *kompiliert*, *komponentenbasiert*, *interpretiert*, *modellgetriebene*, *Cloud-basiert* und den *merged*-Ansatz. Darüber hinaus schlagen die Autoren noch zusätzliche Unterkategorien vor. Der kompilierte Entwicklungsansatz wird in den *cross-kompilierten* und den *trans-kompilierten* Ansatz unterteilt. Der interpretierte Ansatz in: *webbasiert*, *Virtual Machine (VM)* und *Runtime*. Der modellgetriebene Ansatz in: *Model-driven User Interface Development (MD-UID)*

und *Model-Driven Development (MDD)*. Die Autoren erläutern die Vor- und Nachteile der verschiedenen Entwicklungsansätze und deren Unterarten.

2016

Mercado et al. (2016) halten sich bei ihrer Klassifizierung an die Einteilung von Xanthopoulos und Xinogalos (Xanthopoulos and Xinogalos, 2013), also hybrid, interpretiert und generiert. Die Autoren ignorieren den Web-Ansatz, da diese Art der Anwendung (Stand 2013) nicht über die App Stores (Google Play Store und Apple's App Store) vertrieben werden. Weiters kombinieren die Autoren den interpretierten mit dem generierten Ansatz, da diese ihrer Meinung nach sehr ähnlich sind. Deren Einteilung ist daher: *hybrider Ansatz* und *interpretierter/generierter Ansatz*.

2017

Mohamed and Abdelmounaïm (2017) verwenden die Einteilung: *nativ*, *hybrid* und *Web*-Ansatz. Native Anwendungen werden als Anwendungen mit der höchsten Leistung, nativen Look & Feel sowie vollem Zugriff auf die Gerätefunktionen beschrieben. Im Gegensatz dazu steht der Web-Ansatz, bei welchem nicht die Möglichkeit besteht, auf die gerätespezifische Hardware zugreifen zu können. Der hybride Ansatz wird mit einer zusätzlichen Schicht zwischen dem Benutzer und der Anwendung dargestellt. Diese Zwischenschicht wird bei Android und iOS Webkit genannt.

2018

Nunkesser (2018) schlägt in seiner Arbeit eine neue Taxonomie vor. Nunkesser definiert zwei verschiedene Einteilungen. Eine für den täglichen Gebrauch und eine mit einer genaueren Unterteilung. Die Erstgenannte wird in sechs ver-

schiedene Kategorien eingeteilt: *endemische* Anwendungen, *Webanwendungen*, *hybride* Webanwendungen, *hybride überbrückte* Anwendungen, Anwendungen in *Systemsprache* und *fremdsprachliche* Anwendungen.
Die genauere Unterteilung beinhaltet drei Hauptkategorien (siehe Abbildung 4) und sieben Unterkategorien: Die erste Kategorie umfasst die *endemischen* Anwendungen. Diese sind Anwendungen, die mit den SDKs der Betriebssystemanbieter entwickelt werden. In anderen wissenschaftlichen Arbeiten werden diese als native Anwendungen bezeichnet. Die zweite Kategorie umfasst *pandemische* Anwendungen. Als pandemische Anwendungen bezeichnet Nunkesser Anwendungen, die auf Webtechnologien basieren und somit von den wichtigsten mobilen Betriebssystemen unterstützt werden. Diese Kategorie wird in vier weitere Unterkategorien unterteilt: *Web*-Anwendungen, *hybride Web*-Anwendungen, *hybride überbrückte* Anwendungen und Anwendungen in *Systemsprache*. Als dritte Kategorie nennt Nunkesser *endemische* Anwendungen oder *fremdsprachliche* Anwendungen. Endemische Anwendungen wurden mit plattformübergreifenden Frameworks in einer Sprache programmiert, die nicht endemisch für das mobile Betriebssystem ist. Diese Kategorie wird weiters in *interpretierte* Anwendungen, *generierte* Anwendungen und *VM* Anwendungen unterteilt.

Jia et al. (2018) verwenden in ihrem Paper die Klassifizierung *native* und *plattformübergreifend*, wobei sie plattformübergreifende Ansätze in drei Unterklassen einteilen: *verpackte Webanwendungen/hybride* Anwendungen, *proxy-native* Anwendungen und *Cross-Kompilation mit nativer API-Anbindung*. Als verpackte Webanwendungen nennen die Autoren Anwendungen, die mit Standard-Webtechnologien (HTML, CSS, JS) geschrieben wurden, welche von allen mobilen Plattformen unterstützt werden. Proxy-native Anwendungen verwenden eine JavaScript-API für die Erstellung nativer UIs, weshalb sie wie native Anwendungen aussehen und sich genauso verhalten. Cross-kompilierte Anwendungen benutzen auch API-Bindungen, um es Programmen einer Program-

[5] Darstellung entnommen aus Nunkesser (2018)

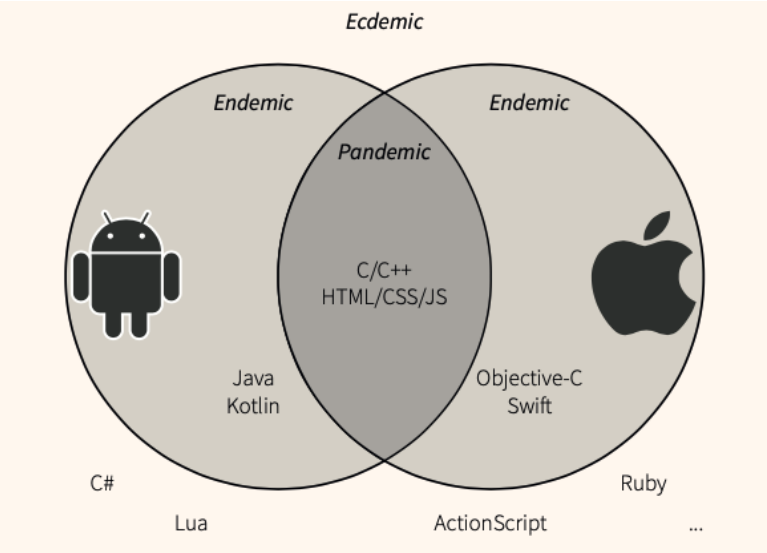

Abbildung 4: Drei Hauptkategorien nach Nunkesser[5]

miersprache zu ermöglichen, Bibliotheken anderer Sprachen zu benutzen und um Anwendungsquellcode in Binärdateien verschiedener Zielplattformen zu kompilieren (Jia et al., 2018).

Biørn-Hansen et al. (2018) teilen plattformübergreifende Entwicklungsansätze in die folgenden fünf Kategorien ein: *hybrid*, *interpretiert*, *cross-kompiliert*, *modellgetrieben* und *progressive Web Apps*. Aufgrund der Aktualität der genannten Arbeit, welche auf einer großen Anzahl an Literatur basiert, wird diese Einteilung für die vorliegende Masterarbeit verwendet. In den kommenden Unterkapiteln werden die fünf Entwicklungsansätze und die dahinter stehenden Technologien mit ihren Vor- und Nachteilen erläutert. Die Autoren erwähnen, dass es durchaus weitere Ansätze gibt, die nicht in dieser Taxonomie vorkommen. Diese seien aber aufgrund ihrer Neuartigkeit zu wenig erforscht und erprobt, weshalb sie keine Aufnahme finden.

2.2.1 Hybrider Entwicklungsansatz

Der hybride Entwicklungsansatz wird als eine Kombination zwischen den Vorteilen von bekannten Webtechnologien und nativen Funktionalitäten bezeichnet. Hybride Anwendungen basieren auf regulären Webtechnologien, wie HTML, CSS und JavaScript, welche zur Implementierung von Benutzeroberflächen und Programmlogik genutzt werden (Biørn-Hansen et al., 2018). Dieser Ansatz verwendet die Browser-Engine im Gerät und bettet den HTML-Inhalt in einen nativen Web-Container ein (Latif et al., 2016). Bei Android heißt dieser Web-Container WebView. WebView wiederum verwendet die Open-Source-Rendering-Engine Web-Kit, um Webinhalte anzuzeigen und auszuführen (Gok and Khanna, 2013). Bei iOS heißt der Web-Container UIWebView. Die WebView ist eine API, welche es der nativen Anwendung ermöglicht, Uniform Resource Locator (URL)- oder HTML-Dateien zu verarbeiten und als Ergebnis eine Webseite innerhalb der Anwendung selbst zu rendern, ohne den Standard-Internetbrowser nutzen zu müssen (Adinugroho et al., 2015). Durch die Verwendung von WebView verhält sich die native Anwendung wie ein Internet-Browser, wobei die WebView-API auch in der Lage ist, CSS und JavaScript zu verarbeiten. Dies führt zu einer besseren Anpassung der Funktionalitäten und der Schnittstelle einer Webseite (Adinugroho et al., 2015). Der WebView ist normalerweise mit der Standard-Web-Engine der Plattform gekoppelt, weshalb sich eine Aktualisierung der Web-Engine auf die WebView und somit auf die Anwendung auswirkt, welche die WebView-API anwendet (Adinugroho et al., 2015).

Bei einer hybriden Anwendung wird die Front-End- und Programmlogik der Anwendung mit Webtechnologien implementiert, welche dann eine native Anwendung ergeben. Diese umhüllt und bündelt den Code um ihn in weiterer Folge über die eingebettete WebView anzuzeigen (Biørn-Hansen et al., 2018). Diese Technik wird auch als Native-Wrapper (Willocx et al., 2015) bezeichnet, da es die Webelemente in eine veröffentlich- und einsetzbare native Anwen-

dung verpackt, welche dann aus den beiden App Stores heruntergeladen und installiert werden kann (Biørn-Hansen et al., 2018).

Die meisten hybriden Anwendungen basieren auf dem Cordova Framework, welches eine Open-Source-Bibliothek ist. Cordova übernimmt die Einrichtung von WebViews und Kommunikationsprotokollen für den Entwickler. Durch die Verwendung eines Befehlszeilen-Tools generiert Cordova eine neue native Anwendung mit einem WebView und einer Zwei-Wege-Kommunikation zwischen dem WebView und dem nativen Code (siehe Abbildung 5). Diese Art der Kommunikation - Bridging oder auch Foreign Function Interface (FFI) genannt - ermöglicht es Entwicklern, mit plattformspezifischen nativen Code aus einer nicht nativen Umgebung heraus zu kommunizieren (Biørn-Hansen et al., 2018). Cordova erleichtert somit die Kommunikation zwischen den Bridges und dem App Packaging (Biørn-Hansen and Ghinea, 2018).

Das Bridging oder FFI wird bei rechenintensiven Aufgaben eingesetzt oder um Funktionalitäten zu nutzen, welche nur in nativen Umgebungen zugänglich sind (Adinugroho et al., 2015; Rieger and Majchrzak, 2016). Die Bridge wird direkt aus dem JavaScript-Code heraus aufgerufen. Danach wird eine native Funktion ausgeführt, die auch einen Wert aus der nativen Funktion an den JavaScript-Code zurückgeben kann. Dabei kann es sich beispielsweise um eine angeforderte Global Positioning System (GPS)-Koordinate handeln. Auf diese Weise wird eine ähnliche Benutzererfahrung wie bei nativen Anwendungen gewährleistet (Biørn-Hansen et al., 2018).

2.2.2 Interpretierter Entwicklungsansatz

Für den interpretierten Entwicklungsansatz kann, wie beim hybriden Entwicklungsansatz auch, die Programmiersprache JavaScript verwendet werden. Diese ist jedoch nicht zwingend, da es auch Frameworks gibt, welche andere Spra-

[7] Darstellung entnommen aus (Biørn-Hansen et al., 2018)

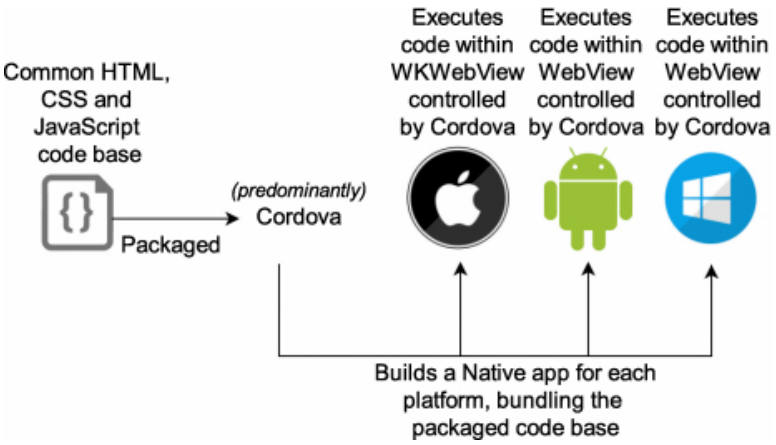

Abbildung 5: Überblick über den hybriden Entwicklungsansatz am Beispiel von Cordova[7]

chen verwenden. Der Unterschied zum hybriden Entwicklungsansatz besteht darin, dass interpretierte Anwendungen keine WebView zum Rendern der Webseite verwenden, da für die Darstellung echter nativer grafischer Benutzeroberflächen auf dem Bildschirm weder HTML noch CSS verwendet wird (El-Kassas et al., 2017). Stattdessen wird die Benutzeroberfläche vollständig programmatisch implementiert (Heitkötter et al., 2012), zur Laufzeit interpretiert und so eine native Benutzeroberfläche generiert. Diese ähnelt im Look & Feel dem typischen Erscheinungsbild der jeweiligen Plattform, da die Benutzeroberfläche aus nativen Elementen besteht (Heitkötter et al., 2012).

Der interpretierte Ansatz, verwendet für das Rendern einen JavaScript-Interpreter, welcher sich bereits auf dem mobilen Endgerät befindet (Majchrzak et al., 2017). Der plattformübergreifende Code der Anwendung, in diesem Fall JavaScript, wird zur Laufzeit interpretiert und in einer VM, welche auf den Endgeräten installiert ist, ausgeführt (El-Kassas et al., 2017). Die nativen Funktionen werden durch eine abstrakte Schicht - dem sogenannten JavaScript-Interpreter (Biørn-Hansen et al., 2018) - bereitgestellt. Dieser interpretiert den Code zur Laufzeit über die verschiedenen Plattformen hinweg, um auf die nativen APIs zuzugreifen (Latif et al., 2016). Der JavaScript-Interpreter ist plattformabhängig. Auf iOS-Geräten ist JavaScriptCore standardmäßig installiert, bei Android-Geräten

variiert der Interpreter je nach verwendetem Framework. Wie in Abbildung 6 zu sehen ist, wird dafür die V8 Engine am häufigsten verwendet (Majchrzak et al., 2017).

Zur Kommunikation zwischen dem JavaScript-Layer und dem nativen Code-Layer, welcher in weiterer Folge Zugriff auf die nativen Gerätefunktionen hat, verwendet der interpretierte Entwicklungsansatz, wie der hybride Entwicklungsansatz, ebenfalls die Technik des Bridging oder FFI (Biørn-Hansen and Ghinea, 2018). Im Gegensatz zum hybriden Entwicklungsansatz, welcher die Open-Source-Bibliothek Cordova als Bridge zur Kommunikation zwischen dem JavaScript-Code und den nativen Gerätefunktionen einsetzt, existiert eine solche Bibliothek für den interpretierten Entwicklungsansatz nicht (Biørn-Hansen and Ghinea, 2018; Biørn-Hansen et al., 2018). Da die Framework-APIs so unterschiedlich sind, gibt es keine Framework-übergreifenden Plugins oder Module (Biørn-Hansen et al., 2018), was ein Zusammenspiel von verschiedenen Frameworks schwierig macht.

Zu den Vorteilen des interpretierten Entwicklungsansatzes gehören die bereits weiter oben genannten Features wie beispielsweise die Möglichkeit der Entwicklung mit JavaScript, einer weit verbreiteten Programmiersprache, die vielen Entwicklerinnen und Entwicklern gut bekannt ist, sowie die Ermöglichung von nativen Benutzeroberflächen. Als Nachteil muss erwähnt werden, dass es zu einer Verschlechterung der Performanz kommen kann, da der Code zur Laufzeit der Anwendung interpretiert werden muss (Ribeiro and da Silva, 2012). Eine weitere Kehrseite ist die Abhängigkeit von der Entwicklungsumgebung. Werden beispielsweise neue plattformspezifische Funktionen veröffentlicht, können diese erst dann in den Anwendungen genutzt werden, wenn sie von der Entwicklungsumgebung integriert wurden (Latif et al., 2016).

[8] Darstellung entnommen aus (Biørn-Hansen et al., 2018)

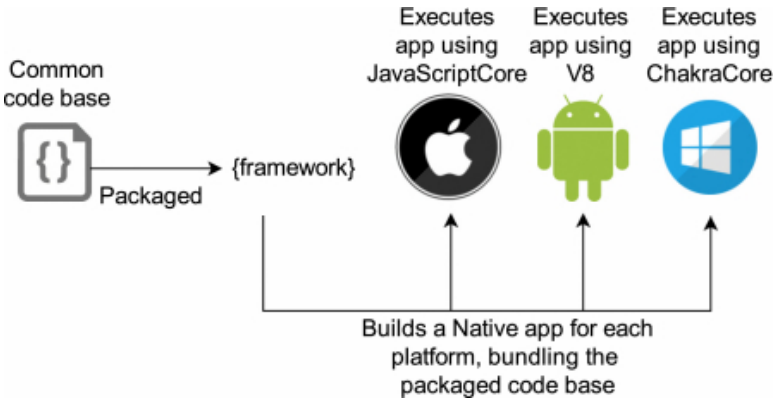

Abbildung 6: Überblick über den interpretierten Entwicklungsansatz[8]

2.2.3 Cross-kompilierter Entwicklungsansatz

Der cross-kompilierte Entwicklungsansatz, auch generierter Ansatz (Xanthopoulos and Xinogalos, 2013; Latif et al., 2016) genannt, unterscheidet sich nicht so stark von dem zuvor beschriebenen interpretierten Entwicklungsansatz. Der cross-kompilierte Ansatz ist ebenfalls nicht auf die WebView oder on-Device Interpreter zur Darstellung der Benutzeroberfläche oder zur Kommunikation zwischen plattformunabhängigen Code und den nativen Gerätefunktionen angewiesen. Der Grund dafür ist, dass die Benutzeroberfläche nicht während der Laufzeit gerendert werden muss. Stattdessen wird eine Anwendung in einer plattformübergreifenden Programmiersprache implementiert und dann in nativen Bytecode kompiliert, welcher auf den Zielplattformen ausführbar ist. Die kompilierte Anwendung verwendet die native Programmiersprache und kann somit als native mobile Anwendung ausgeführt und betrachtet werden (Ciman and Gaggi, 2017). Dadurch enthält die Anwendung eine nativ generierte Benutzeroberfläche, welche auch ein natives Gefühl bietet (Willocx et al., 2015).

Bei diesem Entwicklungsansatz ist eine zusätzliche Abstraktionsschicht, das Bridging beziehungsweise FFI (Latif et al., 2016), wie beim interpretierten und hybriden Ansatz, nicht notwendig. Der Zugriff auf die gerätespezifischen Funktionen wird vom Framework SDK verwaltet, welches mit dem SDK der

zugrundeliegenden Plattform kommuniziert. Jia et al. (2018) bezeichnen diesen Entwicklungsansatz auch als *Cross-Kompilierung mit nativer API-Bindung*. Diese API-Bindungen erlauben es Programmen, die Bibliotheken einer anderen Sprache zu verwenden und den Quellcode der Anwendung in Binärcode der Zielplattform zu cross-kompilieren.

Als Vorteil des Ansatzes wird die gute Performanz beschrieben. Dazu zählt, dass die Anwendungen in der Lage sind, native Leistung zu erreichen und dass alle Funktionen der nativen Anwendungen zusammen mit ihren nativen Schnittstellen geliefert werden (Latif et al., 2016). Auch Jia et al. (2018) heben in ihrer Arbeit hervor, dass mit der nativen API-Bindung und Kompilierung des Quellcodes zu Binärdateien der Zielplattform, die resultierenden Anwendungen, genau wie native Anwendungen aussehen und sich ebenso verhalten.

In der Literatur werden aber auch einige negative Punkte des cross-kompilierten Entwicklungsansatzes aufgezeigt. Es können beispielsweise einige wenige Funktionen nicht genutzt werden, dazu zählen die Daten von Geolokalisierungsdiensten sowie der Kamerazugriff. Dies liegt daran, dass diese Funktionen plattformspezifisch sind und der Zugang von Plattform zu Plattform unterschiedlich ist (Latif et al., 2016). Es wird auch die schlechte Codequalität kritisiert, welche auf die automatische Generierung des Quellcodes von einer Programmiersprache in die Programmiersprache der Zielplattform zurückzuführen ist (Xanthopoulos and Xinogalos, 2013; Ciman and Gaggi, 2017).

In der Literatur werden einige Technologien beziehungsweise Frameworks verschieden klassifiziert. So wird die Technologie Xamarin unterschiedlich eingeordnet. El-Kassas et al. (2017) ordnen Xamarin dem interpretierten Entwicklungsansatz zu, während Willocx et al. (2015) und Biørn-Hansen et al. (2018) diese dem cross-kompilierten Entwicklungsansatz zuordnen. Die Begründung liegt darin, dass Xamarin nicht auf Interpreter angewiesen ist, da die gemeinsame Sprache in nativen Binärcode kompiliert wird (Biørn-Hansen et al., 2018).

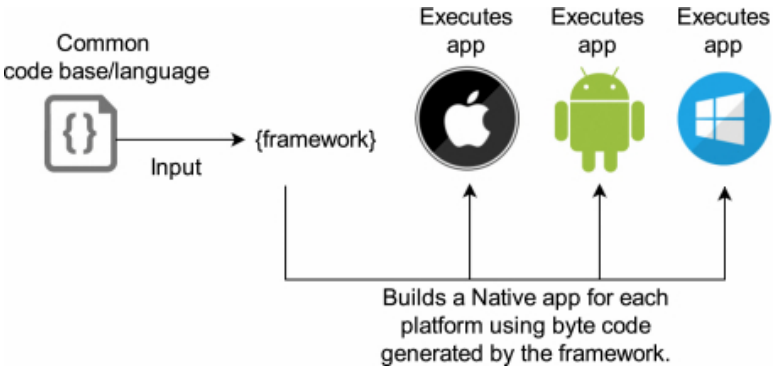

Abbildung 7: Überblick über den cross-kompilierten Entwicklungsansatz[9]

Die Architektur von cross-kompilierten Anwendungen kann in Abbildung 7 betrachtet werden.

2.2.4 Modell-getriebener Ansatz

Wie der Name dieses Entwicklungsansatzes bereits verrät, lehnt sich dieser Ansatz an das Softwareentwicklungsparadigma des Model-Driven Development (MDD) an, welches auch als Model-Driven Software Development (MDSD) bezeichnet wird (Heitkötter et al., 2013). Ribeiro and da Silva (2012) verwenden in ihrer Arbeit den Überbegriff Model-Driven Engineering (MDE), beschreiben aber auch den MDD-Ansatz .

Das Ziel der modell-getriebenen Softwareentwicklung ist es, ein Problem in einem Modell zu beschreiben und aus dieser Darstellung die Software zu generieren (Heitkötter et al., 2013). Dazu werden keine Informationen über plattformspezifische Programmiersprachen benötigt (Heitkötter and Majchrzak, 2013). Zur Darstellung der Funktionalität und des Verhaltens der Anwendung wird eine Modellierungssprache verwendet, welche als Domain Specific Language (DSL) bezeichnet wird (Xanthopoulos and Xinogalos, 2013). Das bedeutet, dass die Anwendung zuerst mit DSL beschrieben wird und in weiterer Folge wird das Modell automatisch in den Quellcode der jeweiligen Plattform

[9]Darstellung entnommen aus (Biørn-Hansen et al., 2018)

überführt (Heitkötter and Majchrzak, 2013; Usman et al., 2017) und eine komplett native Anwendung erstellt. Dies ist möglich, da jede Anwendung direkt das native Source Development Kit der jeweiligen mobilen Plattform und die jeweiligen nativen Benutzerschnittstellenenlemente verwendet. Deshalb sind die generierten Anwendungen stark in die jeweilige Plattform integriert und weisen ein natives Erscheinungsbild auf (Heitkötter and Majchrzak, 2013). Die so generierten Anwendungen sind nativ und weisen keine Probleme auf, wie es zum Beispiel die webbasierten Ansätze im Hinblick auf das native Look & Feel aufweisen. (Heitkötter et al., 2013). Latif et al. (2016) weisen auf die Probleme hin, die bei diesem Entwicklungsansatz entstehen. Zum einen ist die Anwendungsdomäne auf die Modellsprache beschränkt, somit können nur Anwendungen modelliert werden, welche in die vom Modell unterstützten Kategorien fallen. Zum anderen ist der generierte Quellcode unvollständig und sollte manuell unter der Verwendung der nativen Programmiersprachen und SDK-Werkzeuge vervollständigt werden. Die Autoren weisen darauf hin, dass die manuelle Integration von geschriebenem Code eine Herausforderung darstellt.

Es gibt verschiedene Frameworks, die hier Abhilfe schaffen können. Die Frameworks des modell-getriebenen Entwicklungsansatzes unterscheiden sich im Hinblick auf die integrierte Funktionalität, wie Heitkötter and Majchrzak (2013) in ihrer Arbeit darlegen. Sie stellen in ihrer Studie das, sich noch in Entwicklung befindliche, Framework MD^2 vor, welches keinerlei Kenntnisse der nativen Programmiersprachen verlangt. Dieses basiert auf einer textbasierten DSL und ermöglicht so die Generierung lauffähiger nativer Anwendungen für Android und iOS (Latif et al., 2016).

Der Einsatz von DSLs soll verhindern, dass Entwickler plattformspezifische Programmiersprachen, wie etwa Objective-C oder Java, zur Programmierung erlernen müssen. Darüber hinaus werden keine vertieften Kenntnisse in der Entwicklung für iOS oder Android benötigt. Somit können auch Nicht-Entwickler

eine Anwendung für Smartphones erstellen. Anstelle dieser spezifischer Sprachen bedarf es aber der Kenntnis der jeweiligen Framework-spezifischen DSLs (Heitkötter and Majchrzak, 2013). Dadurch, dass die DSLs Framework-spezifisch sind, gibt es eine große Menge an verschiedenen DSLs, da jedes Framework seine eigene DSL entwickelt. Umuhoza et al. (2015) stellen in ihrer Arbeit einige DSLs vor und Le Goaer and Waltham (2013) stellen in ihrer Arbeit mit dem Titel *„Yet Another DSL for Cross-Platforms Mobile Development“* eine weitere DSL vor. Der Titel ist eine Anspielung darauf, dass bereits sehr viele DSLs existieren.

Bei der modell-getriebenen Entwicklung gibt es verschiedene Abstraktionslevel der Modelle und unterschiedliche Ansätze. Umuhoza et al. (2015) sowie Latif et al. (2016) verwenden Model-Driven Architekture (MDA) als Bezugsrahmen, um die einzelnen Schritte zu veranschaulichen. Es geht darum, dass die Entwickler eine Anwendung auf hoher Abstraktionsebene beschreiben können, ohne sich mit den technischen Fragen auf der untersten Ebene befassen zu müssen. Zu den Fragen auf unterster Ebene zählen zum Beispiel Fragen zur Speicherung von Daten oder Systemberechtigungen (Latif et al., 2016).

Die modell-getriebene Architektur wurde von der Object Management Group (OMG) als Standard definiert (Latif et al., 2016). Dabei beginnt man mit einem Platform Independent Model (PIM), welcher durch eine Reihe von Transformationen, die zur Erstellung einer oder mehrerer Platform Specific Models (PSMs) führen, welche an die Zielplattform angepasst sind. Diese Transformation von PIM zu PSM wird als Model-to-Model (M2M) bezeichnet (Vranić and Staraček, 2014). Auf der Ebene der plattformunabhängigen Modelle wird die Anwendung ohne plattformspezifische Eigenheiten modelliert. Das plattformspezifische Modell hat bereits die plattformspezifischen Details (Latif et al., 2016).

El-Kassas et al. (2017) unterteilen den Modellierungs-Ansatz nochmals und unterscheiden zwischen dem Model-Based User Interface Development (MB-UID) und dem bereits beschriebenen MDD. Beim MB-UID wird zwischen der Logik für die Benutzerschnittstelle und der Anwendungslogik unterschieden.

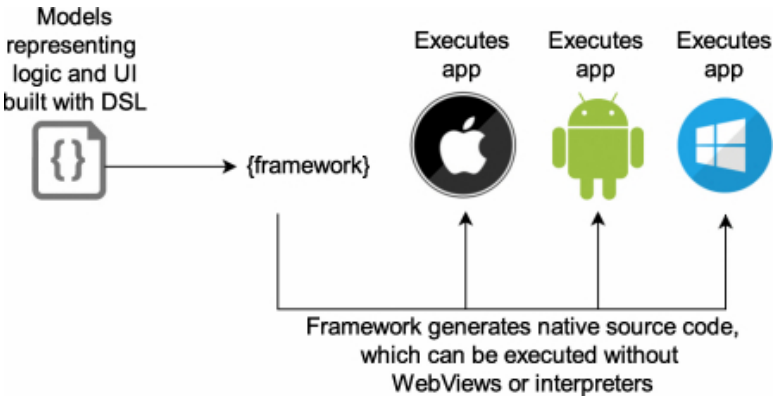

Abbildung 8: Überblick über den modell-getriebenen Entwicklungsansatz[10]

Wohingegen das Hauptkonzept bei MDD darin besteht, aus einem plattformunabhängigen Modell plattformspezifische Versionen der App zu generieren (El-Kassas et al., 2017).

Auf Grund des noch experimentellen Status dieses Entwicklungsansatzes wird dieser zur Zeit noch eher im wissenschaftlichen Umfeld eingesetzt. Aus einer empirischen Studie zur Verwendung von Software-Entwurfsmodellen geht hervor, dass nur ein sehr kleiner Burchteil der Entwickler Modelle verwenden. Gorschek et al. (2014) sprachen 2014 sogar von einem rückläufigen Trend.

2.2.5 Progressive Web Apps

Progressive Web Apps (PWAs) sind ein neuer Ansatz zur Erstellung plattformübergreifender Anwendungen. Der Begriff wurde erst 2015 durch einen Blogbeitrag von Russell (2015) geprägt. PWAs basieren auf Standard-Webtechnologien (HTML, CSS, JavaScript) und sind somit grundsätzlich Webseiten beziehungsweise Webanwendungen mit erweiterten Fähigkeiten. Sie werden auf einem Webserver gehostet und können mittels URL über einen Browser aufgerufen werden (Biørn-Hansen et al., 2018). Das Ziel dieses Ansatzes ist es, Webanwendungen wie eine native oder plattformübergreifende Anwendung aussehen zu

[10]Darstellung entnommen aus (Biørn-Hansen et al., 2018)

lassen. Zu den erweiterten Fähigkeiten zählen zum Beispiel die Installation und die Offline-Verfügbarkeit der Webseite. Durch die Installation auf dem Gerät werden alle erforderlichen Dateien heruntergeladen, dazu zählen alle erforderlichen JavaScript-, HTML- und CSS-Dateien, sowie Bilder und Schriftarten, welche die Offline-Nutzung der Webseite ermöglichen (Biørn-Hansen et al., 2018). Als zusätzliche Funktionalitäten können Push-Notifications und die Synchronisierung im Hintergund genannt werden (Majchrzak et al., 2017). Die Anwendung wird zunächst als Standard-Webanwendung über einen Browser aufgerufen und nach einigen Zugriffen kann die Benutzerin oder der Benutzer entscheiden ob er oder sie die PWA installieren möchte. Ab dem Zeitpunkt der Installation können die zusätzlichen Funktionalitäten in Anspruch genommen werden (Malavolta et al., 2017). PWAs arbeiten mit responsiven Designtechniken sowohl auf mobilen, als auch auf Desktop-Geräten (LePage, 2017).

Der Unterschied zwischen Web-Apps und PWAs besteht darin, dass PWAs sogenannte Service Workers implementieren (Gaunt, 2019). Der Service Worker wird beim ersten Seitenaufruf des Benutzers registriert. Er besteht aus einer JavaScript-Datei, welche Lifecycle-Hooks für die Geschäftslogik und Cache-Kontrolle enthält (Biørn-Hansen et al., 2017). Die Service Worker sind bereits seit 2009 von der W3C standardisiert. Sie sind eine Reihe von APIs, die es unter anderem ermöglichen Assets und Daten programmgesteuert zwischenzuspeichern und vorzuladen, Push-Benachrichtigungen zu verwalten und einiges mehr (Malavolta et al., 2017).

Im Gegensatz zu normalen Webseiten, wo der Benutzer nach dem eingeben einer URL darauf warten muss, dass der gesamte Inhalt der Webseite heruntergeladen wird, ist dies bei PWAs nur beim ersten Besuch der Fall. Nach der Installation werden alle erforderlichen statischen Daten für die Webseite auf dem Gerät des Benutzers gespeichert und können offline verwendet werden. Dynamische Daten können für die Offline-Nutzung zwischengespeichert und wenn notwendig, zum Beispiel wenn neue Daten verfügbar sind und das Ge-

rät über eine stabile Verbindung verfügt, erneut abgerufen werden (Richard and LePage, 2020). Weiters können Service Worker dazu genutzt werden, andere Inhalte, wie Bilder, Video- oder Audiodateien für die Offline-Nutzung zwischenzuspeichern sowie für die Implementierung langlebiger Sessions wie beispielsweise um sicherzustellen, dass die Benutzerinnen und Benutzer authentifiziert bleiben (Richard and LePage, 2020).

PWAs verfügen über eine App-Shell-Architektur. Die *Shell* der Anwendung besteht aus einem Minimum an HTML, CSS und JavaScript, welche für die Benutzerschnittstelle erforderlich ist. Wenn die App-Shell heruntergeladen wurde, wird sie beim Aufruf sofort geladen und bietet ein Gefühl einer nativen Anwendung. Die App-Shell hält ihre Benutzeroberfläche lokal und ladet die Inhalte dynamisch über eine API herunter (Osmani, 2019a).

Weiters benötigt eine PWA eine Manifest-Datei, welche eine JSON-Datei ist. Im Manifest werden unter anderem der Bildpfad des Logos und der Name der Anwendung angegeben. Das Manifest kann dazu verwendet werden, das Verhalten und den Stil von PWA-Anwendungen zu ändern (Osmani, 2019b).

Als letzte Voraussetzung für PWAs ist es erforderlich, dass aus Sicherheitsgründen HyperText Transfer Protocol Secure (HTTPS) auf dem Server eingerichtet ist. Somit kann sich ein Service Worker im Browser registrieren und entsprechend auf Ereignisse reagieren (Gaunt, 2019).

PWAs können entweder mit gängigen Web-Frameworks, wie Ionic, Angular, Vue, Polymer, React, oder mit frameworklosen Ansätzen wie StencilJS oder Svelte entwickelt werden.

Abbildung 9 zeigt die Architektur von Progressive Web Apps.

Apple unterstützt PWAs aktuell noch nicht zu 100 %. Denn der Apple interne Safari-Browser löscht in einem siebentägigen Rhythmus (Fristbeginn ist die

[11] Darstellung entnommen aus (Biørn-Hansen et al., 2018)

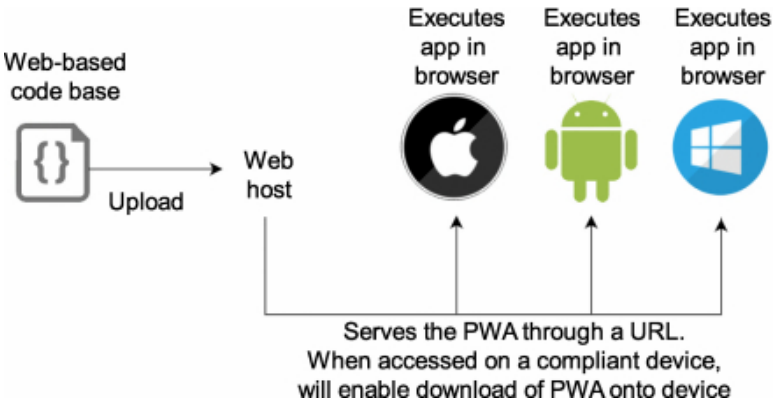

Abbildung 9: Progressive Web App Architektur[11]

letzte Nutzung der Anwendung) die gespeicherten Daten (Wilander, 2020), welche die PWAs benötigen, um sinnvoll funktionieren zu können. Aus diesem Grund werden PWAs in dieser Masterarbeit nicht betrachtet.

2.3 Studienlage zu Messungen der Ressourcennutzung

Es gibt viele Studien, die sich mit der Bewertung und dem Vergleich plattformübergreifender Entwicklungsansätze auf der Grundlage qualitativer Eigenschaften befassen. Amatya and Kurti (2013) geben einen globalen Überblick über den Stand der Technik für die Entwicklung mobiler Anwendungen und gehen auf die wichtigsten Herausforderungen und Chancen für die Entwickler ein. Die Autoren führten eine Umfrage durch, welche zeigte, dass der webbasierte Ansatz und der hybride Ansatz am besten für die plattformübergreifende Entwicklung geeignet sind.

Dalmasso et al. (2013) implementieren und testen vier Anwendungen, welche mit den plattformübergreifenden Ansätzen PhoneGap und Titanium für das Betriebssystem Android entwickelt wurden. Die Autoren haben die Leistung der Android-Testanwendung in Bezug auf CPU- und Speichernutzung sowie

den Stromverbrauch gemessen. Die Studie kommt zum Ergebnis, dass PhoneGap weniger Ressourcen verbraucht als Titanium, da es keine dedizierten Benutzerschnittstellen-Komponenten enthält.

Willocx et al. (2015) entwickeln zwei nicht-triviale Anwendungen mit mehreren Bildschirmen mit zwei plattformübergreifenden Entwicklungsansätzen (PhoneGap und Xamarin) und führen eine quantitative Leistungsanalyse durch. Zu den gemessenen Leistungsparametern zählen die Antwortzeiten für vier verschiedene Aktionen, die Startzeit der Anwendung, der Verbrauch des RAM-Speichers, die CPU-Nutzung sowie der Batterieverbrauch. Die Studie wird sowohl auf Low-End- als auch auf High-End-Geräten durchgeführt und mit nativen Android- und iOS-Implementierungen der gleichen Anwendung verglichen. Die Autoren kommen zum Ergebnis, dass plattformübergreifende Entwicklungsansätze einen Leistungsnachteil mit sich bringen.

Ein Jahr später wiederholen Willocx et al. (2016) ihre Messung, diesmal an einer Anwendung, welche mit zehn verschiedenen plattformübergreifenden Entwicklungsansätzen entwickelt wurden. Diese Anwendung wurde ebenfalls nativ für Android, iOS und das Windows Phone implementiert. Die Leistungsanalyse wurde ebenfalls wie im Jahr zuvor auf einem High-End- und Low-End Android- und iOS-Gerät sowie auf einem Windows Phone durchgeführt. Die Autoren kommen zu dem Ergebnis, dass plattformübergreifende Werkzeuge der gleichen Kategorie ein ähnliches Verhalten aufzeigen. Als CPU-intensivste plattformübergreifende Technologie werden JavaScript-Frameworks genannt.

Ajayi et al. (2018) führen eine Analyse der Leistung von unterschiedlichen Berechnungen durch, die sowohl nativ mit Android als auch mit einem hybriden Entwicklungsansatz implementiert wurden. Die Autoren messen den Leistungsunterschied bei der CPU- und Speichernutzung während rechenintensiver Berechnungen wie Rechnen mit Armstrong-Zahlen, Fibonacci-Zahlenberechnung sowie dem Quicksort-Algorithmus. Die Ergebnisse zeigen teils sehr große Unterschiede hinsichtlich der CPU- und Speichernutzung zwischen den beiden

Entwicklungsansätzen, wobei die native Implementierung deutlich besser abschneidet als die hybride Implementierung.

Huber and Demetz (2019) analysierten in ihrer Studie den Ressourcenverbrauch (CPU- und Speicherverbrauch) einer Kontakt-Anwendung im Hinblick auf eine spezifische UI-Interaktion: das Scrollen durch eine virtuelle Liste. Diese Anwendung wurde mit den beiden plattformunabhängigen Entwicklungsansätzen Ionic/Cordova sowie React Native und als eine native Android-Anwendung implementiert. Getestet wurden alle drei Anwendungen auf einem einzigen Android-Gerät, dem LG Nexus 5. Die Autoren kamen zu dem Ergebnis, dass die beiden plattformübergreifenden Entwicklungsansätze Ionic/Cordova und React Native doppelt so viel CPU verbrauchten als die nativ entwickelte Anwendung. Beim Speicherverbrauch verhielt es sich ähnlich. Die mit Ionic/Cordova implementierte Anwendung verbrauchte 28 mal mehr Speicher und die mit React Native implementierte Anwendung 14 mal mehr als die nativ implementierte Anwendung.

In einer nachfolgenden Studie erweiterten Huber et al. (2020) die Ressourcenmessung auf drei unterschiedliche UI-Interaktionen: Öffnen und Schließen des Navigation Drawers, Bildschirmübergang zwischen zwei Screens sowie das virtuelle Scrollen durch eine Liste. Auch diesmal wurde die Anwendung mit Ionic/Capacitor und React Native sowie nativ für Android implementiert. Folgende vier Leistungsmetriken wurden gemessen: CPU-Nutzung, Hauptspeichernutzung, Graphics Processing Unit (GPU)-Speichernutzung sowie das Frame-Rendering. Die Tests wurden vollautomatisch auf drei unterschiedlichen Android-Geräten durchgeführt. Die Autoren kommen zum Ergebnis, dass die beiden plattformübergreifenden Entwicklungsansätze sowohl die CPU als auch den Hauptspeicher stärker belasten als der native Ansatz. Ionic/Capacitor hatte auch eine höhere GPU-Belastung. Die Flüssigkeit der Benutzeroberfläche (Rate der janky frames) ist bei der Anwendung, die mit React Native implementiert wurde, vergleichbar mit der nativen Android-Anwendung, jedoch bei

einer höheren Belastung der CPU und des Hauptspeichers.

Diese Masterarbeit knüpft an die beiden letztgenannten Studien (Huber and Demetz, 2019; Huber et al., 2020) an. In beiden Studien wurde der Ressourcenverbrauch auf Android-Geräten getestet. Die vorliegende Arbeit erweitert die Messung um ein Testgerät der Firma Apple, also die iOS-Plattform.

2.4 Interaktion mit mobilen Anwendungen

Mobile Anwendungen für das Smartphone werden durch spezifische Gesten auf einem Multitouch-Display bedient. So wird es dem Benutzer ermöglicht, virtuelle Elemente direkt auf dem Display seines Smartphones zu berühren und zu manipulieren, ohne die Notwendigkeit externer Eingabegeräte wie Maus, Tastatur oder Ähnlichem. Dies führt zu einer natürlicheren und intuitiveren Art der Interaktion (Hesenius et al., 2014). Laut dem Material Design der Firma Google gibt es drei Arten von Gesten (Google, 2020a): Gesten zur Navigation, Aktionsgesten und sogenannte Verwandlungsgesten. Mit Navigationsgesten kann der Benutzer durch eine Anwendung navigieren. Dazu gehören unter anderem Navigationskomponenten wie der Navigation Drawer als auch Schaltflächen, wie Buttons, welche auf einen Klick reagieren. Als Aktionsgesten werden Gesten bezeichnet, die bestimmte Aktionen auslösen. Ein langer Klick auf ein Listenelement, welches dadurch ausgewählt wird, wäre ein Beispiel für eine Aktionsgeste. Als Verwandlungsgesten werden Gesten bezeichnet, mit denen die Größe, Position oder Rotation eines Elements verändert werden kann. Ein Beispiel für diese Art von Gesten ist das Hinein- und Hinauszoomen. Diese Geste entsteht durch ein auf- oder zuziehen von zwei Fingern, welche auf ein bestimmtes Element auf dem Bildschirm gelegt und zusammen oder auseinander gezogen werden. Dieses vergrößert oder verkleinert das ausgewählte Element.

Die vorliegende Masterarbeit geht auf folgende drei Navigationsgesten ein: Touch, Swipe und Scroll. Eine nähere Beschreibung dieser Gesten befindet sich in Kapitel 3.3.

3. Studiendesign

Der methodische Teil der vorliegenden Masterarbeit befasst sich mit der Durchführung des Experiments. Zu Beginn wird die Auswahl der Entwicklungsansätze argumentiert. Danach werden die drei getesteten Interaktionsszenarien vorgestellt. Anschließend werden die Implementierungen der fünf Anwendungen detailliert dargestellt. Im Anschluss werden die Abläufe der automatisierten Testfälle sowohl auf Android als auch auf iOS beschrieben. Abgeschlossen wird das Kapitel mit den Bewertungskriterien, welche für die Auswertung ausgewählt wurden.

3.1 Auswahl der Entwicklungsansätze

Für den Performanzvergleich der unterschiedlichen plattformübergreifenden Entwicklungsansätze wurden drei Anwendungen mit jeweils einem unterschiedlichen Entwicklungsansatz programmiert. Die Frameworks wurden nach Aktualität und Popularität gemäß den Trends auf Stack Overflow[1], Github[2] sowie Appfigures[3] ausgewählt.

[1] Stack Overflow Trends: https://insights.stackoverflow.com/trends?tags=
[2] GitHub: https://github.com/
[3] Appfigures: https://appfigures.com/top-sdks/development/apps

3.1.1 Stack Overflow Trends

Stack Overflow ist eine Internetplattform, die es angemeldeten Entwicklerinnen und Entwicklern ermöglicht Fragen zum Thema Softwareentwicklung zu stellen, die von anderen Nutzerinnen und Nutzern beantwortet werden. Es bietet auch ein Tool an, welches „Stack Overflow Trends“ heißt. Dieses Tool analysiert die gestellten Fragen pro Monat und macht die Technologien und Programmiersprachen über Tags vergleichbar. Nicht alle Technologien und Programmiersprachen schaffen es, als Tags abgebildet zu werden. So können ältere und nicht ganz so verbreitete Technologien nicht gefunden werden. Von den 44 verschiedenen Frameworks, welche Biørn-Hansen et al. (2018) in den drei Entwicklungsansätzen Hybrid, Interpreted und Cross-compiled auflisten, konnten lediglich acht Tags gefunden und verglichen werden, siehe Abbildung 10. Dazu muss gesagt werden, dass alle drei hybriden Frameworks auf der Open-Source-Bibliothek Cordova aufbauen. Die Grafik lässt erkennen, dass aktuell die meisten Fragen auf Stack Overflow zum cross-kompilierten Entwicklungsansatz Flutter (über 2 % aller gestellten Fragen) und zum interpretierten Entwicklungsansatz React Native (etwa 1,4 % aller Fragen) gestellt werden. Bei Flutter ist ein deutlicher Anstieg seit der Veröffentlichung 2018 bis heute erkennbar. Dies deutet darauf hin, dass die Technologie sehr gut von Entwicklern aufgenommen wurde. React Native wurde 2015 veröffentlicht und auch hier lässt sich bis heute ein stetiger Aufwärtstrend feststellen. Die dritte Position teilen sich das hybride Ionic Framework und der cross-kompilierte Entwicklungsansatz Xamarin. Bei beiden Frameworks ist die Entwicklungskurve relativ ähnlich mit einem Peak zwischen 2016 und 2018. Seit 2018 geht es aber langsam wieder bergab. Aktuell pendeln sich beide bei etwa 0,2 % aller Fragen ein.

[2]Grafik entnommen aus https://insights.stackoverflow.com/trends?tags=react-native%2Cnativescript%2Cflutter%2Ccordova%2Cxamarin%2Cionic-framework%2Csencha-touch%2Ccodenameone

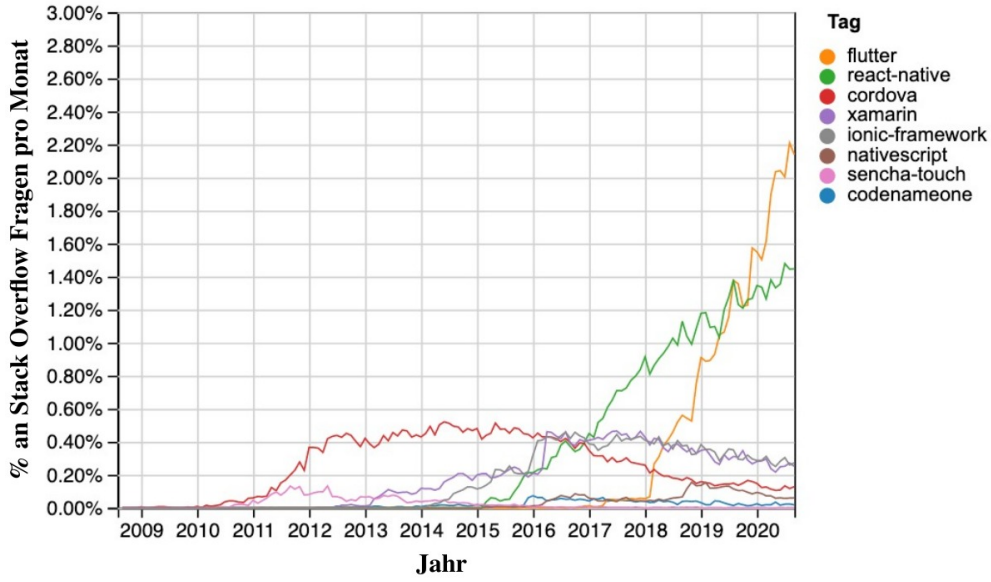

Abbildung 10: Trend-Vergleich aller plattformübergreifender Framework-Tags auf Stack Overflow[2]

3.1.2 Appfigures

Es gibt auch Unternehmen, die App Stores crawlen und aufzeigen, mit welchen SDKs die Anwendungen entwickelt werden. Eines dieser Unternehmen ist Appfigures, Inc. Auf deren Webseite veröffentlicht die Firma aktuelle Rankings. Um die folgenden Prozentangaben zu verstehen, muss zuerst ein Überblick über die verfügbaren Anwendungen in den jeweiligen App Stores aufgezeigt werden. Im ersten Quartal 2020 befanden sich im Google Play Store rund 2 560 000 Anwendungen und im Apple App Store waren es rund 1 847 000 Anwendungen (Appfigures and VentureBeat, 2020b). Aus dem Ranking in der Abbildung 11 ist ersichtlich, dass die meisten Anwendungen in den beiden führenden App Stores mit den nativen Entwicklungsansätzen entwickelt wurden. Bei iOS ist dies Swift mit 46 % und bei Android Kotlin mit 36 % aller Anwendungen. 46 % entsprechen in etwa 850 000 nativen iOS Anwendungen und 36 % entsprechen in etwa 922 000 nativen Android Anwendungen.

An zweiter Stelle kommt in beiden App Stores die Technologie Cordova. An-

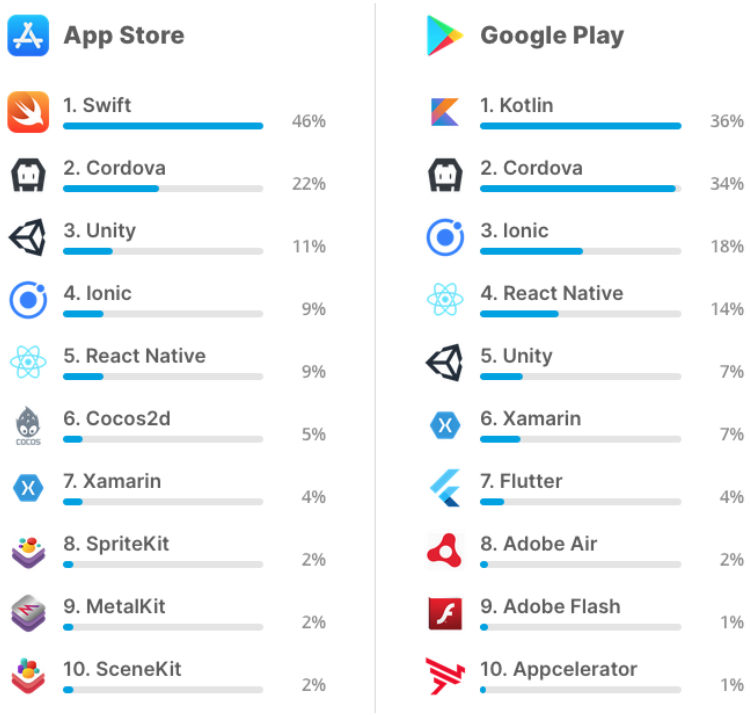

Abbildung 11: Top 10 der verwendeten SDKs in beiden App Stores[3]

wendungen, die mit dem Ionic Framework programmiert wurden, kommen im App Store mit 9 % auf den vierten Platz und im Google Play Store mit 18 % auf den dritten Platz. React Native kommt im Google Play Store mit 14 % an vierter Stelle und im App Store mit 9 % an fünfter Stelle. An dritter beziehungsweise vierter Stelle befindet sich Unity. Unity ist eine Laufzeit- und Entwicklungsumgebung für die Programmierung von Spielen. Diese Arbeit befasst sich jedoch mit der Analyse von Benutzerinteraktionen, weshalb Unity keine Beachtung findet. Weiters ist Xamarin im Google Play Store mit 7 % an sechster Stelle und im App Store mit 4 % an siebter Stelle vertreten. Flutter hingegen kommt im Google Play Store mit rund 4 % an siebter Stelle aller Anwendungen. Im App Store aber mit nur rund einem Prozent erst an vierzehnter Stelle aller Anwendungen. Alle anderen Frameworks sind mit unter 2 % vertreten.

[3]Grafik entnommen aus https://appfigures.com/top-sdks/development/apps

3.1.3 Anzahl der Repositories auf GitHub

Eine andere Möglichkeit zu sehen, wie beliebt eine Technologie ist, ist diese auf GitHub zu suchen. GitHub bietet ein Hosting für Softwareprojekte sowie Versionskontrolle mit Git an. Anfang November 2018 befanden sich bereits über 100 Millionen Repositories auf GitHub (Warner, 2018). Um zu sehen, wie viele Repositories eine bestimmte Programmiersprache oder Technologie benutzen, muss der Suchbegriff in der Suchleiste eingetragen werden. Gibt man beispielsweise „React Native“ ein, so kommt als Resultat, dass sich aktuell 184 408 Repositories auf GitHub befinden, welche das React Native Framework verwenden. Flutter wird in 156 365 Repositories verwendet, Ionic kommt auf 98 426 Repositories, Xamarin wird in 39 100 Repositories verwendet und PhoneGap in 25 352. Alle anderen gefundenen Frameworks haben maximal vierstellige Nummern. Zu den drei mit Abstand beliebtesten Frameworks auf GibHub zählen aktuell React Native, Flutter und Ionic.

3.2 Auswahl der Frameworks

Auf Grund der vorhergehenden Recherche zur Beliebtheit und Anzahl an Verwendungen wurden für die Studie folgende drei Frameworks ausgewählt: Ionic, Flutter sowie React Native. Tabelle 2 gibt einen Überblick über alle Frameworks inklusive der beiden nativen Entwicklungsansätzen und Technologien.

Für den **hybriden Entwicklungsansatz** wurde das **Ionic Framework** ausgewählt. Um die Ionic App als native mobile Anwendung erscheinen zu lassen, wurde **Capacitor** verwendet. Capacitor wurde vom Ionic Team entwickelt und stellt zu einer Webanwendung einen nativen Container bereit, welcher alle benötigten API-Zugriffe erlaubt.

Technologie	Version	Entwickungsansatz	Sprache
Android native	29 SDK	Native	JAVA
iOS Native	Xcode 12	Native	Swift
React Native	16.11.0	Interpretiert	JavaScript
Flutter	1.0.0+1	Cross-kompiliert	Dart
Ionic/Capacitor	5.0.7	Hybrid	TypeScript

Tabelle 2: Liste der verwendeten Frameworks und Technologien

Da Flutter von Anfang an gut von der Entwickler-Community aufgenommen wurde und immer häufiger zum Einsatz kommt, wurde für den **cross-kompilierten Entwicklungsansatz** das relativ neue **Flutter Framework** ausgewählt.

Für den **interpretierten Entwicklungsansatz** wurde **React Native** gewählt. Nach Ionic ist es das am meisten verwendete plattformübergreifende Framework.

3.3 UI-Interaktionsszenarien

Die Gesten, welche in dieser Masterarbeit analysiert und ausgewertet werden sind: Swipe, Scroll und Touch. Bei einem Swipe wird ein Finger an den Touchscreen gelegt und horizontal in eine bestimmte Richtung, entweder nach rechts oder links, über den Bildschirm gezogen und wieder abgehoben. Das Scrollen ist hingegen eine vertikale Swipe-Bewegung, also von unten nach oben oder von oben nach unten. Ein Touch ist die Interaktion mit beispielsweise einem Button oder einem anderen Element auf dem Touchscreen.

Der Schwerpunkt der gewählten Interaktionen liegt auf der Interaktion mit der Benutzerin und dem Benutzer. Die drei ausgewählten Interaktionsszenarien werden in den nachfolgenden Unterkapiteln näher erläutert. Die zum Ein-

satz kommende Anwendung, welche die Funktionalität für die Interaktionen beziehungsweise Gesten bereitstellt, wurde eigens dafür implementiert.

Das erste getestete Interaktionsszenario ist das *Öffnen und Schließen des Navigation Drawers*. Der erste Bildschirm ist ein weißer Screen. Hier kann der Navigation Drawer geöffnet werden. Im rechten oberen Eck befindet sich die Navigation zum nächsten Interaktionsszenario. Das zweite Interaktionsszenario ist der *Übergang zwischen zwei Bildschirmen*. Hier befindet sich ein Button mit der Aufschrift NEXT. Wird dieser geklickt, gelangt die Benutzerin oder der Benutzer zur nächsten Seite auf welcher sich ebenfalls ein Button befindet. Diesmal mit der Aufschrift PREVIOUS. Der Übergang zum letzten Szenario befindet sich ebenfalls im rechten oberen Eck des Bildschirmes. Der erste Bildschirm des dritten Interaktionsszenarios *Scrollen durch virtuelle Liste* ist eine Liste, welche Demo-Einträge anzeigt. Beim Start der App werden 1000 Demo-Einträge generiert, welche den Aufbau *Person + Anzahl*, zum Beispiel Person 1, Person 2, Person 3 et cetera aufweisen.

3.3.1 Öffnen und Schließen des Navigation Drawers

Für die Navigation durch eine mobile Anwendung stehen drei unterschiedliche Navigationsdarstellungen zur Verfügung, welche im Material Design festgehalten sind (Google, 2020b). Sie kann entweder über den Navigation Drawer, eine Navigation Bar im unteren Bereich des Bildschirms, oder über Tabs realisiert werden. Welches Navigationsmodell für die Anwendung gewählt wird, hängt davon ab, wie viele Navigationspunkte beziehungsweise Ziele abgebildet werden müssen. Sind es nur wenige (zwei bis maximal fünf), dann können diese über Tabs oder über eine Navigation Bar am unteren Ende des Bildschirms abgebildet werden. Müssen komplexere Strukturen abgebildet werden oder sind mehr als fünf Navigationspunkte vorhanden, dann sollte ein Navigation Drawer bevorzugt werden. Dabei ist zu beachten, dass die Navi-

gation über Tabs und Navigation Bar viel Platz auf dem Bildschirm einnimmt, welcher für das Darstellen des Inhalts nicht mehr zur Verfügung steht. Dies ist vor allem bei kleinen Smartphone-Bildschirmen problematisch. Die Navigation über einen Navigation Drawer nimmt hingegen deutlich weniger Platz in Anspruch und kann auf allen Unterseiten verwendet werden.

Der Drawer kann entweder über das Antippen eines Menü-Icons oder über einen Swipe von der linken Bildschirmkante zur Mitte hin geöffnet werden. Durch das Ziehen der Bildschirmkante öffnet sich die „Schublade“, in welcher die Menüpunkte angezeigt werden. Dabei rückt der aktuell angezeigte Bildschirm - durch einen immer dunkler werdenden Grauschleier während dem Swipe - langsam in den Hintergrund. Der Drawer erscheint hell und im Vordergrund (siehe Abbildung 12, rechtes Bild). Um den geöffneten Drawer zu schließen, stehen zwei Methoden zur Verfügung. Einerseits kann dieser durch ein einmaliges Antippen außerhalb des Drawers erfolgen, also auf den sich im Hintergrund befindlichen Bildschirm, andererseits über einen Swipe von rechts Richtung des linken Bildschirmrandes (siehe Abbildung 12).

Für die Einheitlichkeit der Performanz-Analyse wird der Navigation Drawer über einen Swipe von der linken Bildschirmkante zur Mitte hin geöffnet und über ein einmaliges Antippen außerhalb des Drawers wieder geschlossen. In der Abbildung 12 ist die native Android Implementierung des Interaktionsszenarios *Öffnen und Schließen des Navigation Drawers* dargestellt. Das linke Bild zeigt den Ausgangszustand. Durch das Streichen von links nach rechts wird der resultierende Zustand im rechten Bild erzeugt. Für die Performanz-Analyse wird der Navigation Drawer durch einen entsprechenden Swipe geöffnet. Nach einer Pause von drei Sekunden wird der Drawer durch das einmalige Antippen außerhalb des Drawers geschlossen. Der exakte Ablaufplan ist in Kapitel 3.5.3 dargestellt.

[4]Eigene Darstellung

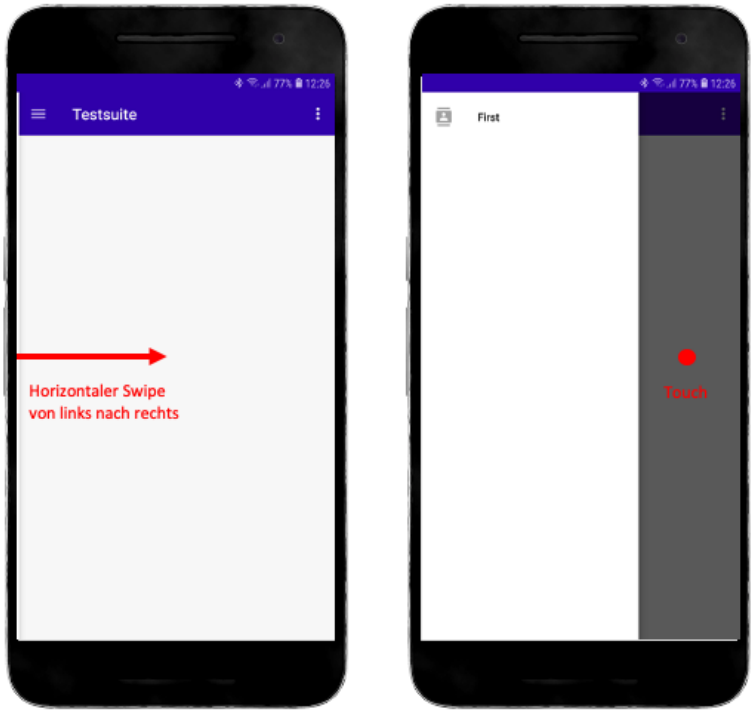

Abbildung 12: Interaktion Öffnen und Schließen des Navigation Drawers[4]

3.3.2 Übergang zwischen zwei Bildschirmen

Da die meisten Anwendungen aus mehr als nur einem Bildschirm bestehen, gehören Übergänge zwischen Bildschirmen ebenfalls zu den grundlegenden UI-Interaktionen. In Abbildung 13 ist die native Android Implementierung des Interaktionsszenarios *Übergang zwischen zwei Bildschirmen* dargestellt. Das linke Bild zeigt den Ausgangszustand. Durch das einmalige Antippen des Buttons mit der Beschriftung NEXT in der Mitte des Bildschirmes wird ein Bildschirmübergang gestartet. Der neue Bildschirm ist dem ersten ident, lediglich der Button in der Mitte enthält mit PREVIOUS eine abweichende Beschriftung. Wird der PREVIOUS-Button einmalig angetippt, so gelangt man zurück zum Ausgangsbildschirm, welcher links in der Abbildung zu sehen ist. Somit kann durch das Antippen der jeweiligen Buttons zwischen beiden Bildschirmen gewechselt werden. Für die Performanz-Analyse wird der Button NEXT einmal angetippt, dann wird vier Sekunden gewartet. Nach vier Sekunden wird der

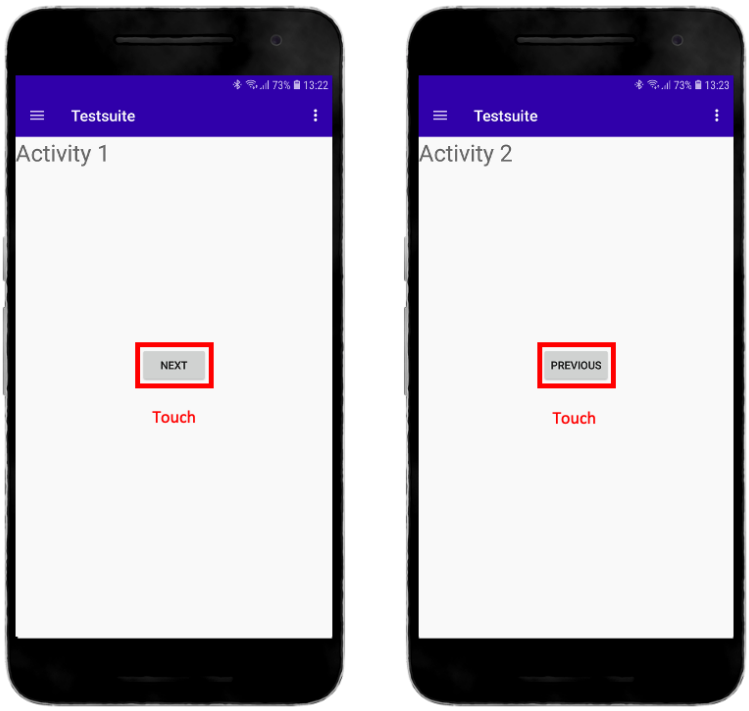

Abbildung 13: Interaktion Übergang zwischen zwei Bildschirmen[5]

Button PREVIOUS angetippt. Der exakte Ablaufplan ist in Kapitel 3.5.3 dargestellt.

3.3.3 Scrollen durch virtuelle Liste

Da der Platz auf einem Smartphone-Bildschirm begrenzt ist, werden häufig virtuelle scrollbare Listen verwendet. Dabei wird vorerst nur der Teil der Daten geladen, welcher auf dem Bildschirm angezeigt werden kann. Alle nicht sichtbaren Einträge müssen deshalb nicht im Vorhinein geladen und gerendert werden. Neue Einträge werden erst zur Laufzeit, wenn eine vertikale Scrollbewegung mit einem Finger durchgeführt wird, geladen. In Abbildung 14 ist die native Android Implementierung des Interaktionsszenarios *Scrollen durch virtuelle Liste* dargestellt. Das linke Bild zeigt den Ausgangszustand. Durch eine vertikale Scroll-Geste von unten nach oben wird der resultierende Zustand

[5]Eigene Darstellung

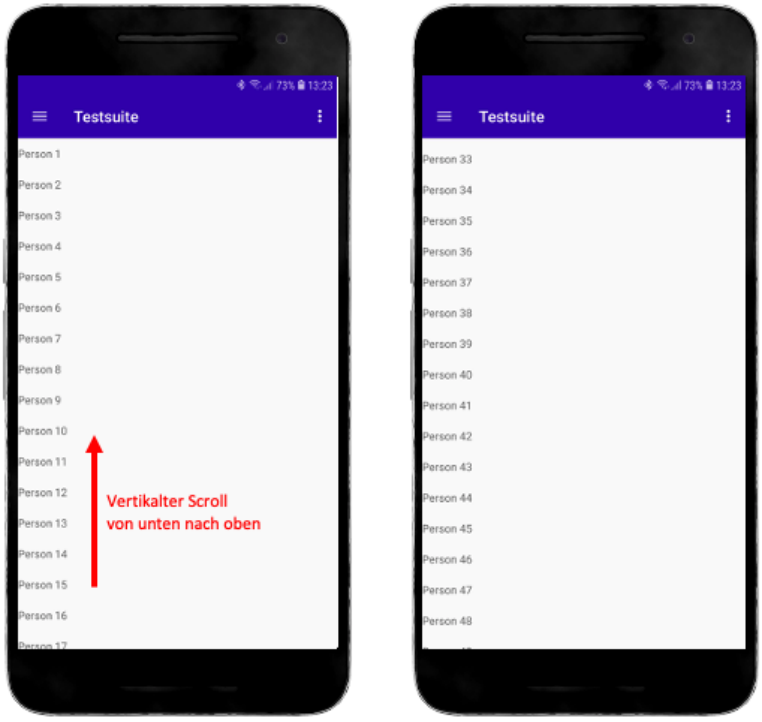

Abbildung 14: Interaktion Scrollen durch eine virtuelle Liste[6]

auf der rechten Seite erzeugt. Die Elemente, welche sich weiter unten in der Liste befinden, werden geladen, gerendert und am Bildschirm angezeigt. Für die Performanz-Analyse wird ein vertikaler Scroll durchgeführt, danach wird vier Sekunden gewartet. Der genaue Ablaufplan ist in Kapitel 3.5.3 dargestellt.

3.4 Einzelheiten der Implementierungen

Um die Forschungsfrage zu beantworten, wurden fünf Instanzen der im Kapitel 3.3 beschriebenen Anwendung samt den drei Interaktionsszenarien entwickelt. Zwei Instanzen wurden mit den nativen Entwicklungsansätzen Google Android und iOS Swift entwickelt und drei Instanzen mit plattformübergreifenden Entwicklungsansätzen React Native, Flutter und Ionic/Capacitor. Um die Anwendungen möglichst sauber und ohne unnötige Zwischenschritte zu

[6]Eigene Darstellung

implementieren, wurde für drei Anwendungen (iOS native, Flutter, React Native) auf die Expertise von Kollegen zurückgegriffen. Der Code wurde zumeist an offizielle Tutorials angelehnt.

3.4.1 Native Android

Der native Entwicklungsansatz für Android wurde bereits genauer in Kapitel 2.1.1 beschrieben. Android bietet alle erforderlichen Komponenten, welche für die Realisierung aller drei Interaktionsszenarien benötigt werden. Somit sind keine zusätzlichen Bibliotheken erforderlich. Für die Realisierung des Navigation Drawers stellt Android das *DrawerLayout* zur Verfügung. Die Navigation zwischen zwei Bildschirmen wurde über Android *Activity* realisiert. Die Implementierung der virtuellen scrollbaren Liste erfolgte über die Komponente *RecyclerView*. Diese benötigt die Adapter-Komponente *ContactsAdapter*.

3.4.2 Native iOS

Die native Entwicklung für iOS Anwendungen wird von Apple bezüglich der Auswahl an Entwicklungstools stark begrenzt. Lange Zeit gab es nur eine einzige IDE namens Xcode, welche von Apple selbst bereitgestellt wurde. Ebenso war lange Zeit die Programmierung lediglich mit Objective-C möglich. Mittlerweile gibt es zumindest eine weitere IDE (Appcode von JetBrains) und als Programmiersprache wird Objective-C immer mehr von Swift abgelöst. Für die Entwicklung selbst, war zusätzlich ein Apple Gerät (Macbook etc.) notwendig. Natürlich bietet das SDK von Apple alle notwendigen Bibliotheken um auf alle Funktionalitäten des Zielgerätes (iPhone, iPad, etc.) zugreifen zu können, weshalb keine externen Bibliotheken in dieser Arbeit verwendet wurden. Die nativen Anwendungen für diese Arbeit wurden in Xcode 12 unter Verwendung der Sprache Swift und dem offiziellen SDK entwickelt.

Für die Implementierung der nativen iOS Anwendung musste zuerst das UIKit importiert werden. Dadurch wird die gesamte Infrastruktur für die Ereignisbehandlung von Multitouch-Gesten bereitgestellt. iOS bietet keine direkte Komponente für einen Navigation Drawer, welcher mit einem Swipe von links nach rechts geöffnet werden kann. Dieser lässt sich jedoch leicht durch das UIViewControllerAnimatedTransitioning Protokoll implementieren. Dieses Protokoll ermöglicht es, Animationen für den Übergang eines View Controllers auf den Bildschirm zu erstellen. Der Übergang zwischen zwei Bildschirmen wurde über die Komponente UIViewController erstellt. Diese stellt die Möglichkeit bereit, auf Benutzerinteraktionen mit den Views zu reagieren. Der Klick selbst erfolgt über ein UIButton Element. Die Implementierung der virtuellen scrollbaren Liste erfolgte über die Komponente UITableView. Diese stellt eine einzelne Spalte mit vertikal scrollenden Inhalten, die in Zeilen unterteilt sind, bereit. Die Demodaten werden beim Start der Anwendung erzeugt und im Speicher abgelegt.

3.4.3 Interpretierter Entwicklungsansatz: React Native

Das React Native Framework wurde 2015 von Facebook als ein Open Source-Projekt veröffentlicht. Es basiert auf der Webbibliothek React und ermöglicht native Anwendungen mit JavaScript zu erstellen. Auch nativer Code für Android oder iOS kann in den React Native Code eingebettet werden. Dies kann zum Beispiel notwendig sein, wenn es ein besonders Element als Komponente noch nicht gibt oder eine Gerätefunktion (zum Beispiel Sensordaten) durch die bereitgestellten APIs nicht abgedeckt sind. Ein besonderes Feature bei React Native ist das sogenannte *Hot Reloading*, welches es ermöglicht, Veränderungen im Code während der Entwicklung sofort in der laufenden Anwendung abzubilden.

Der Navigation Drawer wurde über die Komponente *React Navigation Drawer*

realisiert. Die Navigation zwischen zwei Bildschirmen wurde mit *React Stack Navigator* implementiert. Wenn der Bildschirm gewechselt wird, wird der neue Bildschirm oben auf den Stapel gelegt. Unter Android und iOS unterschiedet sich der Wechsel zwischen Bildschirmen. Android blendet einen neuen Bildschirm von unten ein, wohingegen unter iOS dieser von rechts eingeblendet wird. Für die Implementierung der virtuellen scrollbaren Liste wurde die *FlatList*-Komponente verwendet. Die Demodaten werden beim Start der Anwendung generiert und im Hauptspeicher abgelegt.

3.4.4 Hybrider Entwicklungsansatz: Ionic/Capacitor

Beim Ionic Framework handelt es sich um ein Open-Scoure JavaScript Framework, mit welchem mobile hybride Anwendungen und Progressive Web Apps entwickelt werden können. Die erste Version des Frameworks wurde bereits 2015 vorgestellt und befindet sich aktuell in der fünften Version. Ionic stellt sogenannte UI-Komponenten zur Verfügung, welche sofort eingesetzt oder auch individuell angepasst werden können. Diese Komponenten werden durch die Webstandards HTML, CSS und JavaScript generiert. Wird Ionic für die Erstellung der Benutzeroberfläche verwendet, so sorgt das Framework für die Anpassung der Komponenten an die jeweilige Zielplattform. Somit werden Anwendungen generiert, welche das native Aussehen der jeweiligen Zielplattform aufweisen. Um auf native Bibliotheken des Betriebssystems zugreifen zu können, dazu gehören unter anderem Kamera, GPS oder Bluetooth, muss die Anwendung zuerst in einen nativen Container verpackt werden. Dafür kommt Capacitor zum Einsatz. Capacitor erzeugt zu einer Webanwendung einen nativen Container, welcher alle benötigten Schnittstellen bereitstellt.

Für die Realisierung des Navigation Drawer stellt Ionic die Komponente *<ion-menu>* zur Verfügung. Die Navigation zwischen zwei Bildschirmen wurde über die Ionic Komponente *<ion-button>* und der Property *routerLink="/..."* imple-

mentiert. Wobei *routerLink* den Pfad angibt, der geöffnet werden soll. Für die Implementierung der virtuellen scrollbaren Liste bietet Ionic die Komponente <*ion-virtual-scroll*> an. Dabei wird der Komponente <*ion-item*> ein Array von Datensätzen an den Bildlauf übergeben, aus welchen die Vorlage erstellt wird. Die Demodaten für die virtuelle Liste werden beim Start der Anwendung erzeugt und im Speicher abgelegt. Anschließend wurde die Anwendung in einen Capacitor-Wrapper verpackt und für beide Plattformen bereitgestellt.

3.4.5 Cross-kompilierter Entwicklungsansatz: Flutter

Flutter wurde 2017 von Google ins Leben gerufen und das Flutter SDK wurde im Dezember 2018 als Release 1.0 freigegeben. Es ist Open Source und nutzt als Programmiersprache Dart. Dart ist eine statisch typisierte Skriptsprache, welche bereits 2011 vorgestellt wurde und von Google weiterentwickelt wird. Mit Flutter können sowohl mobile Anwendungen, Desktopanwendungen als auch Webanwendungen programmiert werden. Die wichtigste Komponente in einem Flutter-Projekt sind *Widgets*. Mit diesen wird auch die Benutzerschnittstelle aufgebaut, die weiter ein oder mehrere Children Widgets haben können. In diesen Widgets befindet sich die Anwendungslogik, die über Methoden definiert ist. Das bedeutet, dass jedes Element auf dem Screen ein Widget ist. Somit sind auch einzelne Bildschirme der Anwendung ebenfalls einzelne Widgets.

Jede Seite der Anwendung ist ein Widget vom Typ *Scaffold*. Damit wird die grundlegende visuelle Layout-Struktur des Material Design implementiert. Flutter stellt für die Realisierung des Navigation Drawers die Komponente *Drawer* zur Verfügung, welche in ein Scaffold Widget eingebettet ist. Die Navigation zwischen zwei Bildschirmen, in Flutter als Routes bezeichnet, findet über die Komponente *Navigator*, welche die Methode push() zur Verfügung stellt, statt. Für die Implementierung der virtuellen scrollbaren Liste stellt Flutter den *ListView.builder*-Konstruktor bereit. Die Demodaten werden beim Start

der Anwendung erzeugt und im Hauptspeicher abgelegt.

3.5 Automatische Testfälle

In diesem Abschnitt wird der Ablauf der automatischen Tests beschrieben. Beginnend mit dem Ablauf unter Android gefolgt von dem unter iOS. Im Anschluss werden die drei Testfälle detailliert beschrieben.

3.5.1 Ablauf Android

Im ersten Schritt werden alle vier Anwendungen auf dem Android Smartphone installiert. Als nächstes wird die Bildschirmgröße des Smartphones ermittelt. Diese wird für die automatische Gestensteuerung benötigt, da die Koordinaten der Gesten (Touch, Swipe und Scroll) angegeben werden müssen. Für eine einzelne Touch-Geste für zum Beispiel den Bildschirmwechsel wird eine x- und y-Koordinate benötigt. Eine Swipe- und Scroll-Geste hingegen benötigt je zwei x/y-Tupel mit den entsprechenden Koordinaten. Bei dem ersten x/y-Tupel handelt es sich um den Start der Gesten und das zweite x/y-Tupel gibt die Endkoordinaten an, bei welchen die Gesten aufhören sollen. Auf diese Weise kann die Länge der Swipe- und Scroll-Geste bestimmt werden.

Als Nächstes wird die erste Interaktion ausgewählt. Diese Interaktion wird auf allen vier Anwendungen durchlaufen. Die Anwendungen werden in folgender Reihenfolge getestet: Android nativ, Ionic, ReactNative und abschließend Flutter. Die Interaktionen werden in folgender Reihenfolge getestet: Öffnen und Schließen des Navigation Drawers, Wechsel zwischen zwei Bildschirmen und abschließend das Scrollen durch eine Liste. Nachfolgend werden die Koordinaten für die einzelnen Gesten berechnet. Diese werden immer am Anfang jedes Durchgangs neu errechnet.

Berechnung der Koordinaten für das Öffnen und Schließen des Navigation Drawers

Das Öffnen des Navigation Drawers erfolgt durch eine horizontale Swipe-Geste von links nach rechts. Diese Geste benötigt ein Start- und End-x/y-Tupel. Das Start-Tupel gibt an, wo die Swipe-Geste beginnt, also wo der Finger an den Bildschirm angesetzt wird. Das End-Tupel wiederum gibt an, wo der Finger vom Bildschirm wieder abgehoben wird. Der mit dem Finger zurückgelegte Weg zwischen dem Start- und dem End-Tupel ergibt die Swipe-Geste. Die Start-x-Koordinate ist Null, da der Finger direkt am linken Smartphone-Rand angesetzt wird. Die entsprechende Start-y-Koordinate wird folgender Maßen berechnet: y = Höhe des Bildschirms * 0,5. Der Swipe beginnt also an der linken Kante in der Mitte des Bildschirms. Da die Bewegung horizontal von links nach rechts geführt wird, bleibt die y-Koordinate für das Ende des Swipes gleich. Die x-Koordinate bestimmt, welche Entfernung der angesetzte Finger nach rechts zurücklegt. Im Testfall wurde folgende Entfernung bestimmt: x = Breite des Bildschims * 0,85. Das bedeutet, dass der Finger den Swipe nach 85 % Länge der Bildschirmbreite beendet.

Das Schließen des Navigation Drawers erfolgt über einen Touch außerhalb des ausgezogenen Navigation Drawers. Somit benötigt der Touch nur eine Koordinate. Bei dem Touch werden die Koordinaten folgendermaßen berechnet: x = Breite des Bildschirms * 0,7 und y = Höhe des Bildschirms * 0,5.

Berechnung der Koordinaten für den Übergang zwischen zwei Bildschirmen

Der Übergang zwischen den Bildschirmen funktioniert über einen Touch auf einen Button. Da die Buttons der einzelnen Anwendungen nicht an der exakt gleichen Position platziert sind, werden die Koordinaten für die jeweilige Anwendung einzeln berechnet. Dazu wird die am Anfang ermittelte Bildschirmgröße mit verschiedenen Faktoren multipliziert, sodass der Touch der jeweiligen Anwendung an der korrekten Position durchgeführt wird. Im Falle

der nativen Android Anwendung wird die x-Koordinate folgendermaßen berechnet: x = Breite des Bildschirms * 0,29514. Die dazugehörige y-Koordinate: y = Höhe des Bildschirms * 0,3625. Bei den anderen Anwendungen weichen die Faktoren leicht ab.

Berechnung der Koordinaten für das Scrollen durch virtuelle Liste

Die Scroll-Geste benötigt, wie der Swipe beim Öffnen des Navigation Drawers, sowohl einen Anfangs- als auch einen Endpunkt. Hier wird der Finger am unteren Ende des Bildschirms angesetzt und vertikal nach oben gezogen und wieder aufgehoben. Die Startkoordinaten der Geste werden wie folgt berechnet: x = Breite des Bildschirms * 0,4 und y = Höhe des Bildschirms * 0,7. Die Endkoordinaten: x = Breite des Bildschirms * 0,4 und y = Höhe des Bildschirms * 0,3.

Wenn die Koordinatenberechnung abgeschlossen ist, wird die erste Anwendung in der Liste gestartet. Anschließend wird zehn Sekunden gewartet, da die unterschiedlichen Entwicklungsansätze unterschiedliche Ladezeiten haben. Danach wird das Tool vmstat für die Aufzeichnung der Performanzparameter gestartet, dieses zeichnet 20 Sekunden lang auf. Nach dem Start wird wieder drei Sekunden gewartet, dann folgt der Ablauf der jeweiligen Interaktion. Die Interaktionen werden in einer Schleife durchlaufen, nach jedem Durchlauf wird die Anwendung gewechselt. Der detaillierte Ablaufplan der einzelnen Interaktionsszenarien ist nachfolgend in den Unterkapiteln 3.5.3, 3.5.3 und 3.5.3 zu finden. Für Android wurden die Testfälle mit der Bibliothek AndroidVewClient (Milano, 2020) erstellt. Mit dieser Bibliothek ist eine Skripterstellung der UI-Interaktionen mittels Python-Code möglich. Die Geräteverwaltungsaufgaben werden durch die Android Debugging Bridge (adb) übernommen. Nachdem alle festgelegten Durchläufe einer Interaktion und Anwendung beendet sind, wird Vmstat beendet und alle erhobenen Daten werden in einer Textdatei gespeichert und im Ordner *data* im entsprechenden Unterordner persistiert. Für jede Anwendung ist ein eigener Ordner angelegt. Das bedeutet,

dass nach 20 Durchgängen der vier getesteten Anwendungen, jeweils 20 Textdateien pro Ordner (Anwendung) gespeichert wurden. Nach der Speicherung der Daten wird die jeweilige Anwendung geschlossen. Anschließend werden alle Textdateien einer Interaktion und Anwendung nach Entfernung des Headers in eine CSV-Datei automatisiert zusammengefügt und gespeichert. Somit können sie in weiterer Folge automatisiert ausgelesen und verarbeitet werden. Der genaue Ablauf ist als Diagramm in Abbildung 15 dargestellt.

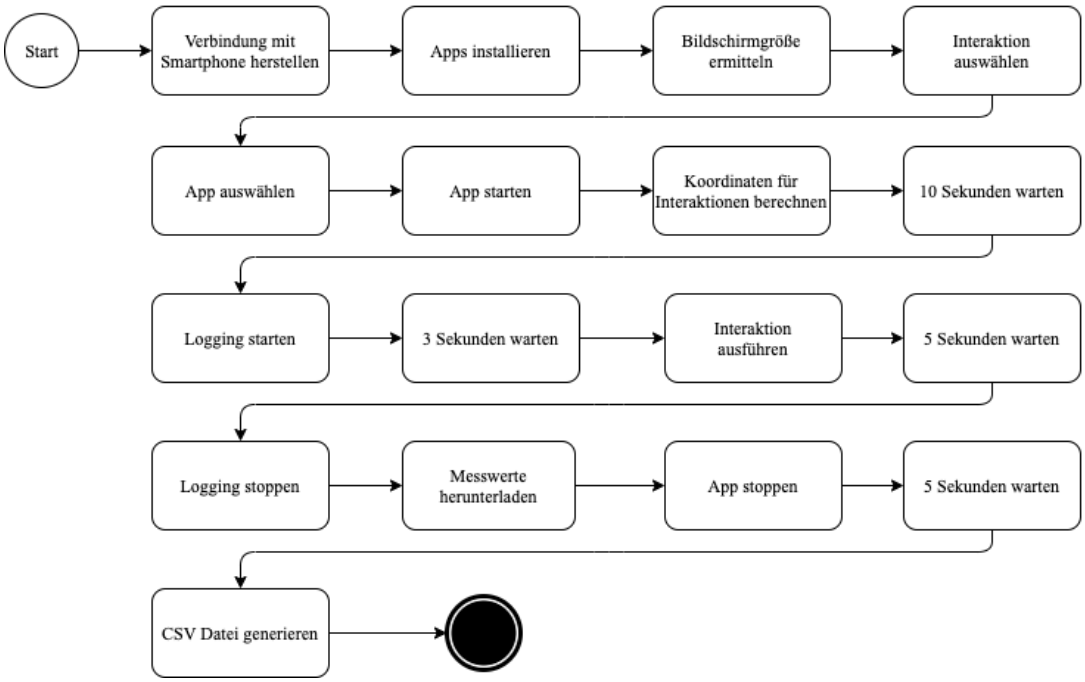

Abbildung 15: Ablaufdiagramm auf Android

3.5.2 Ablauf iOS

Im Falle von iOS ist lediglich die Ausführung der einzelnen Interaktionsszenarien automatisiert, alle anderen Schritte werden wie folgt manuell ausgeführt.

Im ersten Schritt wird die erste Anwendung auf dem iPhone installiert. Bei iOS befinden sich die einzelnen Testfälle direkt in der zu testenden Anwendung. Im Gegensatz zum Android-Testablauf, wo eine Interaktion automatisiert abwech-

selnd auf allen vier Anwendungen getestet wurde, wurde bei iOS auf Grund der vielen erforderlichen manuellen Schritte alle Durchgänge einer Interaktion nacheinander getestet.

Im nächsten Schritt wird der Test ausgewählt und über einen Rechtsklick auf den Test aus dem Dropdown-Menü *Profile "test... "* ausgewählt. Danach startet Xcode das Profiling Tool Instruments aus welchen der *Activity Monitor* ausgewählt wird. Das Starten des Testdurchlaufs erfolgt über einen Klick auf den roten *Recording*-Button. Danach startet die Anwendung auf dem iPhone und hält am zuvor gesetzten Breakpoint, direkt vor der Ausführung der ersten Geste.

Bei Sekunde 20 wird der Breakpoint durch einen Klick auf den Fortsetzungs-Button aufgehoben und ein Durchgang des Interaktionsszenarios wird durchgeführt. Während der Ausführung der Gesten zeichnet der Activity Monitor weiterhin die Performanzparameter auf. Bei Sekunde 45 wird der Aufzeichnungsmodus beendet, dadurch wird auch die Anwendung am iPhone beendet. Das Ergebnis des Durchlaufs wird als eine .trace-Datei gespeichert. Auch dieser Ablauf ist als Diagramm in Abbildung 16 dargestellt.

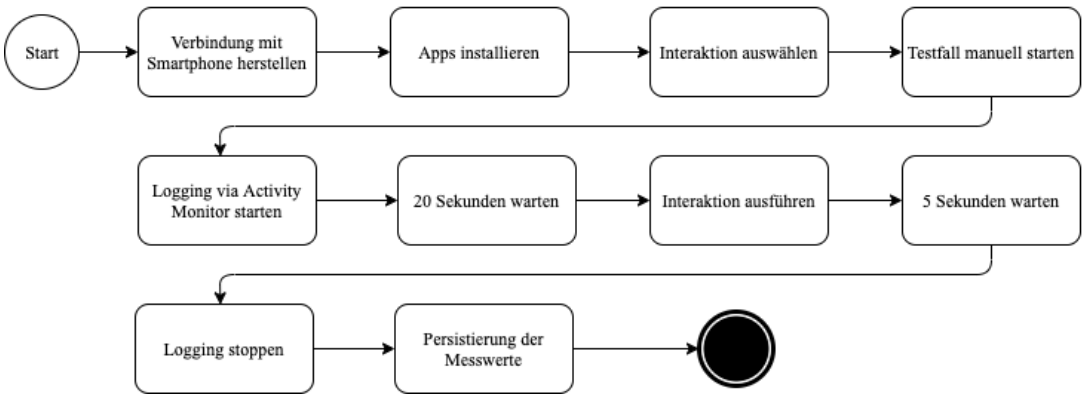

Abbildung 16: Ablaufdiagramm auf iOS

3.5.3 Ablauf der Interaktionen

Zur Durchführung der drei UI-Interaktionsszenarien wurden für jede der vier entwickelten Anwendungen, sowohl auf einem Android als auch auf einem iOS-Smartphone, getimte vollautomatische Testfälle erstellt. Der Ablauf jedes einzeln Testfalls wurde für alle vier Anwendungen eingesetzt. Auf diese Weise ist es möglich, die Ergebnisse direkt miteinander zu vergleichen. Jeder einzelne Schritt eines Testfalls ist zeitlich genau festgelegt. Im Folgenden wird eine detaillierte Beschreibung der drei Testfälle als Ablaufdiagramm gegeben.

Testfall Öffnen und Schließen des Navigation Drawers

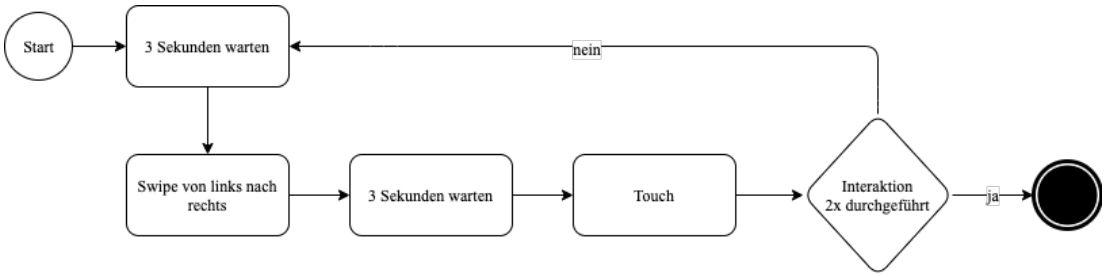

Abbildung 17: Ablaufdiagramm des Testfalls Öffnen und Schließen des Navigation Drawers

Testfall Übergang zwischen zwei Bildschirmen

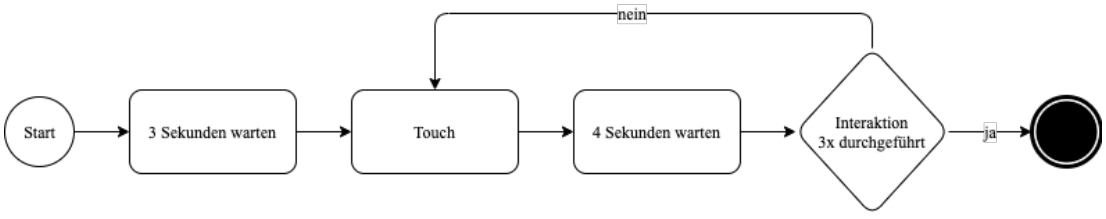

Abbildung 18: Ablaufdiagramm des Testfalls Übergang zwischen zwei Bildschirmen

Testfall Scrollen durch virtuelle Liste

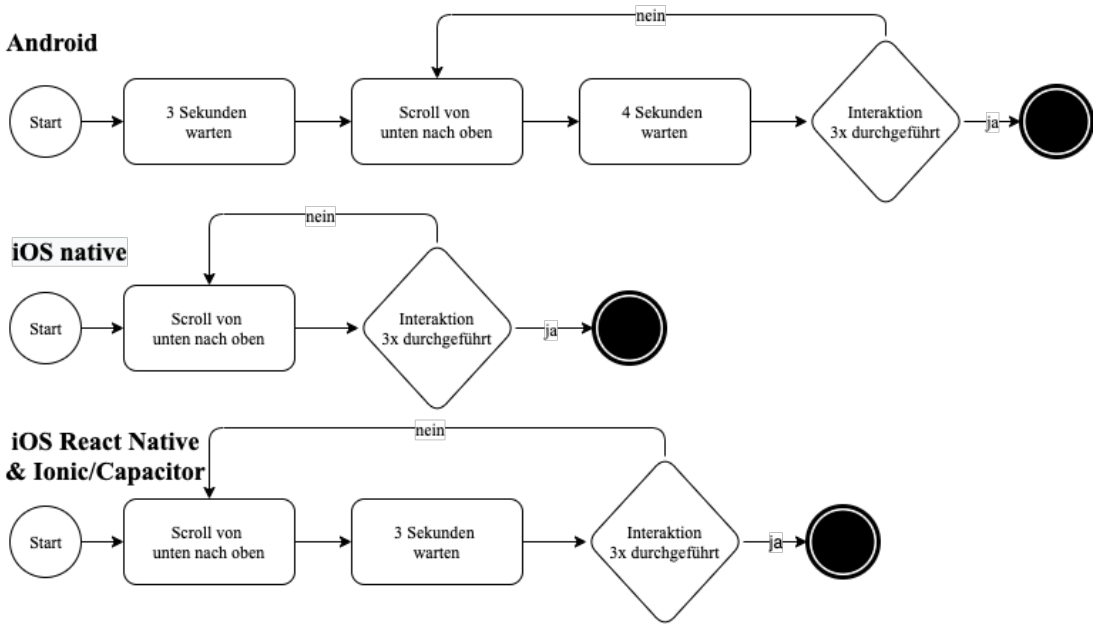

Abbildung 19: Ablaufdiagramm des Testfalls Scrollen durch virtuelle Liste

Im Falle der Scroll-Geste muss erwähnt werden, dass das iOS Betriebssystem mit einem sogenannten *idle Status* zur Schonung des Energieverbrauchs arbeitet. Damit das Gerät möglichst wenig Ressourcen verbraucht, fährt iOS alle nicht genutzten Ressourcen hinunter. Dies geschieht sogar für Mikrosekunden zwischen den Tastenanschlägen (Apple, 2016) und wird beim Scrollen durch eine virtuelle Liste ersichtlich. Während bei der nativen iOS Anwendung das Programm nach einem Scroll wartet bis sich die Anwendung im idle-Status befindet, wird der idle Status bei React Native und Ionic/Capacitor sofort als solcher erkannt. Dies führt dazu, dass die drei Scroll-Gesten sofort nacheinander durchgeführt werden, wenn nicht eine kurze Pause mittels der sleep()-Methode eingefügt wird. Wenn jedoch bei der nativen iOS Anwendung ebenfalls die Pausen eingefügt werden, dauert die Abfolge der drei Scroll-Gesten deutlich über 20 Sekunden. Da das Betriebssystem unterschiedlich lange braucht, um den idle Status herzustellen, sind die Ergebnisse der drei iOS Anwendungen nicht exakt auf die Sekunde miteinander vergleichbar. Dies zeigt sich in den

ausgewerteten Daten in Form der leicht versetzten Nutzungsspitzen, siehe Abbildung 28, nach Durchführung der 20 Testläufe pro Anwendung.

3.6 Bewertungskriterien und Messwerkzeuge

Dieses Kapitel gibt zunächst einen Überblick über die Parameter, welche bei den Performancetests gemessen werden. Danach werden Messwerkzeuge vorgestellt, welche zur Durchführung der Messungen verwendet wurden.

3.6.1 Bewertungskriterien

In diesem Abschnitt werden die beiden gemessenen Parameter aufgelistet und definiert. Des Weiteren wird die Auswahl der Parameter im Hinblick auf die Gesamtleistung einer Anwendung begründet.

CPU-Auslastung

Die CPU ist das Kernstück eines jeden Computers. Sie berechnet und steuert alle Benutzereingaben, Programme, Anfragen und Vorgänge. Je leistungsfähiger die CPU ist, desto schneller werden Berechnungen durchgeführt. Die CPU-Auslastung ist der Prozentsatz der gesamten CPU-Kapazität eines Geräts, welcher von einer Anwendung in einem bestimmten Zeitintervall verwendet wird. Sehr rechenintensive Anwendungen können sich auf andere laufende Prozesse negativ auswirken, die parallel auf dem Smartphone laufen und so die Benutzerfreundlichkeit herabsetzten, da die Anwendungen beispielsweise etwas langsamer laufen oder verlangsamt auf Eingaben reagieren. Während der Testdurchläufe wird die CPU-Auslastung periodisch jede Sekunde über ein Zeitintervall von 20 Sekunden gemessen.

Speicherverbrauch

Im Arbeitsspeicher (RAM) werden alle gerade benötigten Daten abgelegt, welche die gerade auszuführenden Programme oder Programmteile benötigen. Dazu gehören unter anderem das Betriebssystem, alle ausgeführten Anwendungen und auch alle Hintergrundprozesse. Im Gegensatz zu anderen Speichermedien wie der Festplatte, als Hard Disk Drive (HDD) oder Solid State Drive (SSD) bezeichnet, kann auf einen Arbeitsspeicher deutlich schneller zugegriffen (gelesen und geschrieben) werden. Der Arbeitsspeicher ist aber kein permanenter, sondern ein flüchtiger Speicher. Wird der Arbeitsspeicher nicht mehr mit Energie versorgt, gehen auch alle dort gespeicherten Daten verloren. Je mehr Anwendungen auf einem Smartphone offen sind, desto mehr Speicherplatz wird im Arbeitsspeicher belegt aber desto schneller kann zwischen den Anwendungen gewechselt werden. Ist die ganze RAM-Kapazität ausgeschöpft, so werden automatisch Daten gelöscht oder auf die Festplatte verschoben (Swapping). Bei Android ist Swapping per default nicht aktiviert. iOS hingegen erlaubt kein Swapping (Apple, 2013). Wird eine Anwendung, dessen Daten gelöscht wurden wieder aufgemacht, müssen die Daten neu geladen werden, was einen erhöhten Zeit- und Energiebedarf mit sich bringt.

Wenn eine Anwendung geöffnet wird, wird ihr ein Teil des RAM-Speichers zugewiesen. Der Speicherverbrauch gibt die Menge des von der Anwendung zugewiesenen RAM-Speichers an. Die Auslastung des Arbeitsspeichers wird während der Durchführung der einzelnen Interaktionen jede Sekunde über ein Zeitintervall von 20 Sekunden gemessen.

3.6.2 Messwerkzeuge

Tabelle 3 gibt einen Überblick über die Tools, die zur Messung der beiden Leistungsparameter bei Android und iOS herangezogen werden.

	Android	iOS
CPU	vmstat	Instruments Tool (Activity Monitor)
Memory	vmstat	Instruments Tool (Activity Monitor)

Tabelle 3: Messwerkzeuge für beide Plattformen

Android

Für die Messungen an Android wurde das Monitoring-Tool vmstat (Virtual Memory Statistics) (Fischer, 2020) verwendet. Sowohl Android als auch vmstat basieren auf dem Betriebssystem Linux. Über vmstat konnten beide Leistungsparameter (CPU und RAM) gemessen werden. Während der Testfälle zeichnet vmstat 20 Sekunden lang mit einem Aufzeichnungsintervall von einer Sekunde auf. Dabei wurden die CPU-Auslastung sowie der freie Arbeitsspeicher des Geräts während der Ausführung der Gesten aufgezeichnet.

iOS

Apple stellt für das Testen der Performanz das Tool *Instruments*, welches mit Xcode standardmäßig mitgeliefert wird, zur Verfügung. Instruments besteht aus einzelnen Messfunktionen. Für die Messung der CPU-Auslastung und des Arbeitsspeichers wurde die Funktion Activity Monitor verwendet. Activity Monitor lässt eine Aufnahme der einzelnen Testfälle zu. Das Resultat einer Messung ist eine Zeitleiste, welche als Graph dargestellt ist.

3.6.3 Testgeräte

Tabelle 4 gibt einen Überblick über die für die Performanz-Analyse verwendeten Smartphones und deren gerätespezifischen Details. Das Betriebssystem der Smartphones wurde auf die maximal unterstütze Version des Herstellers

Hersteller	Modell	OS Version	Memory (RAM)	CPU
Samsung	Galaxy S7	Android 8.0.0	4 GB	Samsung Exynos 8890 (64 Bit) 4 x 2,3 GHz 4 x 1,6 GHz
Apple	iPhone 8 Plus	iOS 14.1	3 GB	Hexa-Core (64 Bit) 2 x 2,39 GHz 4 x 1,42 GHz

Tabelle 4: Liste der verwendeten Smartphones

aktualisiert. Das iPhone 8 Plus wurde auf Werkseinstellungen zurückgesetzt. Das Samsung Galaxy S7 wurde nicht auf Werkseinstellungen zurückgesetzt, da es sich in Verwendung befindet. Vor dem Testen wurden bei beiden Smartphones der Airplane-Modus aktiviert und alle offenen Anwendungen geschlossen. Das Samsung Galaxy S7 wurde erstmalig im März 2016 und das iPhone 8 Plus im September 2017 veröffentlicht.

4. Ergebnisse

Dieses Kapitel widmet sich den Ergebnissen dieser Studie und ist in die Unterkapitel Android und iOS unterteilt. Zuerst werden jeweils die Ergebnisse für die CPU-Auslastung dargestellt, gefolgt vom Speicherverbrauch der entwickelten Anwendungen. Die Ergebnisse wurden erzielt, indem jeder Testfall 20 mal ausgeführt wurde. Jeder Testfall dauert exakt 20 Sekunden. Die nachfolgenden Diagramme zeigen die Durchschnittswerte der 20 Testdurchläufe pro Testfall. Dabei wird die blaue Linie für die nativen Anwendungen (Android und iOS) verwendet, die orange Linie zeigt die React Native Anwendung, die graue Linie zeigt die Flutter Anwendung und die gelbe Linie stellt die Ionic/Capacitor Anwendung dar.

Anhang A und B zeigen die Ergebnis-Diagramme der CPU-Nutzung und der Arbeitsspeichernutzung pro Testfall und pro Anwendung. Die zusätzlichen grauen Linien zeigen die Varianz der 20 Testdurchläufe, genauer zeigen sie eine Streuung zwischen dem 5- und dem 95 %-Quantil um die durchschnittliche CPU-Nutzung beziehungsweise um den durchschnittlichen Speicherverbrauch der verschiedenen Anwendungen.

4.1 Android

Die in den Unterkapiteln 3.5.3, 3.5.3 und 3.5.3 vorgestellten Testfälle wurden in einem Python-Programm verfasst. Die Ergebnisse wurden erzielt, indem jeder

einzelne Testfall 20 Mal für jede Anwendung in abwechselnder Reihenfolge durchgeführt wurde. In den Diagrammen ist zu erkennen, dass die einzelnen Gesten mit etwa einer Sekunde Verzögerung eine Auswirkung sowohl auf die CPU-Auslastung als auch auf den Speicherverbrauch haben. Alle Interaktionsszenarien starten mit der ersten Geste bei Sekunde 3, die Auswirkungen sind jedoch zumeist ab Sekunde 4 zu beobachten.

4.1.1 CPU-Auslastung

Die CPU-Auslastung wurde während der Testfälle periodisch jede Sekunde gemessen. In den nachfolgenden Diagrammen zeigt die vertikale Achse die durchschnittliche CPU-Auslastung in Prozent. Auf der horizontalen Achse ist die Zeit in Sekunden dargestellt.

Öffnen und Schließen des Navigation Drawers

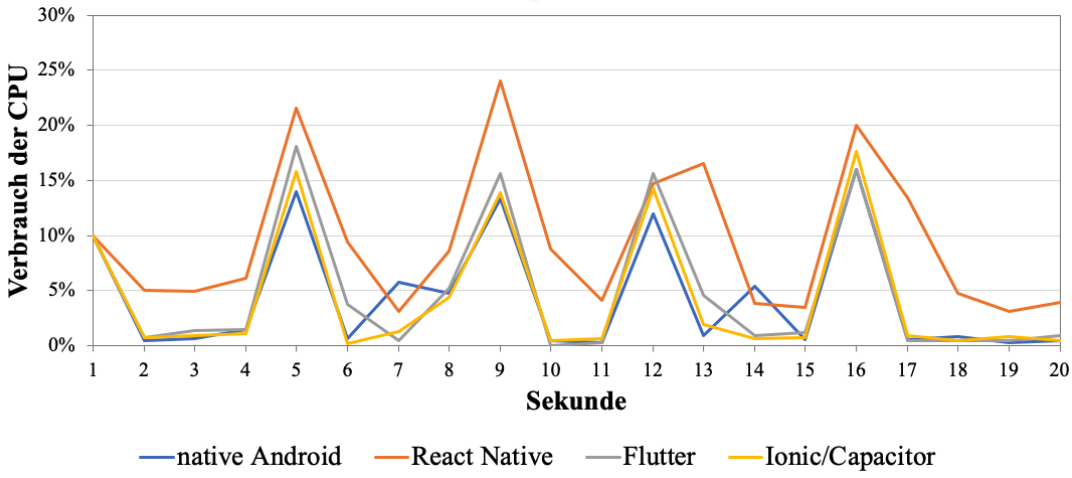

Abbildung 20: Durchschnittswerte aller vier Anwendungen beim Öffnen und Schließen des Navigation Drawers auf Android

In Abbildung 20 sind bei allen Anwendungen vier Nutzungsspitzen zu sehen. Dabei handelt es sich um die durchgeführten Gesten: Swipe von links nach

rechts, Touch, Swipe von links nach rechts und Touch. Es lässt sich beobachten, dass die beiden Anwendungen, welche mit den plattformübergreifenden Entwicklungsansätzen Flutter und Ionic/Capacitor programmiert wurden, bei den einzelnen Gesten fast die gleiche Menge an CPU verbrauchen. Bei den Nutzungsspitzen sind es zwischen 14 und 18,1 %, während der Pausen zwischen den Gesten sinkt der CPU-Verbrauch auf 0 bis 1 %. Nach dem letzten Schließen des Navigation Drawers durch einen Touch sinkt der CPU-Verbrauch unter 1 %. Die native Android Anwendung verbraucht ähnlich viel CPU bei den Nutzungsspitzen, wie die Flutter und Ionic/Capacitor Anwendungen. Der Unterschied ist jedoch nach dem Öffnen des Navigation Drawers durch einen Swipe von links nach rechts. Hier zeigen sich zwei weitere kleine Nutzungsspitzen, welche zwischen 4 und 6 % liegen. Nach dem Schließen des Navigation Drawers durch einen Touch hingegen, sinkt der Verbrauch der CPU auf etwa 1 % ab. Auch nach der letzten Geste sinkt der Verbrauch, wie bei Flutter und Ionic/Capacitor auf 0 bis 1 % ab. Am meisten CPU benötigt die React Native Anwendung. Sie hat von allen Anwendungen die höchsten Nutzungsspitzen. Auch sinkt der CPU-Verbrauch in den drei sekündigen Pausen zwischen den einzelnen Gesten auf maximal 3 % ab. Bei den anderen Anwendungen sinkt der Verbrauch auf 0 bis 1 % ab.

Übergang zwischen zwei Bildschirmen

Beim Übergang zwischen zwei Bildschirmen sind in Abbildung 21 drei Nutzungsspitzen sichtbar. Dabei handelt es sich um die drei Wechsel zwischen den Bildschirmen. Dazwischen liegt eine vier sekündige Pause. Bei diesem Interaktionsszenario verbraucht die native Android Anwendung die wenigste CPU. Die drei Spitzen liegen zwischen 7 und 12 %, in den Pausen benötigt die Anwendung keine bis maximal 1 % CPU. Die Flutter und Ionic/Capacitor Anwendungen haben auch bei diesem Szenario einen ähnlichen CPU-Verbrauch, wobei Flutter beim ersten und dritten Wechsel um 2 % weniger CPU benö-

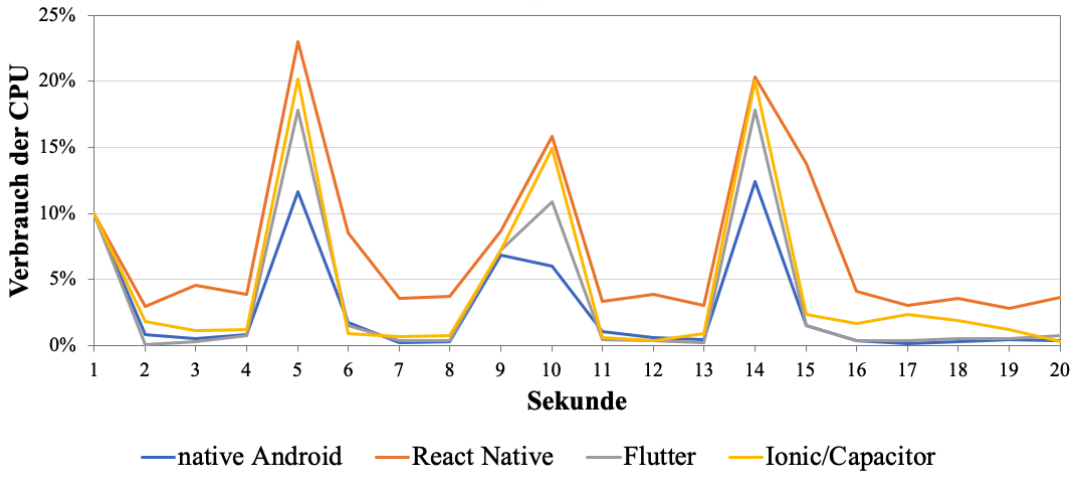

Abbildung 21: Durchschnittswerte aller vier Anwendungen beim Übergang zwischen zwei Bildschirmen auf Android

tigt. Ein größerer Unterschied ist nur beim zweiten Wechsel ersichtlich, wo die Flutter Anwendung um etwa 4 % weniger CPU als die Ionic/Capacitor Anwendung benötigt. In den Pausen zwischen den Gesten sinkt bei beiden der CPU-Verbrauch auf 0 bis 1 % ab. Nach der letzten Geste verbraucht die Ionic/Capacitor Anwendung mit 2 % etwas mehr, als die Flutter und die native Android Anwendung. Am meisten CPU benötigt auch bei diesem Interaktionsszenario die React Native Anwendung. Bei den drei Nutzungsspitzen sind die Unterschiede mit maximal 3 %, jedoch nicht so groß. Aber auch hier sinkt der CPU-Verbrauch in den Pausen nicht ab sondern pendelt sich immer bei 3 bis 4 % ein.

Scrollen durch virtuelle Liste

Beim dritten Interaktionsszenario, dem Scrollen durch eine virtuelle Liste wurden drei von unten nach oben geführte Scroll-Gesten durchgeführt, dazwischen war eine vier sekündige Pause. Wie in Abbildung 22 zu sehen, benötigt die native Android Anwendung am wenigsten CPU. Der erste Scroll benötigt etwa 15 % CPU zum Berechnen und Laden der noch nicht sichtbaren Personenein-

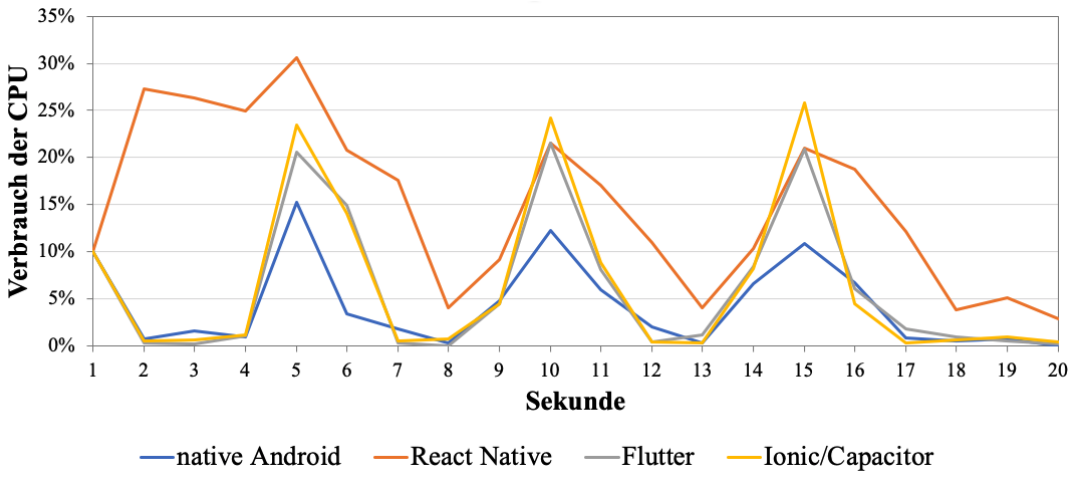

Abbildung 22: Durchschnittswerte aller vier Anwendungen beim Scrollen durch virtuelle Liste auf Android

träge in der Liste. Der zweite Scroll benötigt etwas weniger und der dritte mit 11 % am wenigsten CPU. Es ist auch zu beobachten, dass die Berechnung nach einer Scroll-Geste etwas länger dauert, da der CPU-Verbrauch in der vier sekündigen Pause nur für eine Sekunde auf Null absinkt. Nach dem letzten Scroll sinkt der CPU-Verbrauch auf Null ab. An zweiter Stelle befindet sich die Flutter Anwendung, dicht gefolgt von der Ionic/Capacitor Anwendung. Beide haben einen sehr ähnlichen CPU-Verbrauch, wobei die Flutter Anwendung in den Nutzungsspitzen zwischen 2 und 4 % weniger CPU für die Berechnung benötigt. Ab Sekunde 17 verhalten sich beide Anwendungen wie die native Anwendung und sinken auf Null ab. Ganz anders verhält sich der CPU-Verbrauch der React Native Anwendung. Konträr zu den anderen drei Anwendungen, ist hier gleich von der ersten Sekunde an ein Anstieg des CPU-Verbrauchs zu erkennen. Bei der ersten Scroll-Geste steigt der CPU-Verbrauch auf 31 %, die Berechnung der Personeneinträge dauert etwas länger als bei den anderen drei Anwendungen, denn der Tiefstwert ist in Sekunde 8 mit 4 %. Danach steigt der CPU-Verbrauch nach der zweiten Scroll-Geste auf 22 % ex aequo mit der Flutter Anwendung. Auch die dritte Nutzungsspitze ist mit 21 % gleich hoch, wie der Verbrauch der Flutter Anwendung. Der Unterschied zu Flutter und Ionic/Capacitor besteht darin, dass nach einer Scroll-Geste die Berechnung etwas

länger andauert. Dies zeigt sich in der langsamer abfallenden Linie. Auch beim dritten Interaktionsszenario sinkt der CPU-Verbrauch in den Pausen nicht auf Null sondern bleibt bei 4 %.

4.1.2 Speicherverbrauch

Während der Ausführung der Testfälle wurde die Menge des zur Verfügung stehenden freien Arbeitsspeichers (in Folge *free memory* genannt) periodisch jede Sekunde gemessen. Als Anfangswert wurde bei jedem der 20 Durchläufe der free memory der ersten Sekunde als Ausgangsbasis ermittelt. In weiterer Folge wurden die free memory Werte der Sekunden 2 bis 20 mit dem free memory Wert der Sekunde 1 verglichen und als Abweichung im Diagramm dargestellt. Die vertikale Achse zeigt die durchschnittliche Abweichung zum Ausgangswert des free memory von Sekunde 1 in Prozent. Auf der horizontalen Achse ist die Zeit in Sekunden dargestellt. Die Minuswerte in den Diagrammen ergeben sich aus frei werdendem Arbeitsspeicher.

Öffnen und Schließen des Navigation Drawers

Wie in Abbildung 23 ersichtlich, verbraucht die native Android Anwendung konstant sehr wenig Speicher, nur bei den einzelnen Gesten sind leichte Anstiege im Verbrauch sichtbar. Die React Native Implementierung verbraucht in etwa um 2 bis 5 % mehr Arbeitsspeicher als die native Anwendung. Dabei zeigt sich ein Peak bei Sekunde 12 mit knapp 7 % über der nativen Anwendung. Nach dem zweiten Öffnen und Schließen des Navigation Drawers pendelt sich der Verbrauch des free memory der Anwendung auf etwa 4,3 % ein. Bis zur ersten Geste - das Öffnen des Navigation Drawers - bei Sekunde 4, hat Flutter anfangs einen konstant niedrigen Speicherverbrauch. Dann steigt der Speicherverbrauch um etwa 4,5 % an. Beim Schließen des Navigation Drawers steigt der Speicherverbrauch nochmals auf etwa 7 % an. Während der drei sekündigen

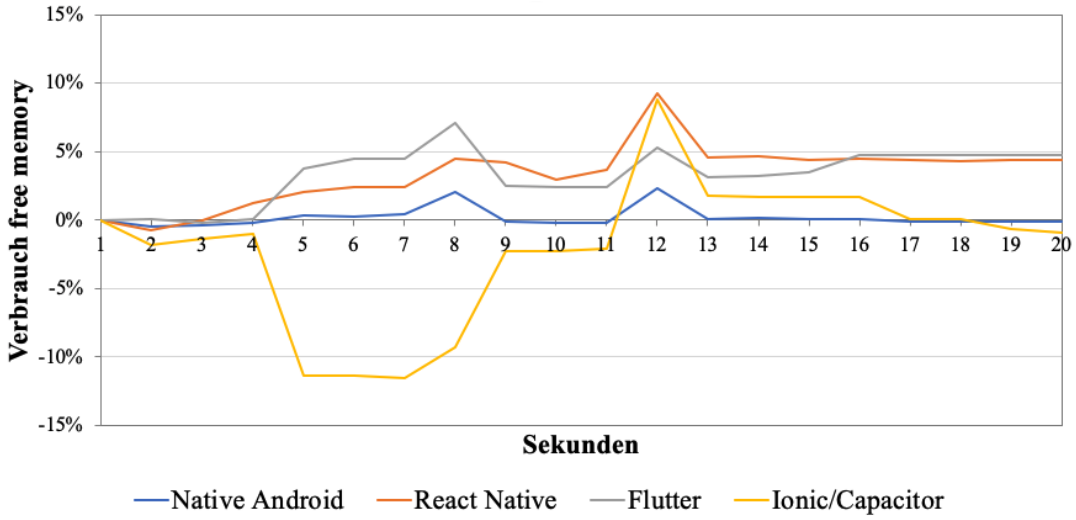

Abbildung 23: Durchschnittswerte aller vier Anwendungen beim Öffnen und Schließen des Navigation Drawers auf Android

Pause sinkt der Verbrauch des free memory auf etwa 2,5 %. Beim wiederholten Öffnen und Schließen steigt der Speicherverbrauch nochmals etwas an, jedoch weniger als beim ersten Öffnen und Schließen und pendelt sich ab Sekunde 16 bei etwa 4,8 % ein. Im Gegensatz dazu verbraucht die native Anwendung zu diesem Zeitpunkt kaum noch Arbeitsspeicher. Die Ionic/Capacitor Anwendung zeigt in der ersten Hälfte eine gegensätzliche Tendenz. Beim ersten Öffnen und Schließen des Navigation Drawers wird free memory frei. Erst bei Sekunde 11 steigt der Speicherverbrauch auf etwa 8,8 %, was in etwa auch der React Native Anwendung entspricht. Nach der zweiten Ausführung der Gesten zu Öffnen und Schließen des Navigation Drawers sinkt der Speicherverbrauch wieder auf etwa den Anfangswert.

Übergang zwischen zwei Bildschirmen

Abbildung 24 zeigt, dass die Interaktion Übergang zwischen zwei Bildschirmen die native Android Anwendung im Durchschnitt am wenigsten Arbeitsspeicher verbraucht. An zweiter Stelle befindet sich die React Native Anwendung. Die Verlaufskurve ist ähnlich zur nativen Anwendung, jedoch stetig leicht

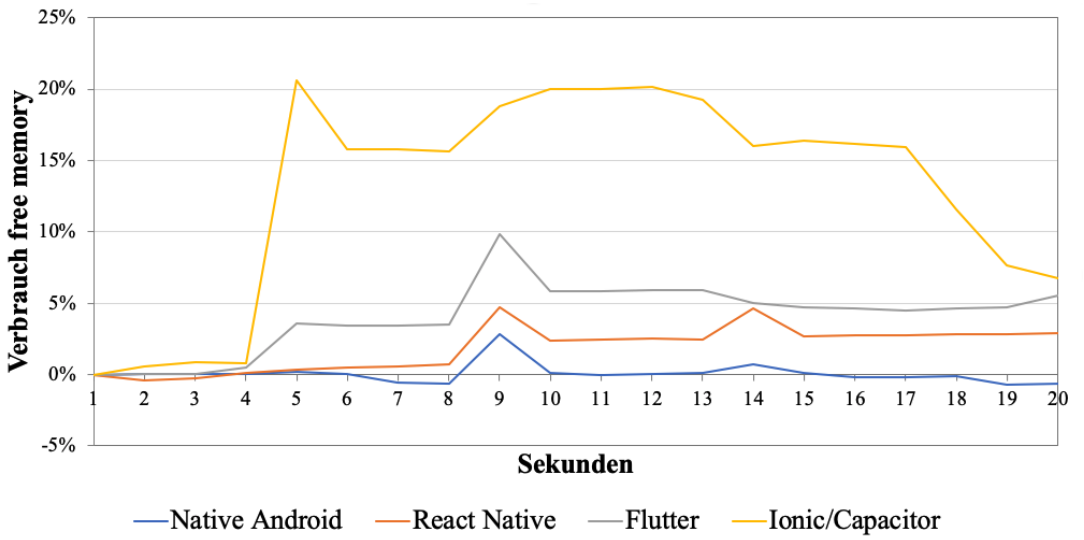

Abbildung 24: Durchschnittswerte aller vier Anwendungen beim Übergang zwischen zwei Bildschirme auf Android

ansteigend. Der Verbrauch des free memory bleibt aber unter 5 %. Flutter zeigt gleich beim ersten Wechsel des Bildschirms bei Sekunde 4 bereits einen leichten Anstieg des Speicherverbrauchs auf etwa 3,5 %. Bei Sekunde 9 verbraucht Flutter etwa 10 % mehr Arbeitsspeicher als am Anfang, danach geht der Speicherverbrauch langsam zurück. In der letzten Sekunde steigt der Speicherverbrauch um etwa 1 % wieder an. Am meisten Speicher verbraucht die Ionic/Capacitor Anwendung. Gleich beim ersten Wechsel des Bildschirms bei Sekunde 4 steigt der Speicherverbrauch um knapp über 20 % an, sinkt in der vier sekündigen Pause aber wieder etwas ab. Beim wiederholten Bildschirmwechsel steigt der Speicherverbrauch wieder auf über 20 % des Anfangswertes an. Nach dem letzten Bildschirmwechsel sinkt der Speicherverbrauch der Anwendung stetig ab und pendelt sich bei knapp unter 7 % ein.

Scrollen durch virtuelle Liste

Bei den Durchschnittsergebnissen der Interaktion Scrollen durch eine virtuelle Liste in Abbildung 25 ist zu erkennen, dass die Ionic/Capacitor Anwendung

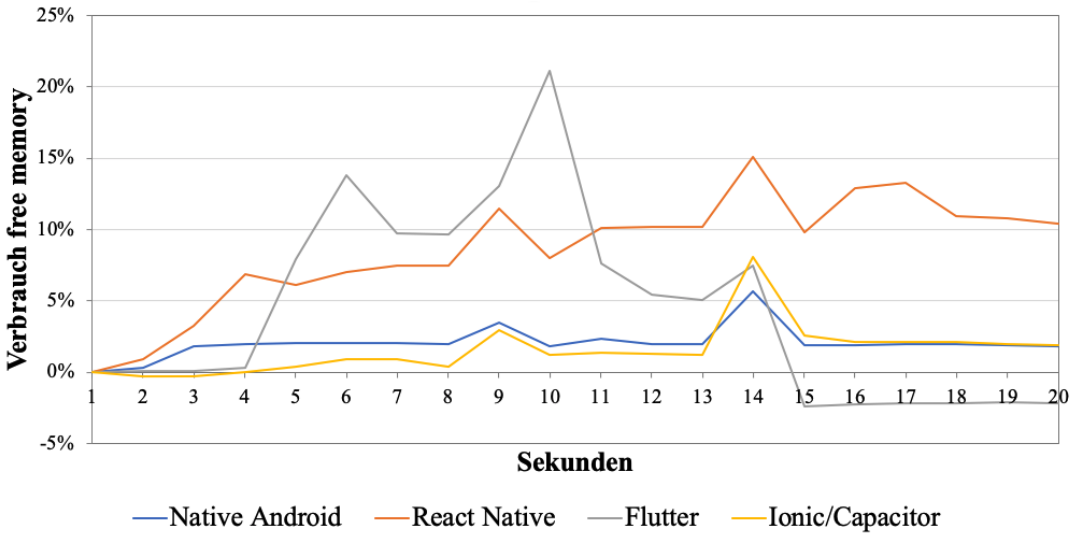

Abbildung 25: Durchschnittswerte aller vier Anwendungen beim Scrollen durch eine virtuelle Liste auf Android

bis Sekunde 13 weniger free memory verbraucht als die native Anwendung. Etwa bei der dritten Scroll-Geste in der Sekunde 14 befindet sich eine Spitze, welche die native Anwendung um knapp 2,4 % übersteigt. Nach der dritten Scroll-Geste pendeln sich beide Anwendungen auf etwa 2 % ein. Sowohl bei der nativen als auch bei der Ionic/Capacitor Anwendung ist die höchste Spitze bei Sekunde 14 mit 5,64 % bei der nativen Anwendung und mit 8,04 % bei der Ionic/Capacitor Anwendung. Ansonsten wird der free memory mit deutlich unter 5 % belastet. Im Gegensatz dazu ist bei Flutter nach der ersten Scroll-Geste ein steiler Anstieg zu beobachten. Für die Berechnung steigt der Verbrauch des free memory auf 13,8 % an. Nach der zweiten Scroll-Geste zeigt die Flutter Anwendung den Peak im Verbrauch des free memory um über 21 % an. In der darauffolgenden vier sekündigen Pause sinkt der Verbrauch des Arbeitsspeichers wieder ab. Nach der dritten Scroll-Geste steigt der Verbrauch deutlich weniger an, als bei der ersten und zweiten Ausführung der Geste, in etwa auf das Niveau der Ionic/Capacitor Anwendung. Danach sinkt der Verbrauch des Arbeitsspeichers unter den Ausgangswert auf knapp über 2 % ab. Die React Native Implementierung zeigt einen stetigen Anstieg im Verbrauch des free memory mit drei Spitzen um die drei Scroll-Gesten. Die erste

und zweite Scroll-Geste verbraucht mehr Arbeitsspeicher als die native und Ionic/Capacitor Anwendungen aber weniger als die Flutter Anwendung. Ab Sekunde 11 verbraucht die React Native Anwendung am meisten Arbeitsspeicher. Im Gegensatz zu den anderen drei Anwendungen, hat die React Native Anwendung auch nach der dritten Scroll-Geste einen Anstieg im Verbrauch des free memory. Nach der Ausführung aller drei Scroll-Gesten bleibt der Verbrauch des Arbeitsspeichers bei etwa 10 % relativ hoch. Hier verbrauchen die anderen drei Anwendungen deutlich weniger.

4.2 iOS

Im Gegensatz zu Android, welches ein offenes Betriebssystem ist und Software von Drittanbietern für Performanzmessungen zulässt, ist iOS ein geschlossenes Betriebssystem. Dies äußert sich insofern, dass für das Testen die Entwicklungsumgebung von Apple, Xcode zum Testen der Anwendungen herangezogen werden muss. Hier ist es nicht möglich einen Testfall über alle Anwendungen abwechselnd zu testen, da jeder einzelne Test in der entsprechenden Anwendung manuell gestartet werden muss. Die einzelnen Tests für die in den Unterkapiteln 3.5.3, 3.5.3 und 3.5.3 beschriebenen Interaktionsszenarien wurden direkt in der zu testenden Anwendung verfasst. Die Ergebnisse wurden erzielt, indem die Aufzeichnung jedes Testfalls manuell erfolgte. Nach Programmstart wurde die Anwendung vor der ersten Geste mittels Breakpoint angehalten. Im nächsten Schritt wurde der Breakpoint nach 20 Sekunden Laufzeit manuell aufgehoben. Deshalb kann es eine Verschiebung um eine halbe Sekunde vor oder zurück geben. Die Testfälle wurden 20 Mal durchgeführt. Dies kann dazu führen, dass die Gesten nicht exakt zur gleichen Sekunde durchgeführt wurden. Nach Beendigung jedes einzelnen Durchgangs wurden aus allen am Smartphone ablaufenden Prozessen der entsprechende Testprozess herausgefiltert. So können die Ergebnisse auf den Prozess eingeschränkt betrachtet werden.

Da eine automatisierte Extraktion der Werte nicht möglich war, wurden die Ergebnisse im nächsten Schritt manuell in eine Excel-Datei übertragen und für die Auswertung gespeichert.

Die Testfälle betreffend der Flutter Anwendung konnten auf Grund technischer Probleme nicht auf iOS ausgeführt werden. Die Flutter-Anwendung an sich, läuft auf dem iPhone einwandfrei. Wurde jedoch ein automatisierter Test gestartet, brach die Ausführung mit folgender Ausgabe ab:

```
ld: framework not found Flutter
clang: error: linker command failed with exit code 1
(use -v to see invocation)
```

Trotz intensiver Recherche konnte keine Möglichkeit gefunden werden, diesen Fehler zu beheben. Es wurden verschiedene Anleitungen, welche im Internet gefunden wurden ausprobiert, jedoch konnte mit keiner dieser Anleitungen der Fehler behoben werden. Auch eine Darstellung des Problems in einschlägigen Fachforen führte zu keinem Ergebnis. Diese wurden auf den Plattformen Stackoverflow[1] sowie Reddit FlutterDev [2] gepostet.

4.2.1 CPU-Auslastung

Die CPU-Auslastung wurde während der Testfälle periodisch jede Sekunde gemessen. In den nachfolgenden Diagrammen zeigt die vertikale Achse die durchschnittliche CPU-Auslastung in Prozent. Auf der horizontalen Achse ist die Zeit in Sekunden dargestellt.

[1] https://stackoverflow.com/questions/63074892/how-to-run-a-xcode-ui-test-with-flutter-project?noredirect=1#comment111546638_63074892

[2] https://www.reddit.com/r/FlutterDev/comments/hxn2wg/how_to_run_a_xcode_ui_test_with_a_flutter_project/

Öffnen und Schließen des Navigation Drawers

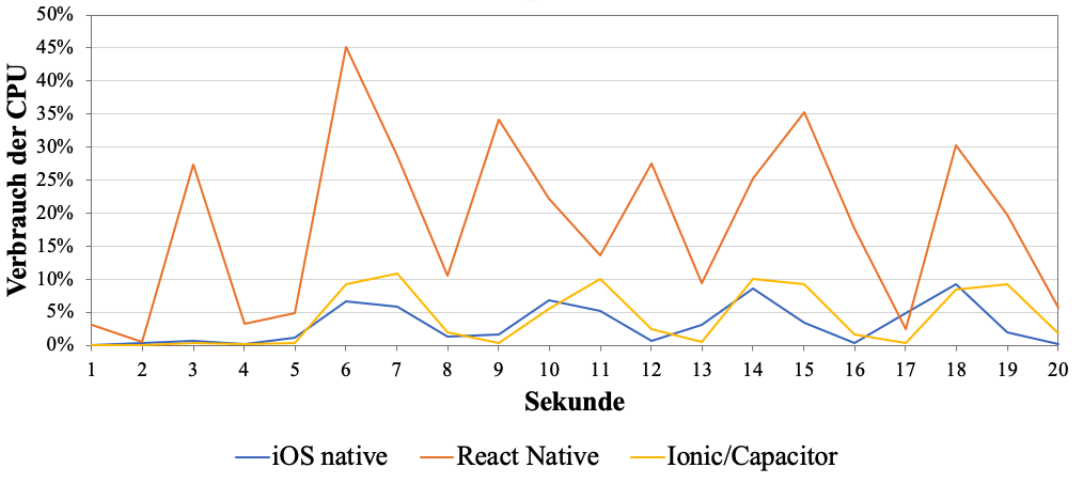

Abbildung 26: Durchschnittswerte aller drei Anwendungen beim Öffnen und Schließen des Navigation Drawers auf iOS

Abbildung 26 zeigt die Durchschnittswerte des CPU-Verbrauchs beim Interaktionsszenario Öffnen und Schließen des Navigation Drawers. Dieses Szenario besteht aus den folgenden Gesten: Swipe von links nach rechts, Touch, Swipe von links nach rechts und Touch. Zwischen den Gesten sind drei Sekunden Pause. Die native iOS Anwendung verbraucht bei allen vier Gesten zwischen 7 und 9 % CPU. Zwischen den Gesten geht der Verbrauch auf 0 bis 1 % hinunter. Die Ionic/Capacitor Anwendung verbraucht bei allen vier Nutzungsspitzen zwischen 9 und 11 % CPU, also nur geringfügig mehr als die native iOS Anwendung. Auch in den Pausen sinkt der Verbrauch auf 0 bis 1 % deutlich. Ganz anders verhält sich die React Native Anwendung. Hier sind sechs Nutzungsspitzen deutlich zu sehen obwohl nur vier Gesten durchgeführt wurden. Bereits bei Sekunde drei ist die erste Spitze zu sehen, wohingegen die anderen beiden Anwendungen keinen CPU-Verbrauch zeigen.

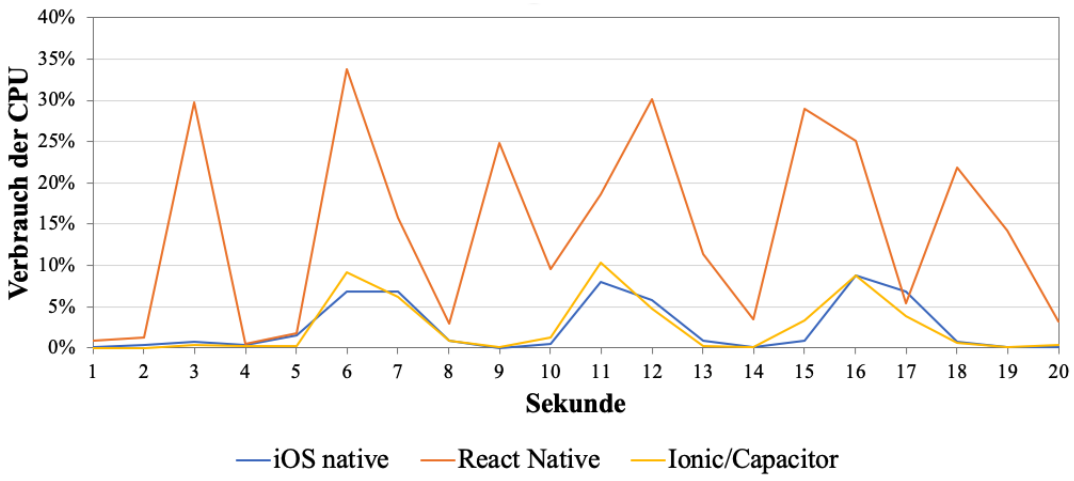

Abbildung 27: Durchschnittswerte aller drei Anwendungen beim Übergang zwischen zwei Bildschirmen auf iOS

Übergang zwischen zwei Bildschirmen

Abbildung 27 zeigt den durchschnittlichen CPU-Verbrauch der drei Anwendungen beim Übergang zwischen zwei Bildschirmen. Während der Testausführung wurde drei Mal der Bildschirm gewechselt. Bei der nativen iOS sowie der Ionic/Capacitor Anwendung sind im Diagramm drei Nutzungsspitzen zu sehen. Die Kurven verlaufen sehr ähnlich, wobei die native Anwendung bei den ersten zwei Wechsel minimal weniger CPU verbraucht als die Ionic/Capacitor Anwendung. Beide Anwendungen benötigen beim Übergang zwischen 7 und 19 % CPU. In den Pausen zwischen den Gesten, sinkt der CPU-Verbrauch auf Null ab. Die React Native Anwendung hingegen, zeigt gleich bei Sekunde 2 eine steile Nutzungsspitze, welche bei 30 % liegt. Beim ersten Übergang ist die zweite Nutzungsspitze bei 34 %. Dies ist der höchste erreichte CPU-Verbrauch bei diesem Interaktionsszenario. In der vier sekündigen Pause sinkt der CPU-Verbrauch bei den anderen zwei Anwendungen auf 0 % ab. Bei der React Native Anwendung ist eine weitere Spitze zu sehen. Beim zweiten Übergang ist die Nutzungsspitze im Gegensatz zu der nativen iOS und der Ionic/Capacitor Anwendung um eine Sekunde nach hinten versetzt. Der CPU-Verbrauch liegt bei Sekunde 12 bei 30 %. Die Nutzungsspitze der beiden anderen An-

wendungen ist bei Sekunde 11 bei 8 beziehungsweise 10 %. Bei Sekunde 15 verursacht der dritte Übergang bei React Native den Peak mit 29 %. Bei den anderen Anwendungen findet dieser eine Sekunde später statt. Danach sinkt der CPU-Verbrauch wieder auf 5 % ab. Während in den letzten zwei Sekunden die anderen Anwendungen keinen CPU-Verbrauch mehr zeigen, hat die React Native Anwendung nochmals eine Nutzungsspitze, bei welcher die Anwendung ebenfalls 22 % CPU benötigt. Bei Sekunde 20 sinkt der Verbrauch auf 3 % ab.

Scrollen durch virtuelle Liste

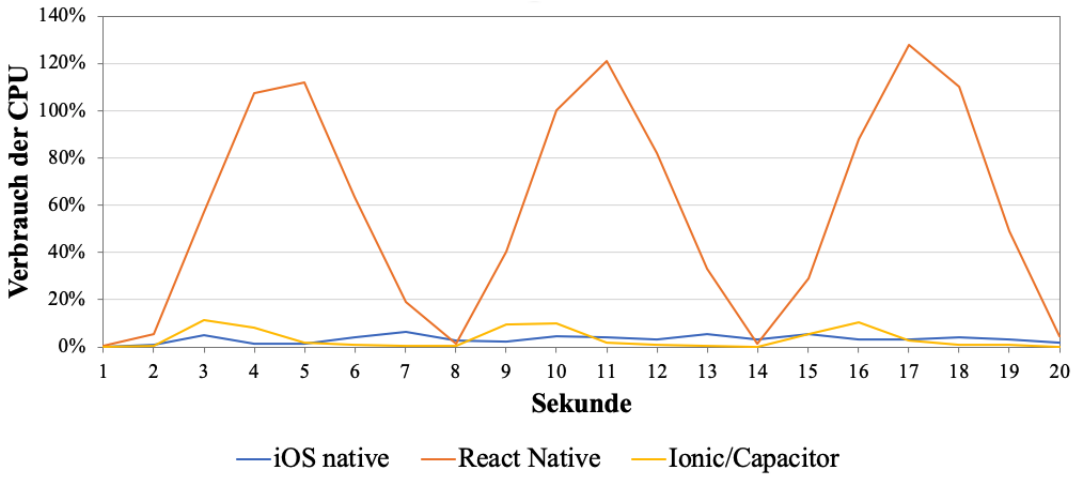

Abbildung 28: Durchschnittswerte aller drei Anwendungen beim Scrollen durch virtuelle Liste auf iOS

Bei diesem Interaktionsszenario trat der Fall auf, dass iOS nach einer Interaktion auf den idle Status der Anwendung wartete, bevor die nächste Geste ausgeführt werden konnte. Dies wurde bereits bei der Beschreibung des Testfalls in Kapitel 3.5.3 näher beschrieben. Das bedeutet, dass die Anwendungen einzeln betrachtet werden müssen.

Wie in Abbildung 28 ersichtlich, verbraucht die native iOS Anwendung konstant sehr wenig CPU für die Berechnung der Listeneinträge. Während der

gesamten Ausführung des Testfalls benötigte die Anwendung zwischen 1 und 7 % der CPU. Das Diagramm zeigt sechs leichte Spitzen, die vermutlich zuerst durch den Scroll und dann durch die Berechnung der Listeneinträge entstanden sind. Bei der Ionic/Capacitor Anwendung sind drei Nutzungsspitzen erkennbar. Diese befinden sich zwischen 10 und 11 %. In den Pausen zwischen den Scroll-Gesten sinkt der CPU-Verbrauch auf Null. Auch beim dritten Interaktionsszenario verbraucht die React Native Anwendung am meisten CPU. Dabei sind ebenfalls drei Nutzungsspitzen zu sehen. Alle drei benötigen über 100 %. Das bedeutet, dass die CPU-Last über einen Kern hinausgeht. Das Testgerät besitzt 6 Kerne. Es lässt sich erkennen, dass jede Scroll-Geste etwas mehr CPU benötigt als die vorherige. Die erste Scroll-Geste benötigt 112 %, die zweite 121 % und die dritte 128 %. Zwischen den Gesten sinkt der Verbrauch kurzzeitig auf Null.

4.2.2 Speicherverbrauch

Während der Ausführung der Testfälle wird der Verbrauch des Arbeitsspeichers in Mebibyte (MiB) gemessen. Die vertikale Achse zeigt den Verbrauch des Arbeitsspeichers in MiB der Testanwendung. Auf der horizontalen Ache ist die Zeit in Sekunden dargestellt.

Öffnen und Schließen des Navigation Drawers

Wie in Abbildung 29 ersichtlich, verbrauchen die native iOS und Ionic/Capacitor Anwendungen im Durchschnitt über die gesamten 20 Sekunden gleichmäßig viel Arbeitsspeicher. Beide Anwendungen verwenden beim Öffnen und Schließen des Navigation Drawers zwischen 11,5 bis 13,5 MiB Arbeitsspeicher. Die Spitzen nach jeder Geste sind kaum wahrnehmbar. Im Gegensatz dazu verbraucht die React Native Anwendung über das vierfache mehr Arbeitsspeicher für die gleichen Interaktionen. Die Durchschnittswerte sind hier bei etwa 50 bis

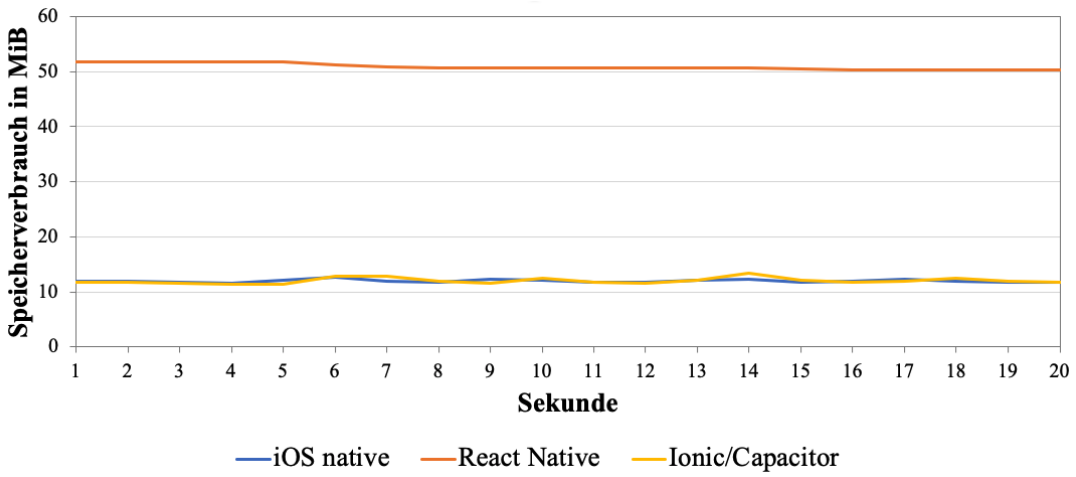

Abbildung 29: Durchschnittswerte aller drei Anwendungen beim Öffnen und Schließen des Navigation Drawers auf iOS

51,8 MiB. Der Verbrauch des Arbeitsspeichers in der React Native Anwendung blieb die gesamten 20 Sekunden ebenfalls relativ gleichmäßig.

Übergang zwischen zwei Bildschirmen

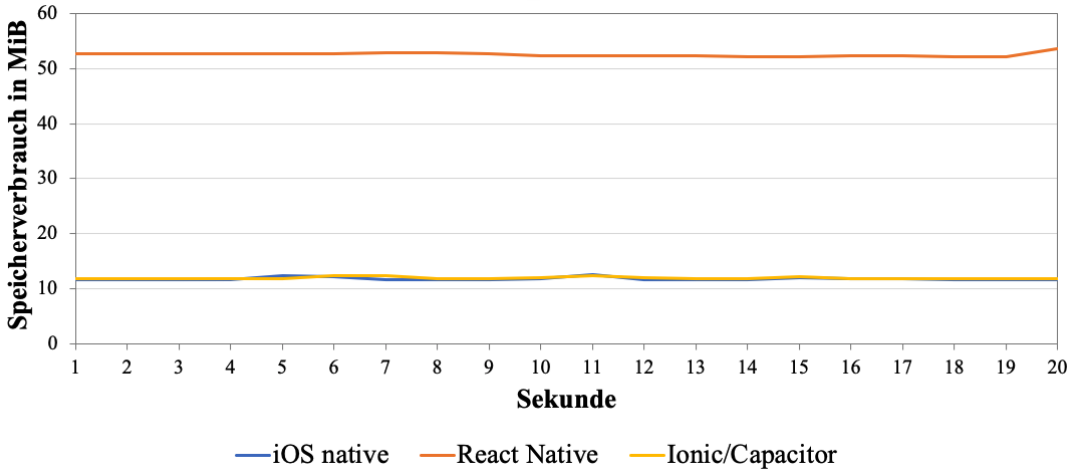

Abbildung 30: Durchschnittswerte aller vier Anwendungen beim Übergang zwischen zwei Bildschirme auf iOS

Abbildung 30 zeigt den Arbeitsspeicherverbrauch für den Übergang zwischen zwei Bildschirmen. Auch hier verhalten sich die native iOS Anwendung und

die Ionic/Capacitor Anwendung sehr ähnlich. Kurz nach den einzelnen Gesten sind minimale Anstiege zu sehen. Der Verbrauch beider Anwendungen liegt zwischen 11,6 und 12,5 MiB. Der Arbeitsspeicherverbrauch der React Native Anwendung liegt deutlich über dem der beiden anderen Anwendungen. Die Werte liegen zwischen 52,1 und 53,6 MiB. Dies entspricht in etwa dem 4,5 fachen Arbeitsspeicherverbrauch im Gegensatz zur nativen iOS und Ionic/Capacitor Anwendung.

Scrollen durch virtuelle Liste

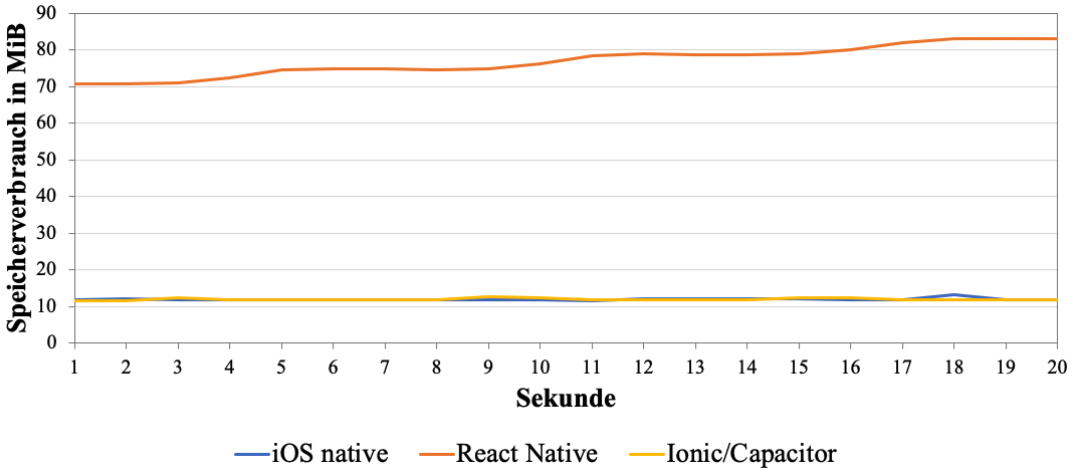

Abbildung 31: Durchschnittswerte aller vier Anwendungen beim Scrollen durch eine virtuelle Liste auf iOS

Wie in Abbildung 31 ersichtlich, zeigt auch das dritte Interaktionsszenario einen ähnlichen Arbeitsspeicherverbrauch bei der nativen iOS und der Ionic/Capacitor Anwendung. Der Verbrauch liegt, wieder ohne erwähnenswerte Spitzen, konstant zwischen 11,6 und 13 MiB. Auch bei diesem Interaktionsszenario verbraucht die React Native Anwendung deutlich mehr Arbeitsspeicher. Der Verbrauch beginnt bei 70,8 MiB und steigt nach jeder Ausführung der Scroll-Geste leicht an. Nach der Ausführung aller drei Scroll-Gesten ist der Verbrauch mit 83,1 MiB am höchsten. Verglichen mit den beiden anderen Anwendungen, entspricht dies in etwa dem siebenfachen Verbrauch.

5. Diskussion

In dieser Masterarbeit wurde eine Untersuchung der Performanzunterschiede von drei typischen UI-Interaktionsszenarien auf Grundlage nativer und von drei verschiedenen plattformübergreifenden Entwicklungsansätzen, sowohl auf einem Android- als auch einem iOS-Gerät, durchgeführt. Als Bewertungskriterien wurde die CPU- und Arbeitsspeichernutzung ausgewählt. Die Ergebnisse zeigen den Unterschied zwischen den nativen Implementierungen und den drei plattformübergreifenden Implementierungen auf den beiden größten mobilen Plattformen. Da es eine fast unüberschaubare Auswahl an plattformübergreifenden Entwicklungsansätzen gibt, erweist sich die Auswahl des geeigneten mobilen Ansatzes oft als schwierig. Wie eine Studie von de Andrade et al. (2015) zeigt, hat die Wahl des richtigen Entwicklungsansatzes durchaus Einfluss auf den Erfolg der zu entwickelnden Anwendung. In dieser Studie bemerkten 13,33 % der Benutzerinnen und Benutzer den Leistungsunterschied zwischen einer nativen und einer hybriden Anwendung. Als ein großer Vorteil wird immer wieder das Argument einer einzigen Codebasis für alle mobilen Plattformen genannt. Die Performanz einer Anwendung kann heutzutage jedenfalls nicht außer Acht gelassen werden. Es ist mittlerweile einfacher den je, auf eine Anwendung eines anderen Anbieters zu wechseln, wenn die Erwartungen der Benutzerinnen und Benutzer nicht erfüllt werden. Wenige Klicks und die Anwendung ist deinstalliert und womöglich ein Konkurrenzprodukt installiert. Die Auswahl des falschen Entwicklungsansatzes kann somit durchaus einen negativen Einfluss auf die Anzahl der aktiven Nutzerinnen und

Nutzer haben. Ein späterer Wechsel auf einen anderen Entwicklungsansatz um einer etwaigen rückläufigen Anzahl an Nutzerinnen und Nutzern entgegenzusteuern, ist mit einem großen Kosten- und Zeitaufwand verbunden. Aus diesem Grund sollen die Ergebnisse aus dieser Studie den Entwicklerinnen und Entwicklern mobiler Anwendungen eine Orientierungshilfe bei der Auswahl des geeigneten Entwicklungsansatzes bieten.

Die Ergebnisse dieser Studie zeigen, dass die React Native Anwendung unter iOS bei allen drei getesteten UI-Interaktionsszenarien deutlich mehr CPU sowie Arbeitsspeicher verbraucht. Am größten ist der Unterschied jedoch bei der Interaktion *Scrollen durch eine virtuelle Liste*, wo unter iOS die React Native Anwendung Nutzungsspitzen zwischen 112 und 128 % und die native Anwendung und Ionic/Capacitor immer unter 15 % CPU-Nutzung blieben. Der CPU-Verbrauch bleibt sowohl bei der nativen iOS Anwendung als auch der Ionic/Capacitor Anwendung in allen drei getesteten UI-Interaktionsszenarien unter 15 %. Auch der Arbeitsspeicherverbrauch ist bei beiden Impementierungen ähnlich und beträgt zwischen 11 und 13 MiB. Die React Native Anwendung hingegen, weist einen deutlich höheren Arbeitsspeicherverbrauch auf. Dieser liegt beim *Öffnen und Schließen des Navigation Drawers* und beim *Übergang zwischen zwei Bildschirmen* bei knapp über 50 MiB und beim *Scrollen durch eine virtuelle Liste* steigt dieser stetig von 70 MiB auf 83 MiB an.

Unter Android ist der Unterschied zwischen den drei plattformübergreifend programmierten Anwendungen im Hinblick auf den CPU-Verbrauch nicht so gravierend. Doch auch hier ist zu erkennen, dass die React Native Anwendung bei den Interaktionen *Öffnen und Schließen des Navigation Drawers* und *Übergang zwischen zwei Bildschirmen* am meisten CPU benötigt. Bei der Interaktion *Scrollen durch eine virtuelle Liste*, wo drei von unten nach oben verlaufende Scroll-Gesten durchgeführt wurden, schneidet die React Native Anwendung bei der zweiten und dritten Scroll-Geste dafür etwas besser ab, als die Ionic/Capacitor Anwendung. Zu den gleichen Ergebnissen kamen auch Huber et al.

(2020) in ihrer Studie.

Der Arbeitsspeicherverbrauch fällt unter Android sehr unterschiedlich aus und muss von Interaktion zu Interaktion betrachtet werden. Bei *Öffnen und Schließen des Navigation Drawers* steigt der Verbrauch des Arbeitsspeichers der nativen Android Anwendung nur minimal an. Bei React Native und Flutter bleiben die Spitzen unter einem Anstieg von 10 %. Auch bei der Ionic/Capacitor Anwendung bleibt der Zuwachst bei der Nutzungsspitze unter 10 %.

Der Arbeitsspeicherverbrauch bei der Interaktion *Übergang zwischen zwei Bildschirmen* zeigt, dass die native Implementierung einen fast vernachlässigbaren Verbrauch aufweist, wobei der höchste Wert bei 2,8 % liegt. Etwas mehr verbraucht die React Native Anwendung, wobei der Zuwachs an den beiden Spitzen unter 5 % bleibt. An dritter Stelle befindet sich die Flutter Anwendung, welche einen weiteren fünf prozentigen Anstieg zeigt. Die Ionic/Capacitor Anwendung verbraucht jedoch deutlich mehr Arbeitsspeicher bei der Ausführung der Touch-Geste, als alle anderen Implementierungen. Während der Ausführung der Gesten steigt der Bedarf auf 15 bis 20 % an.

Beim *Scrollen durch eine virtuelle Liste* verbrauchen die native Android Anwendung sowie die Ionic/Capacitor Anwendung relativ wenig Speicherplatz, wobei die Ionic/Capacitor Anwendung sogar bis auf eine Nutzungsspitze weniger oder gleich viel Arbeitsspeicher verbraucht, wie die native Implementierung. Die Flutter Implementierung zeigt bei den ersten beiden Scroll-Gesten einen deutlichen Anstieg im Speicherverbrauch. Bei der dritten Scroll-Geste sinkt der Speicherbedarf jedoch schnell ab. Während der Ausführung des gesamten Testfalls weist die React Native Implementierung einen stetigen Anstieg im Speicherverbrauch, welcher deutlich höher ist als die native und Ionic/Capacitor Anwendung, auf. Den ansteigenden Speicherverbrauch bei React Native beim *Scrollen durch eine virtuelle Liste* bei der Benutzerinteraktion zeigte bereits die Studie von Huber and Demetz (2019). Im Gegensatz zur Studie von Huber and Demetz (2019) verbrauchte die für diese Masterarbeit implementierte

Ionic/Capacitor Anwendung weniger Arbeitsspeicher, als die React Native Implementierung. An dieser Stelle muss jedoch angemerkt werden, dass Huber and Demetz (2019) eine Ionic/Cordova Anwendung getestet haben. Obwohl beide Anwendungen auf Ionic basieren, ist der Unterschied zwischen dem älteren (Release 2009) Cordova und dem weitaus morderneren (Release 2018) Capacitor auch deutlich im Ressourcenverbrauch zu sehen. Ein möglicher Grund hierfür ist, dass Capacitor weitaus modernere APIs nutzt, welche 2009 schlichtweg nicht existierten. Somit ist eine Capacitor Anwendung deutlich näher an einer reinen nativen Anwendung, was zu einer besseren Performanz führen kann.

In Bezug auf die in dieser Masterarbeit durchgeführten Studien sind eine Reihe von Vorbehalten zu beachten. Das Testen der Performanz erfolgte auf beiden Plattformen mit unterschiedlichen Tools, womit die Ergebnisse von Android nicht direkt mit den Ergebnissen von iOS vergleichbar sind. Außerdem wurden die Testfälle jeweils auf einem einzigen Smartphone pro Plattform durchgeführt. Es gibt jedoch eine große Vielfalt bei Android-Geräten auf dem globalen Markt und der Unterschied zwischen einem Low-End- und einem High-End-Gerät ist sehr groß. Der Ressourcenverbrauch sollte unter der Verwendung mehrerer Smartphones aus dem großen Spektrum von High-End- bis Low-End-Android und iOS-Smartphones erfolgen. Wie die Forschungsergebnisse von Huber et al. (2020) und Willocx et al. (2016) zeigen, können hier unter Umständen große Unterschiede sichtbar werden. Weiters wurden nur drei der fünf plattformübergreifenden Entwicklungsansätze nach Biørn-Hansen et al. (2018) betrachtet. Der Modell-getriebene Ansatz sowie PWAs wurden nicht miteinbezogen. Obwohl die drei getesteten UI-Interaktionsszenarien in Anwendungen häufig verwendet werden, sollten weitere Gesten in Betracht gezogen werden.

Diese Studie bestätigt die Ergebnisse früherer Studien und zeigt, dass plattformübergreifende Entwicklungsansätze mehr Ressourcen verbrauchen als Anwendungen, welche mit dem nativen Entwicklungsansatz entwickelt wurden.

Der aktuelle Literaturbestand wird um die Messung des Ressourcenverbrauchs von drei häufig eingesetzten UI-Interaktionsszenarien mit drei unterschiedlichen Entwicklungsansätzen implementierten Anwendungen auf der iOS Plattform erweitert.

6. Fazit

In dieser Masterarbeit wurde eine Performanzanalyse von drei typischen UI-Interaktionen sowohl auf einem Android- als auch einem iOS-Gerät durchgeführt. Zu diesen zählen: *Öffnen und Schließen des Navigation Drawers*, *Übergang zwischen zwei Bildschirmen* sowie *Scrollen durch eine virtuelle Liste*. Dafür wurden fünf, mit unterschiedlichen Frameworks umgesetzten, Implementierungen der gleichen Anwendung verwendet. Die drei plattformübergreifenden Anwendungen, wurden mit React Native, Flutter und Ionic/Capacitor implementiert. Zusätzlich wurden beide Anwendungen noch nativ für Android- und iOS implementiert.

Die in Sektion 1.2 vorgestellten Forschungsfragen können folgendermaßen beantwortet werden:

Forschungsfrage 1: *Wie kann die Performanz von plattformübergreifenden Entwicklungsansätzen einheitlich gemessen werden?*

Der Ressourcenverbrauch kann pro Plattform einheitlich gemessen werden. Für Anwendungen, welche auf Android basieren, stehen unterschiedliche Programme von Drittanbietern zur Verfügung. Für die Durchführung der Testszenarien auf Android wurde in dieser Masterarbeit das Tool vmstat verwendet. Da Apple bis dato keine Drittanbietersoftware zulässt und bei iOS Xcode zum Einsatz kam, bedeutet dies, dass mit dieser Entwicklungsumgebung sowohl programmiert als auch getestet wurde.

Es wurde während der Recherche keine Möglichkeit gefunden, mit demselben Framework oder Tool, plattformübergreifende Anwendungen sowohl auf einem Android- als auch auf einem iOS-Gerät zu testen.

Forschungsfrage 2: *Wie unterscheidet sich die Performanz von plattformübergreifenden Entwicklungsansätzen unter der Nutzung typischer UI-Interaktionen?*

Es wurde festgestellt, dass plattformübergreifend programmierte Anwendungen durchwegs am meisten die CPU in Anspruch nahmen. Wobei die React Native Anwendung in allen drei Testszenarien am meisten CPU beanspruchte. Auf dem Android-Gerät, verbrauchten die Flutter und Ionic/Capacitor Anwendungen ähnlich viel CPU. Auf dem iOS-Gerät verbrauchte die Ionic/Capacitor Anwendung hingegen ähnlich viel CPU, wie die native Anwendung. Die Flutter Anwendung konnte auf iOS nicht getestet werden.

Der Arbeitsspeicherverbrauch war auf dem Android-Gerät sehr unterschiedlich. Bei den Interaktionsszenarien *Öffnen und Schließen des Navigation Drawers* und *Scrollen durch eine virtuelle Liste* beanspruchte die Ionic/Capacitor Anwendung weniger Arbeitsspeicher, als die beiden anderen plattformübergreifenden Ansätze. Beim *Scrollen durch eine virtuelle Liste* war Ionic/Capacitor bis auf eine Nutzungsspitze sogar effizienter, als die native Android Anwendung. Beim *Übergang zwischen zwei Bildschirmen* hingegen, verbraucht die Ionic/Capacitor Implementierung von allen vier Anwendungen mit Abstand am meisten Arbeitsspeicher. Auf dem iOS-Gerät verbrauchte die React Native Anwendung mit Abstand am meisten Arbeitsspeicher. Ionic/Capacitor und die native Implementierung haben einen sehr ähnlichen, konstant niedrigen Speicherverbrauch.

7. Ausblick

Ohne Zweifel ist eine Anwendung, welche nativ entwickelt wurde, nach wie vor die am ressourcenschonendste Variante. Der Trend der letzten Jahre zeigt jedoch, dass plattformübergreifende Entwicklungsansätze im Gegensatz zu nativen, immer größeren Einfluss nehmen und in Zukunft vermutlich mehr Anwendungen auf diese Art entwickelt werden. Die Vorteile, vor allem die Reduzierung der Kosten der Entwicklung, einer einzelnen Codebasis, überwiegen einen möglichen Mehrverbrauch an CPU oder Arbeitsspeicher. Moderne Smartphones bieten ohnedies schon weitaus mehr Leistung, als der Großteil der Nutzerinnen und Nutzer tatsächlich benötigt, weshalb eine leichte Mehrbelastung der Hardware oft nicht ins Gewicht fällt. Somit werden plattformübergreifende Entwicklungsansätze auch in Zukunft eine große Bedeutung haben und neue Technologien hinzukommen. Zukünftige Arbeiten in diesem Bereich werden sich somit mit den entsprechend aktuellen Ansätzen auseinandersetzen können. Die Art und Weise, wie die Nutzer mit ihren Smartphones interagieren, wird sich in absehbarer Zeit nicht gravierend verändern. Daher können in zukünftigen Arbeiten zumindest die UI-Interaktionen, welche in dieser Masterarbeit getestet werden, übernommen und auf neuen Technologien getestet werden. Dieses Forschungsfeld bietet auf Grund der Aktualität und Relevanz somit viel Potential für zukünftige Forschungen.

Diese Masterarbeit betrachtete drei unterschiedliche plattformübergreifenden Entwicklungsansätze, darunter den interpretierten, hybriden und cross-kompilierten Ansatz. Aus allen drei Entwicklungsansätzen wurde jeweils ein Fra-

mework ausgesucht, welches aktuell am beliebtesten war. Der modell-getriebene Entwicklungsansatz befindet sich zur Zeit noch im experimentellen Status und wird fast nur im wissenschaftlichen Umfeld eingesetzt. Je nachdem wie die Entwicklung fortschreitet, sollte auch dieser in zukünftigen Arbeiten in Betracht gezogen werden.

Die von Google entwickelten und seit fünf Jahren eingesetzten PWAs, unterstützte Apple mit seinem Safari-Browser nicht von Anfang an. Apple sieht PWAs als Konkurrenz, da große Teile der Einnahmen von Apple über den eigenen App Store generiert werden. Bei einem Einsatz von PWAs würde auch noch ein Verlust der Nutzerdaten, welche Apple erhält wenn die Benutzerinnen und Benutzer Anwendungen über den App Store herunterladen, wegfallen. Doch langsam scheint Apple die Technologie, wenn auch verspätet und nur in kleinen Schritten, auch auf iOS zuzulassen. Sobald PWAs auch auf iOS vollinhaltlich unterstützt werden, würde dies ebenso ein interessantes Forschungsfeld öffnen. Hier wäre es in Zukunft von Interesse, UI-Interaktionen zu testen um zu sehen, ob die in den Medien gemachten Versprechen, dass PWAs nativen Anwendungen ebenbürtig sind, tatsächlich eingehalten werden.

Literaturverzeichnis

Adinugroho, T. Y., Reina, and Gautama, J. B. (2015). Review of Multi-platform Mobile Application Development Using WebView: Learning Management System on Mobile Platform. *Procedia Computer Science*, 59:291–297.

Ajayi, O. O., Omotayo, A. A., Orogun, A. O., Omomule, T. G., and Orimoloye, S. M. (2018). Performance Evaluation of Native and Hybrid Android Applications. *Performance Evaluation*, 7(16).

Amatya, S. and Kurti, A. (2013). Cross-Platform Mobile Development: Challenges and Opportunities. In *International Conference on ICT Innovations*, volume Advances in Intelligent Systems and Computing, vol 231, pages 219–229. Springer, Heidelberg.

Appfigures and VentureBeat (2020a). Biggest app stores in the world 2020. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/. Abgerufen am 2020-10-28.

Appfigures and VentureBeat (2020b). Number of apps in leading app stores. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/. Abgerufen am 2020-08-23.

Apple (2013). About the Virtual Memory System. https://developer.apple.com/library/archive/documentation/Performance/Conceptual/ManagingMemory/Articles/AboutMemory.html. Abgerufen am 2020-10-22.

Apple (2016). Energy Efficiency Guide for iOS Apps: Fundamental Concepts. https://developer.apple.com/library/archive/documentation/Performance/Conceptual/EnergyGuide-iOS/FundamentalConcepts.html. Abgerufen am 2020-10-27.

Behrens, H. (2010a). MobileTechCon 2010 - Heiko Behrens about Cross-Platform Mobile Development.

Behrens, H. (2010b). Plattformübergreifende App-Entwicklung (ein Vergleich) - MobileTechCon 2010.

Biørn-Hansen, A. and Ghinea, G. (2018). Bridging the Gap: Investigating Device-Feature Exposure in Cross-Platform Development. In *Proceedings of the 51st Hawaii International Conference on System Sciences*. Hawaii International Conference on System Sciences.

Biørn-Hansen, A., Grønli, T.-M., and Ghinea, G. (2018). A Survey and Taxonomy of Core Concepts and Research Challenges in Cross-Platform Mobile Development. *ACM Computing Surveys (CSUR)*, 51(5):108.

Biørn-Hansen, A., Majchrzak, T. A., and Grønli, T.-M. (2017). Progressive Web Apps: The Possible Web-native Unifier for Mobile Development. In *Proceedings of the 13th International Conference on Web Information Systems and Technologies*, pages 344–351, Porto, Portugal. SCITEPRESS - Science and Technology Publications.

Ciman, M. and Gaggi, O. (2017). An empirical analysis of energy consumption of cross-platform frameworks for mobile development. *Pervasive and Mobile Computing*, 39:214–230.

Corral, L., Janes, A., and Remencius, T. (2012). Potential Advantages and Disadvantages of Multiplatform Development Frameworks–A Vision on Mobile Environments. *Procedia Computer Science*, 10:1202–1207.

Dalmasso, I., Datta, S. K., Bonnet, C., and Nikaein, N. (2013). Survey, comparison and evaluation of cross platform mobile application development tools. In *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*. IEEE.

de Andrade, P. R. M., B.Albuquerque, A., Frota, O. F., Silveira, R. V., and da Silva, F. A. (2015). Cross Platform App : A Comparative Study. *International Journal of Computer Science and Information Technology*, 7(1):33–40.

Delia, L., Galdamez, N., Thomas, P., Corbalan, L., and Pesado, P. (2015). Multiplatform mobile application development analysis. In *2015 IEEE 9th International Conference on Research Challenges in Information Science (RCIS)*, pages 181–186, Athens, Greece. IEEE.

El-Kassas, W. S., Abdullah, B. A., Yousef, A. H., and Wahba, A. M. (2017). Taxonomy of Cross-Platform Mobile Applications Development Approaches. *Ain Shams Engineering Journal*, 8(2):163–190.

Fischer, W. (2020). Linux Performance Messungen mit vmstat – Thomas-Krenn-Wiki. https://www.thomas-krenn.com/de/wiki/Linux_Performance_Messungen_mit_vmstat. Abgerufen am 2020-10-22.

Gaunt, M. (2019). Service Workers: An Introduction | Web Fundamentals. https://developers.google.com/web/fundamentals/primers/service-workers?hl=de. Abgerufen am 2020-05-18.

Gok, N. and Khanna, N. (2013). *Building Hybrid Android Apps with Java and JavaScript*. O'Reilly.

Google, I. (2020a). Gestures. https://material.io/design/interaction/gestures.html#types-of-gestures. Abgerufen am 2020-07-09.

Google, I. (2020b). Understanding navigation. https://material.io/design/navigation/understanding-navigation.html#lateral-navigation. Abgerufen am 2020-08-27.

Gorschek, T., Tempero, E., and Angelis, L. (2014). On the use of software design models in software development practice: An empirical investigation. *Journal of Systems and Software*, 95:176–193.

Hall, S. P. and Anderson, E. (2009). Operating systems for mobile computing. page 8.

Heitkötter, H., Hanschke, S., and Majchrzak, T. A. (2012). Comparing Cross-Platform Development Approaches for Mobile Applications. In *Proceedings of the 8th International Conference on Web Information Systems and Technologies*, pages 299–311, Porto, Portugal. SciTePress - Science and and Technology Publications.

Heitkötter, H. and Majchrzak, T. A. (2013). Cross-Platform Development of Business Apps with MD2. In *Design Science at the Intersection of Physical and Virtual Design*, volume 7939, pages 405–411, Berlin, Heidelberg. Springer Berlin Heidelberg.

Heitkötter, H., Majchrzak, T. A., and Kuchen, H. (2013). Cross-platform model-driven development of mobile applications with md [2]. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing - SAC '13*, page 526, Coimbra, Portugal. ACM Press.

Hesenius, M., Griebe, T., Gries, S., and Gruhn, V. (2014). Automating UI tests for mobile applications with formal gesture descriptions. In *Proceedings of the 16th International Conference on Human-Computer Interaction with Mobile Devices & Services - MobileHCI '14*, pages 213–222, Toronto, ON, Canada. ACM Press.

Huber, S. and Demetz, L. (2019). Performance Analysis of Mobile Cross-platform Development Approaches based on Typical UI Interactions. In *Proceedings of the 14th International Conference on Software Technologies*, pages 40–48, Prague, Czech Republic. SCITEPRESS - Science and Technology Publications.

Huber, S., Demetz, L., and Felderer, M. (2020). Analysing the Performance of Mobile Cross-platform Development Approaches using UI Interaction Scenarios. In *Software Technologies*, volume Communications in Computer and Information Science, vol 1250. Springer International Publishing.

iResearch (2019). Mobile app revenues 2014-2023. https://www.statista.com/statistics/269025/worldwide-mobile-app-revenue-forecast/. Abgerufen am 2020-10-28.

Jia, X., Tan, Y., and Ebone, A. (2018). A Performance Evaluation of Cross-Platform Mobile Application Development Approaches. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 92–93.

Latif, M., Lakhrissi, Y., Nfaoui, E. H., and Es-Sbai, N. (2016). Cross platform approach for mobile application development: A survey. In *2016 International Conference on Information Technology for Organizations Development (IT4OD)*, pages 1–5, Fez, Morocco. IEEE.

Le Goaer, O. and Waltham, S. (2013). Yet another DSL for cross-platforms mobile development. In *Proceedings of the First Workshop on the Globalization of Domain Specific Languages - GlobalDSL '13*, pages 28–33, Montpellier, France. ACM Press.

LePage, P. (2017). Ihre erste Progressive Web App | Web Fundamentals. https://developers.google.com/web/fundamentals/codelabs/your-first-pwapp?hl=de. Abgerufen am 2020-05-18.

Majchrzak, T. A., Biørn-Hansen, A., and Grønli, T.-M. (2017). Comprehensive Analysis of Innovative Cross-Platform App Development Frameworks. In *Proceedings of the 50th Hawaii International Conference on System Sciences*. Hawaii International Conference on System Sciences.

Malavolta, I., Procaccianti, G., Noorland, P., and Vukmirovic, P. (2017). Assessing the Impact of Service Workers on the Energy Efficiency of Progressive

Web Apps. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 35–45, Buenos Aires, Argentina. IEEE.

Mercado, I. T., Munaiah, N., and Meneely, A. (2016). The impact of cross-platform development approaches for mobile applications from the user's perspective. In *Proceedings of the International Workshop on App Market Analytics*, WAMA 2016, pages 43–49, Seattle, WA, USA. Association for Computing Machinery.

Milano, D. T. (2020). Dtmilano/AndroidViewClient. https://github.com/dtmilano/AndroidViewClient. Abgerufen am 2020-10-08.

Mohamed, L. and Abdelmounaïm, A. (2017). Decision Framework for Mobile Development Methods. *International Journal of Advanced Computer Science and Applications*, 8(2).

Newzoo (2019). Smartphone users 2020. https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/. Abgerufen am 2020-10-28.

Nunkesser, R. (2018). Beyond Web/Native/Hybrid: A New Taxonomy for Mobile App Development. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 214–218.

OHA (2007). Alliance FAQ | Open Handset Alliance. https://www.openhandsetalliance.com/oha_faq.html. Abgerufen am 2020-06-06.

Ohrt, J. and Turau, V. (2012). Cross-platform development tools for smartphone applications. In *IEEE Computer*, volume 9, pages 72–79. IEEE.

Osmani, A. (2019a). The App Shell Model | Web Fundamentals | Google Developers. https://developers.google.com/web/fundamentals/architecture/app-shell. Abgerufen am 2020-05-20.

Osmani, A. (2019b). Getting Started with Progressive Web Apps. https://developers.google.com/web/updates/2015/12/getting-started-pwa?hl=ar. Abgerufen am 2020-05-20.

Perchat, J., Desertot, M., and Lecomte, S. (2013). Component based Framework to Create Mobile Cross-platform Applications. *Procedia Computer Science*, 19:1004–1011.

Raj, C. R. and Tolety, S. B. (2012). A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach. In *2012 Annual IEEE India Conference (INDICON)*, pages 625–629. IEEE.

Ribeiro, A. and da Silva, A. R. (2012). Survey on Cross-Platforms and Languages for Mobile Apps. In *2012 Eighth International Conference on the Quality of Information and Communications Technology*, pages 255–260, Lisbon, TBD, Portugal. IEEE.

Richard, S. and LePage, P. (2020). What makes a good Progressive Web App? https://web.dev/pwa-checklist/. Abgerufen am 2020-05-20.

Rieger, C. and Majchrzak, T. A. (2016). Weighted Evaluation Framework for Cross-Platform App Development Approaches. In *Information Systems: Development, Research, Applications, Education*, volume 264, pages 18–39. Springer International Publishing, Cham.

Russell, A. (2015). Progressive Web Apps: Escaping Tabs Without Losing Our Soul. https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/. Abgerufen am 2020-05-18.

Smutny, P. (2012). Mobile development tools and cross-platform solutions. In *Proceedings of the 13th International Carpathian Control Conference (ICCC)*, pages 653–656, High Tatras, Slovakia. IEEE.

StatCounter (2020). Marktanteile der führenden mobilen Betriebssysteme an der Internetnutzung mit Mobiltelefonen weltweit von September 2009 bis März 2020. https://de.statista.com/statistik/daten/studie/184335/umfrage/marktanteil-der-mobilen-betriebssysteme-weltweit-seit-2009/. Abgerufen am 2020-06-06.

Tower, S. and TechCrunch (2020). Annual mobile app downloads worldwide by store 2024. https://www.statista.com/statistics/1010716/apple-app-store-google-play-app-downloads-forecast/. Abgerufen am 2020-10-28.

Umuhoza, E., Ed-douibi, H., Brambilla, M., Cabot, J., and Bongio, A. (2015). Automatic code generation for cross-platform, multi-device mobile apps: Some reflections from an industrial experience. In *Proceedings of the 3rd International Workshop on Mobile Development Lifecycle - MobileDeLi 2015*, pages 37–44, Pittsburgh, PA, USA. ACM Press.

Usman, M., Iqbal, M. Z., and Khan, M. U. (2017). A product-line model-driven engineering approach for generating feature-based mobile applications. *Journal of Systems and Software*, 123:1–32.

Vollmer, G. (2017). *Mobile App Engineering : Eine systematische Einführung – von den Requirements zum Go Live*, volume 1. Auflage. dpunkt.verlag, Heidelberg.

Vranić, V. and Staraček, Ľ. (2014). MDA Based Multiplatform Mobile Application Modeling with Platform Compliant User Interfaces. *INFOCOMP Journal of Computer Science*, 13(2):34–43.

Warner, J. (2018). Thank you for 100 million repositories. https://github.blog/2018-11-08-100m-repos/. Abgerufen am 2020-08-23.

Wilander, J. (2020). Full Third-Party Cookie Blocking and More. https://webkit.org/blog/10218/

full-third-party-cookie-blocking-and-more/. Abgerufen am 2020-11-01.

Willocx, M., Vossaert, J., and Naessens, V. (2015). A Quantitative Assessment of Performance in Mobile App Development Tools. In *2015 IEEE International Conference on Mobile Services*, pages 454–461, New York City, NY, USA. IEEE.

Willocx, M., Vossaert, J., and Naessens, V. (2016). Comparing performance parameters of mobile app development strategies. In *Proceedings of the International Workshop on Mobile Software Engineering and Systems - MOBILESoft '16*, pages 38–47, Austin, Texas. ACM Press.

Xanthopoulos, S. and Xinogalos, S. (2013). A comparative analysis of cross-platform development approaches for mobile applications. In *Proceedings of the 6th Balkan Conference in Informatics on - BCI '13*, page 213, Thessaloniki, Greece. ACM Press.

A. Ergebnisse CPU-Verbrauch pro Anwendung und Testfall

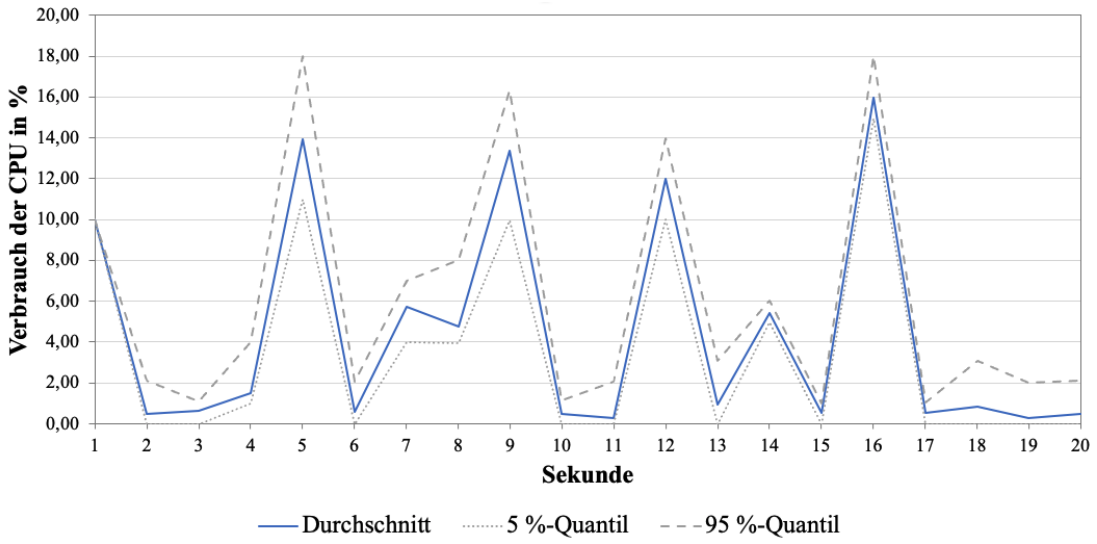

Abbildung 32: CPU - Android Native - Interaktion: Öffnen und Schließen des Navigation Drawers

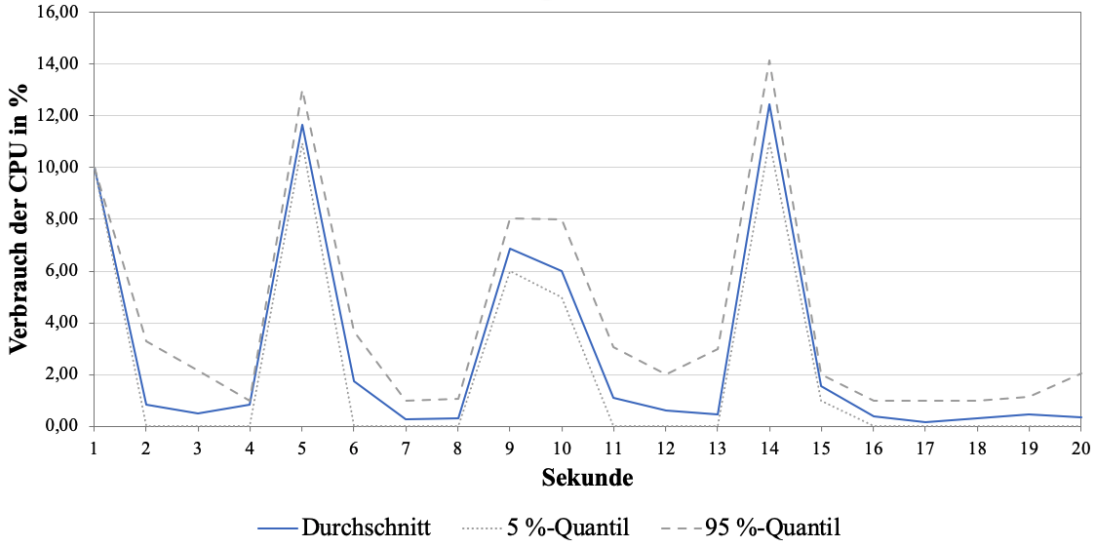

Abbildung 33: CPU - Android Native - Interaktion: Übergang zwischen zwei Bildschirmen

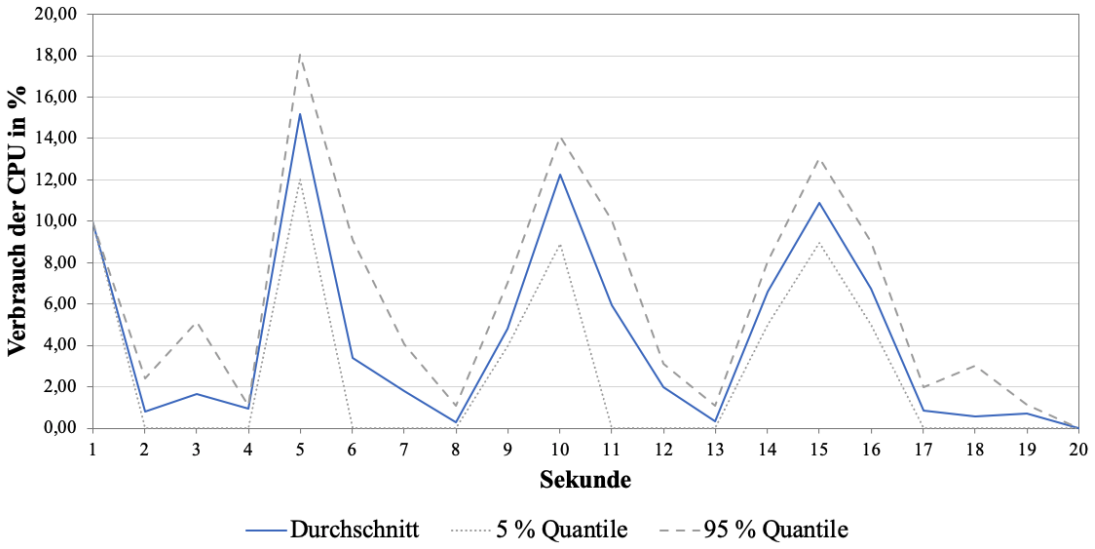

Abbildung 34: CPU - Android Native - Interaktion: Scrollen durch eine virtuelle Liste

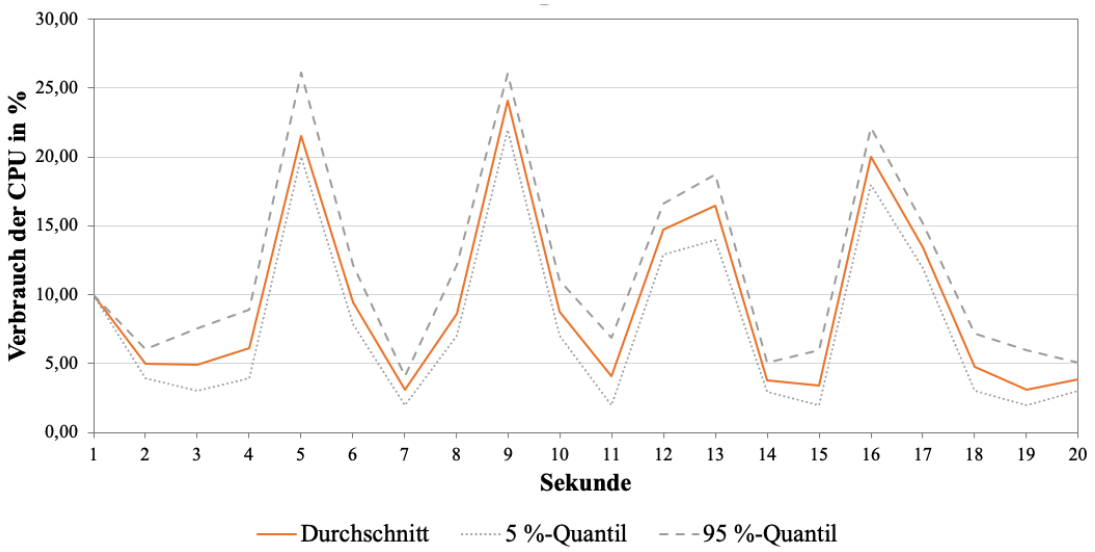

Abbildung 35: CPU - Android - React Native - Interaktion: Öffnen und Schließen des Navigation Drawers

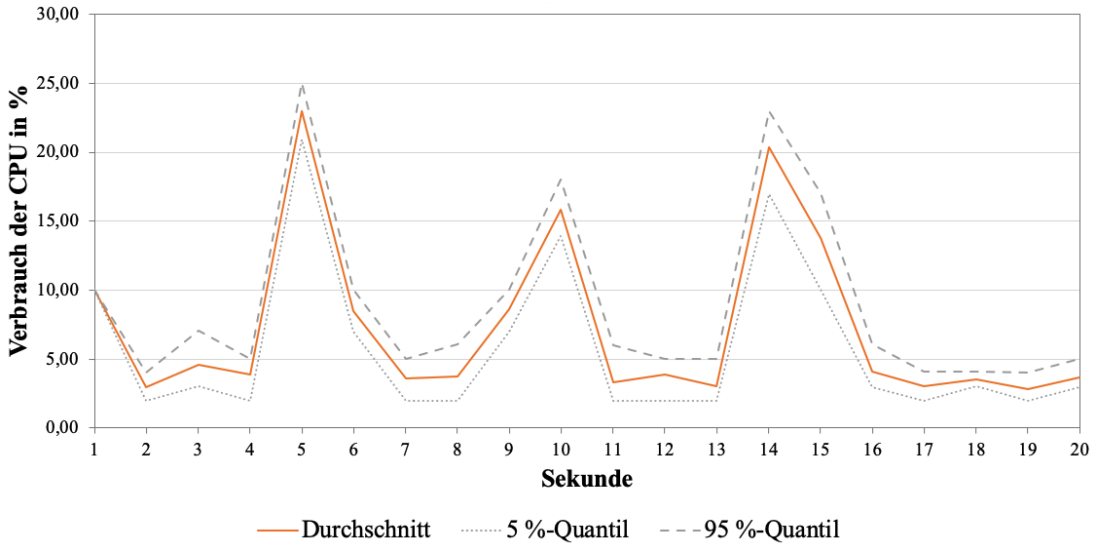

Abbildung 36: CPU - Android - React Native - Interaktion: Übergang zwischen zwei Bildschirmen

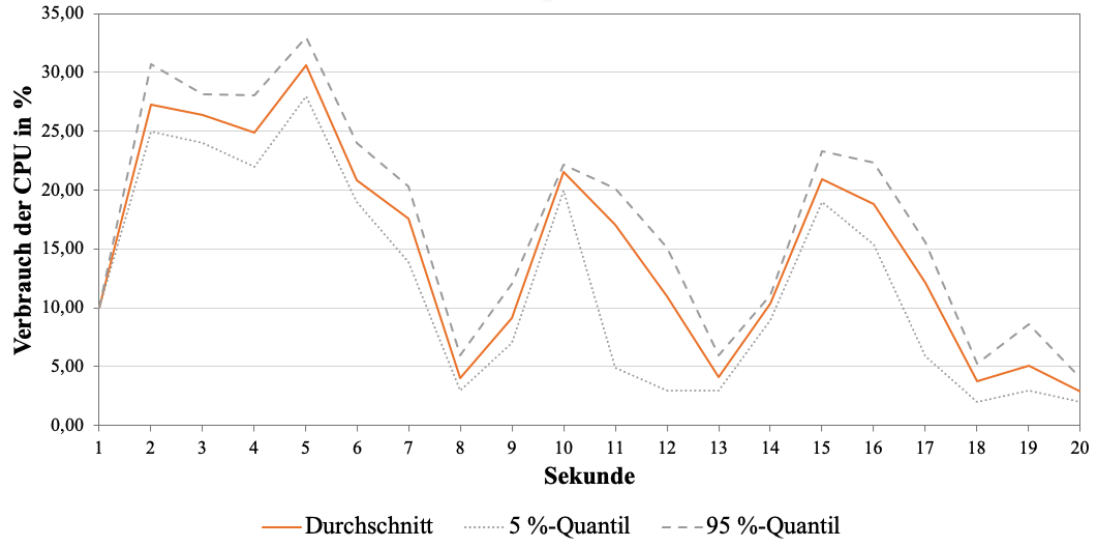

Abbildung 37: CPU - Android - React Native - Interaktion: Scrollen durch virtuelle Liste

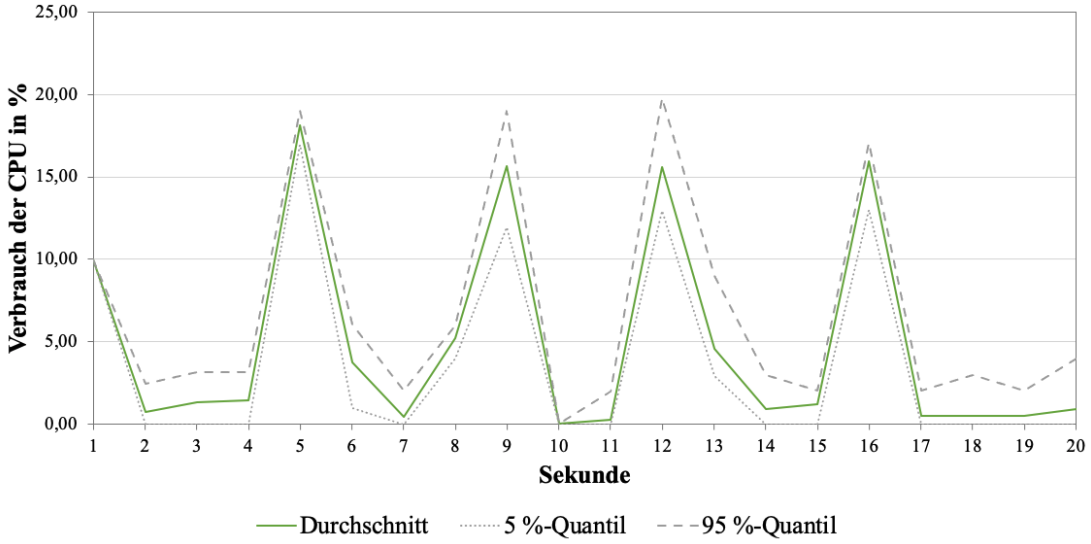

Abbildung 38: CPU - Android - Flutter - Interaktion: Öffnen und Schließen des Navigation Drawers

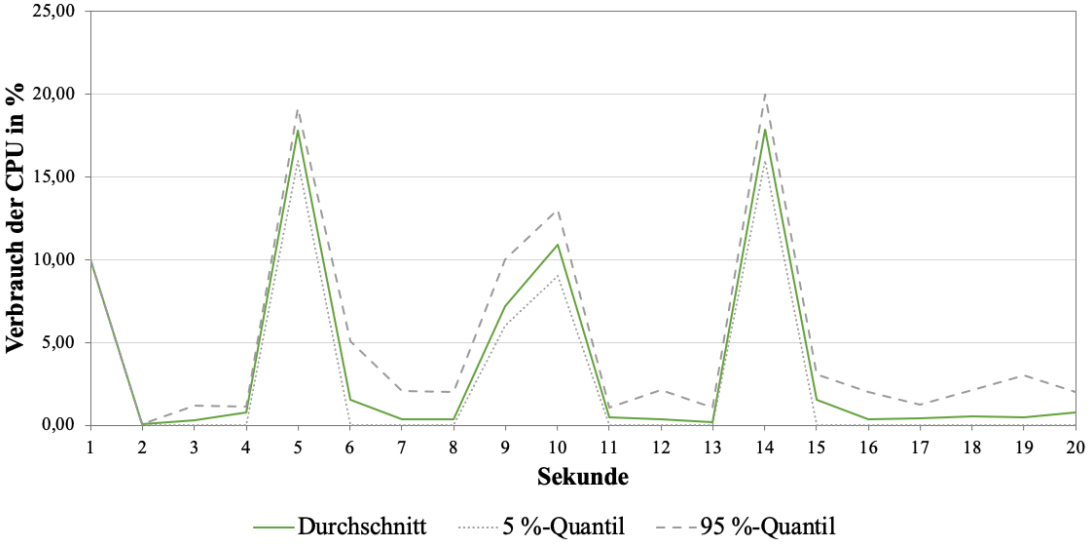

Abbildung 39: CPU - Android - Flutter - Interaktion: Übergang zwischen zwei Bildschirmen

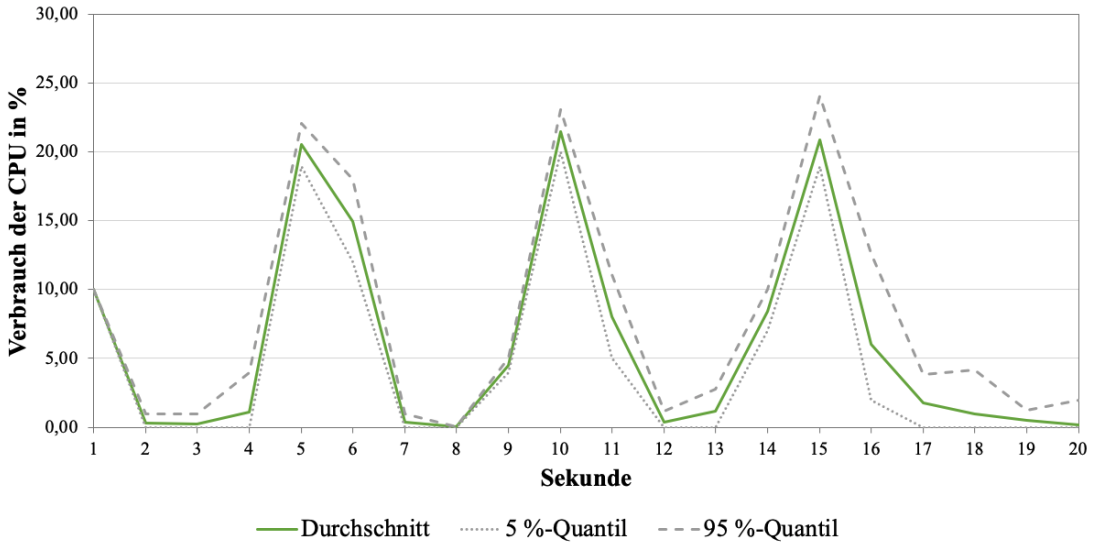

Abbildung 40: CPU - Android - Flutter - Interaktion: Scrollen durch virtuelle Liste

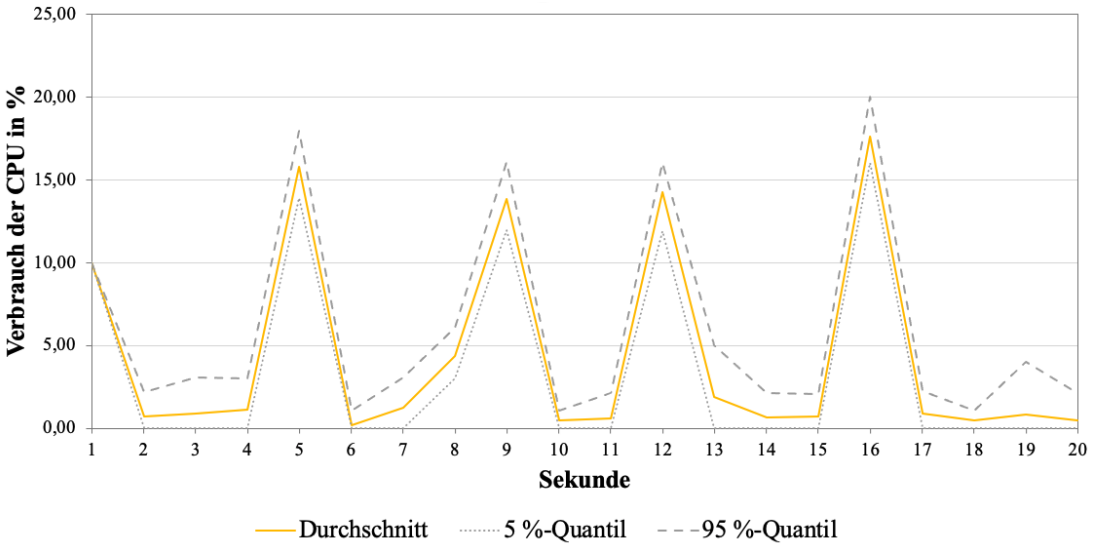

Abbildung 41: CPU - Android - Ionic/Capacitor - Interaktion: Öffnen und Schließen des Navigation Drawers

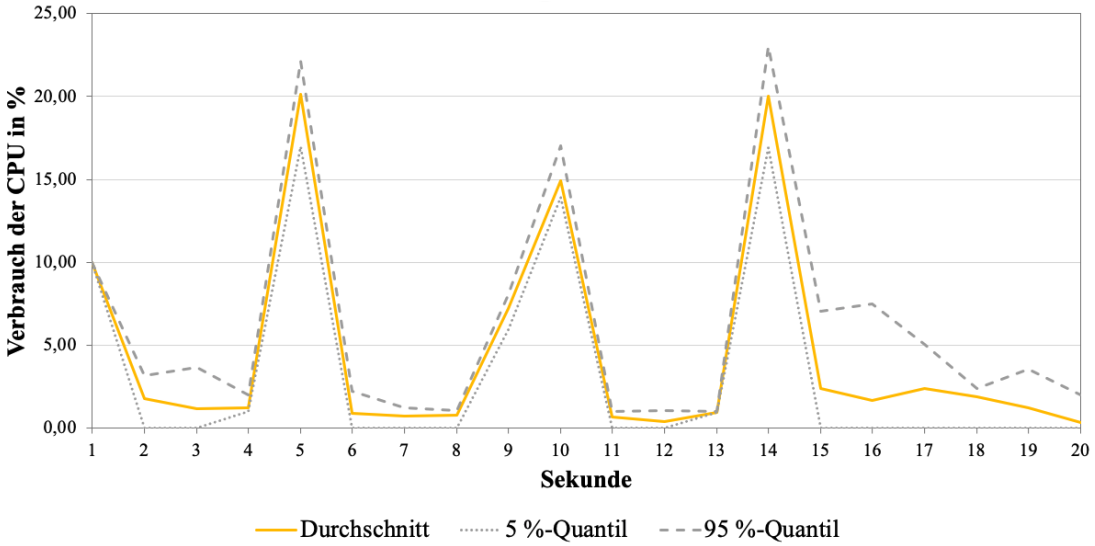

Abbildung 42: CPU - Android - Ionic/Capacitor - Interaktion: Übergang zwischen zwei Bildschirmen

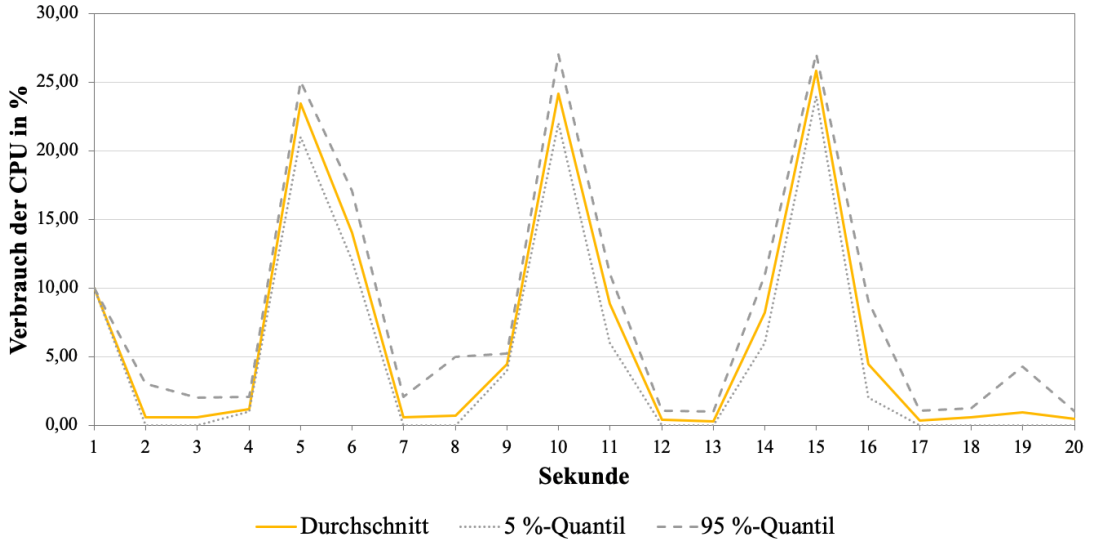

Abbildung 43: CPU - Android - Ionic/Capacitor - Interaktion: Scrollen durch virtuelle Liste

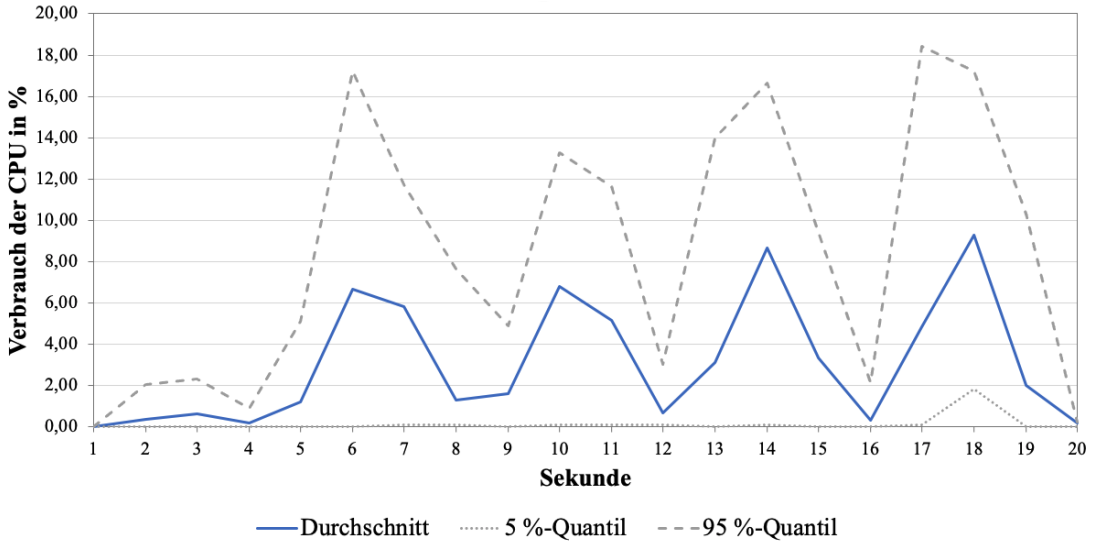

Abbildung 44: CPU - iOS Native - Interaktion: Öffnen und Schließen des Navigation Drawers

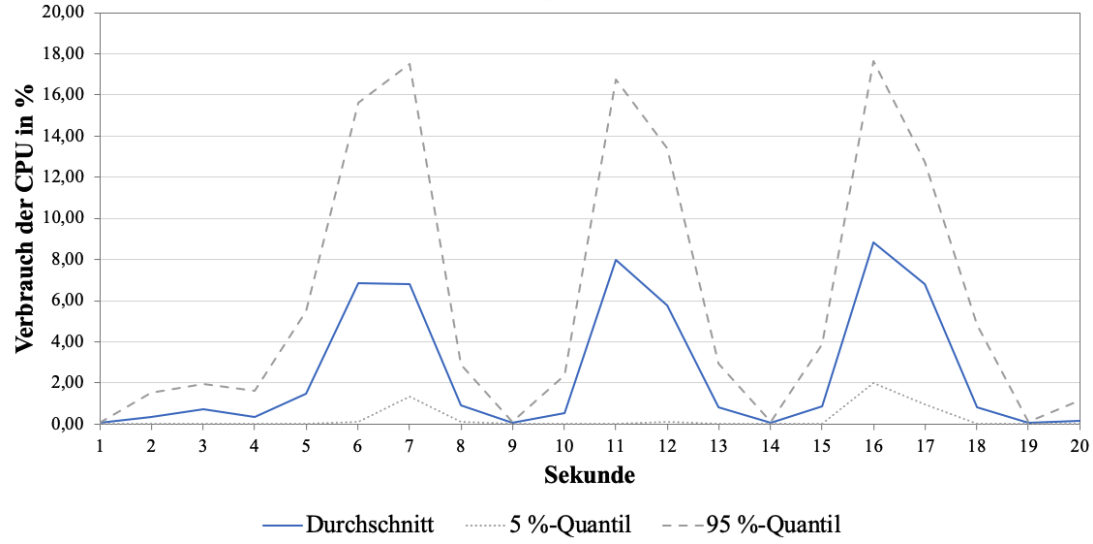

Abbildung 45: CPU - iOS Native - Interaktion: Übergang zwischen zwei Bildschirmen

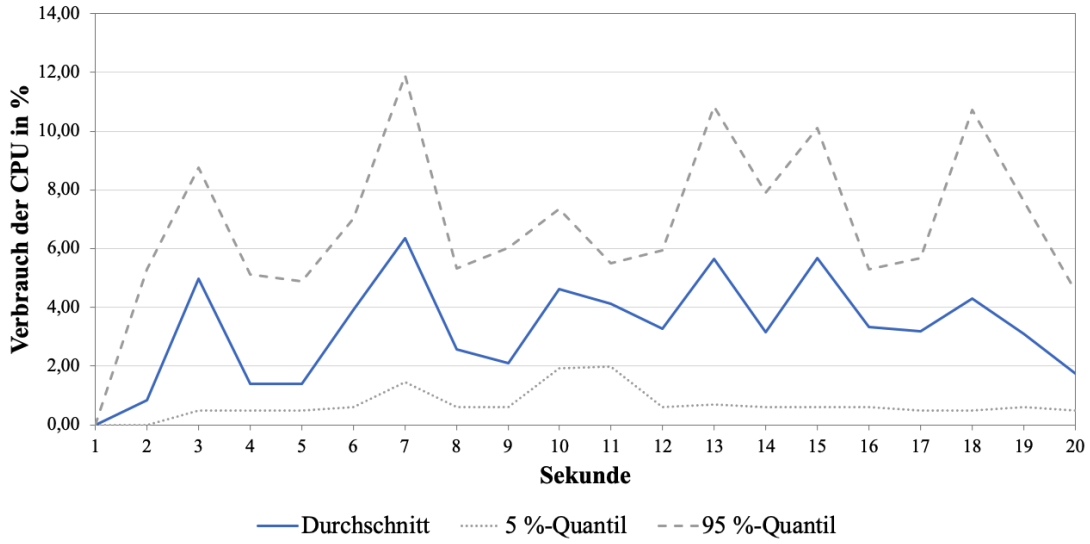

Abbildung 46: CPU - iOS Native - Interaktion: Scrollen durch virtuelle Liste

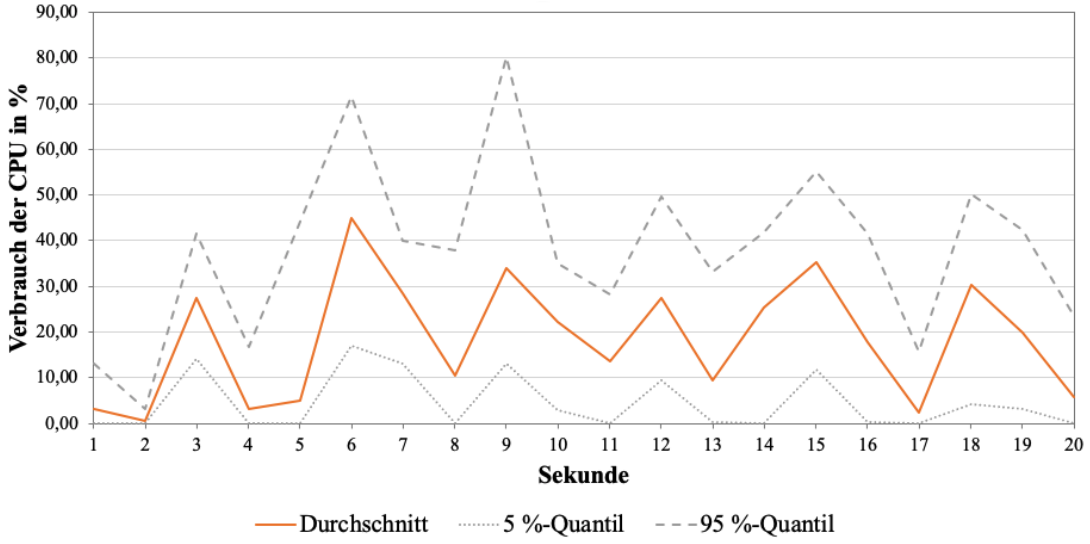

Abbildung 47: CPU - iOS - React Native - Interaktion: Öffnen und Schließen des Navigation Drawers

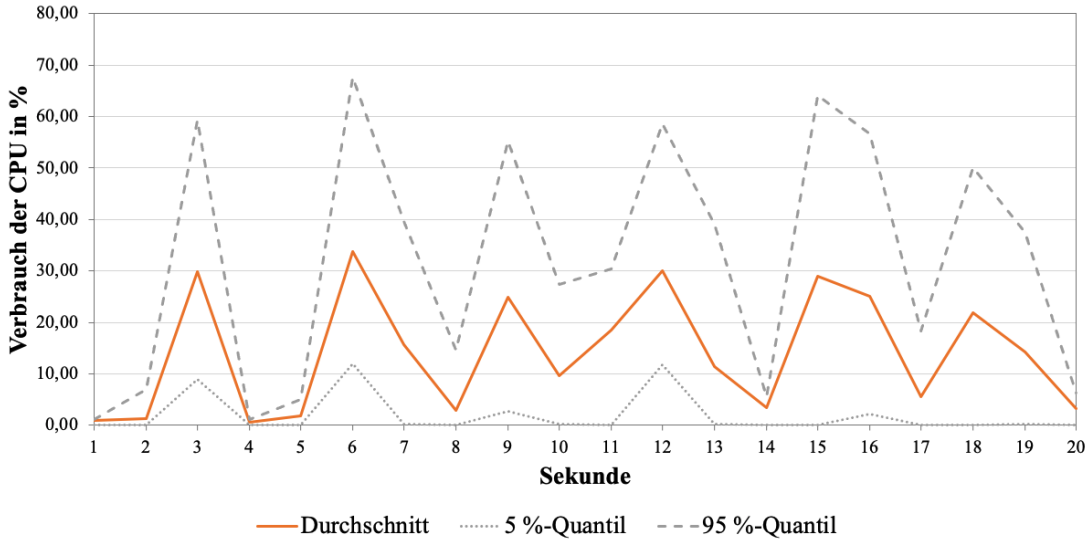

Abbildung 48: CPU - iOS - React Native - Interaktion: Übergang zwischen zwei Bildschirmen

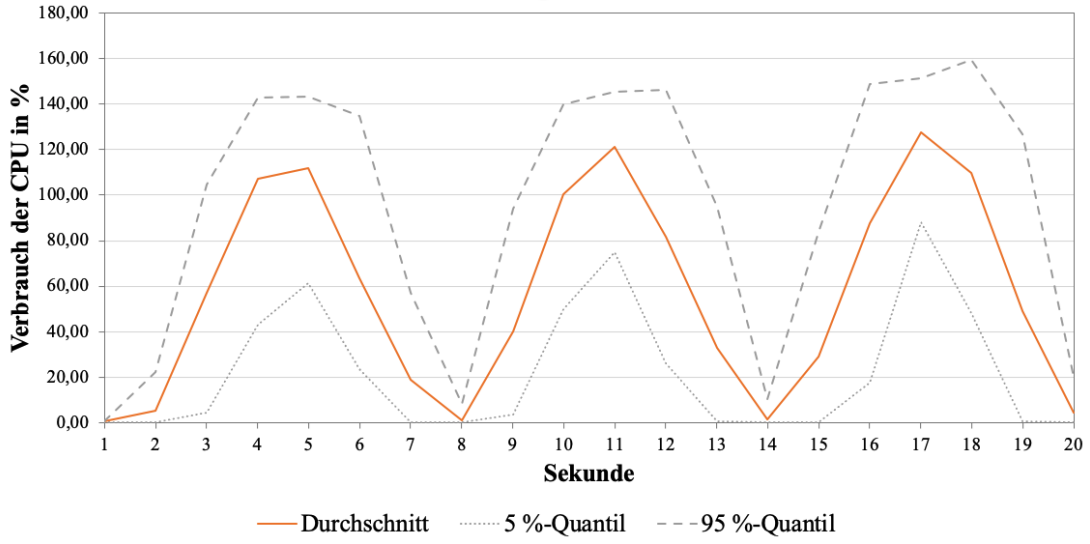

Abbildung 49: CPU - iOS - React Native - Interaktion: Scrollen durch virtuelle Liste

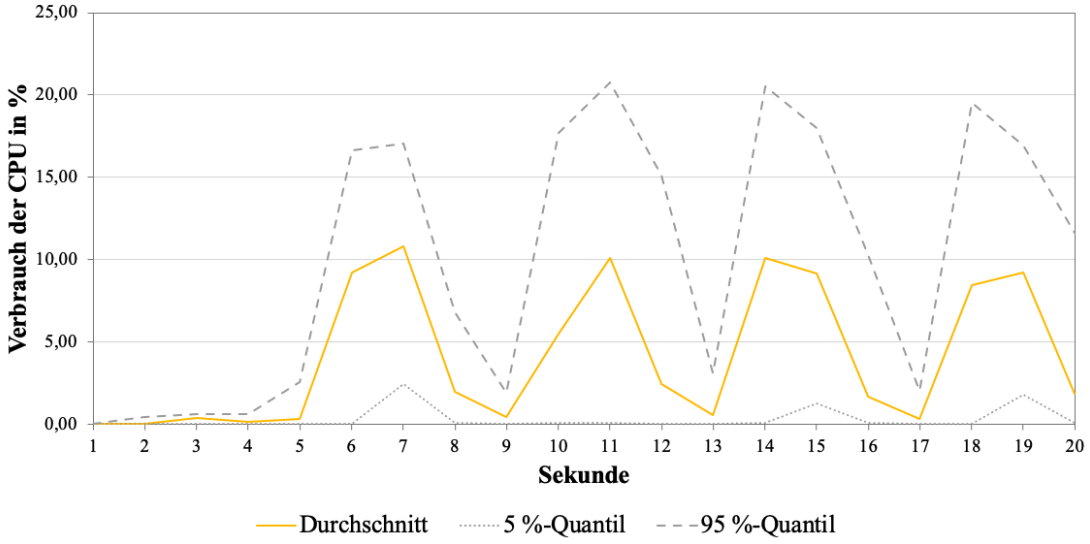

Abbildung 50: CPU - iOS - Ionic/Capacitor - Interaktion: Öffnen und Schließen des Navigation Drawers

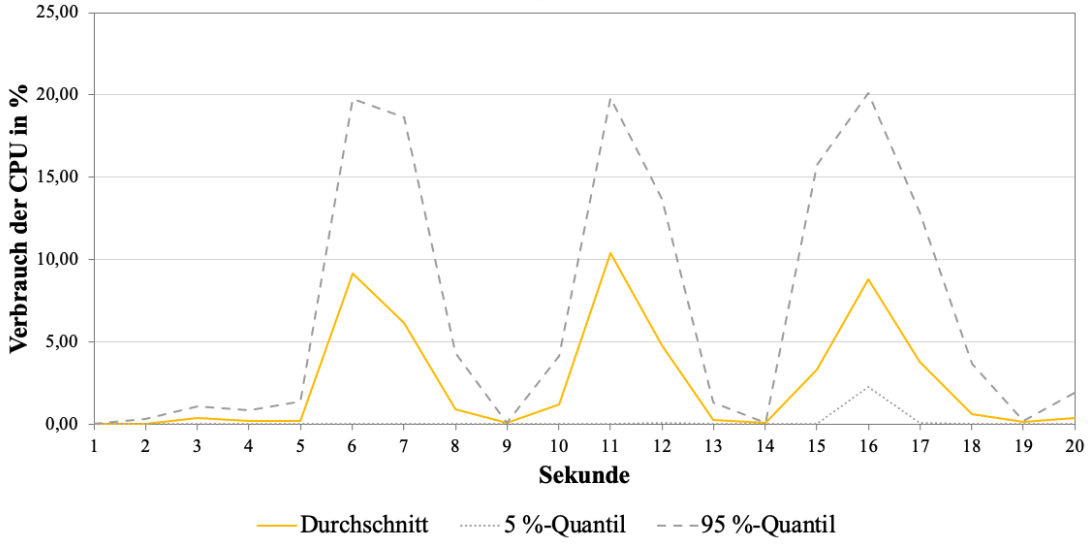

Abbildung 51: CPU - iOS - Ionic/Capacitor - Interaktion: Übergang zwischen zwei Bildschirmen

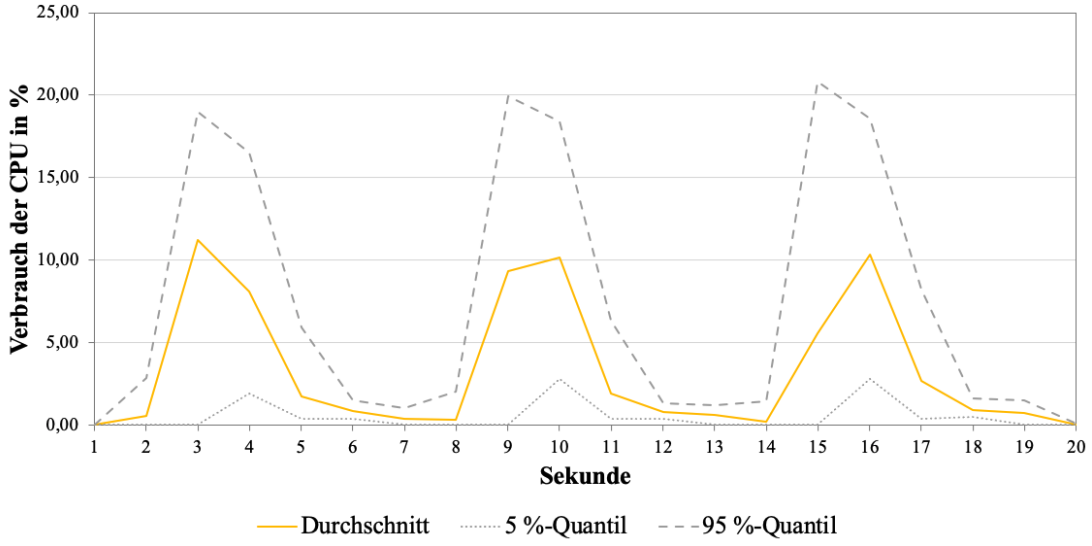

Abbildung 52: CPU - iOS - Ionic/Capacitor - Interaktion: Scrollen durch virtuelle Liste

B. Ergebnisse Verbrauch Arbeitsspeicher pro Anwendung und Testfall

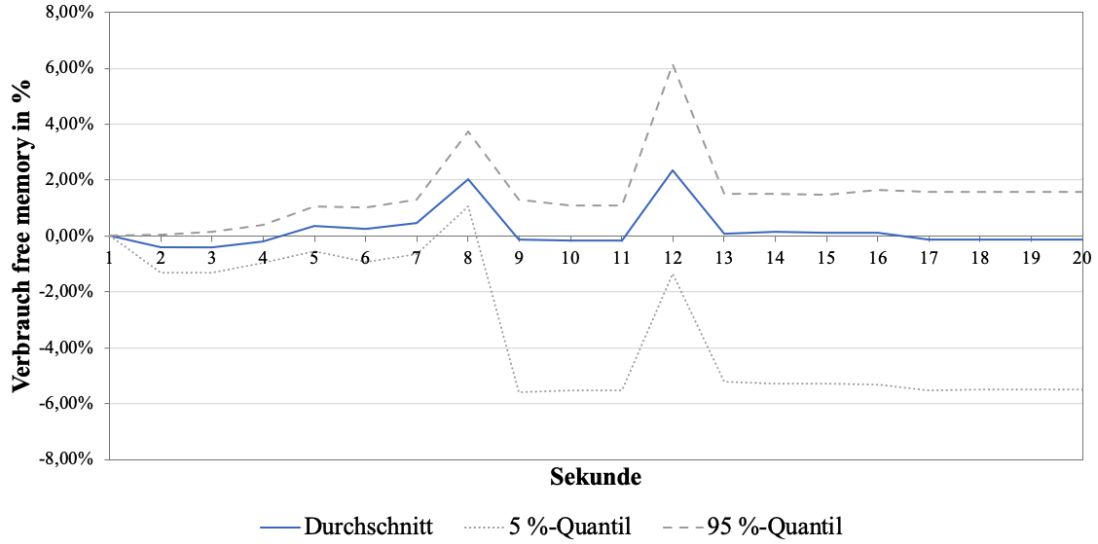

Abbildung 53: Memory - Android Native - Interaktion: Öffnen und Schließen des Navigation Drawers

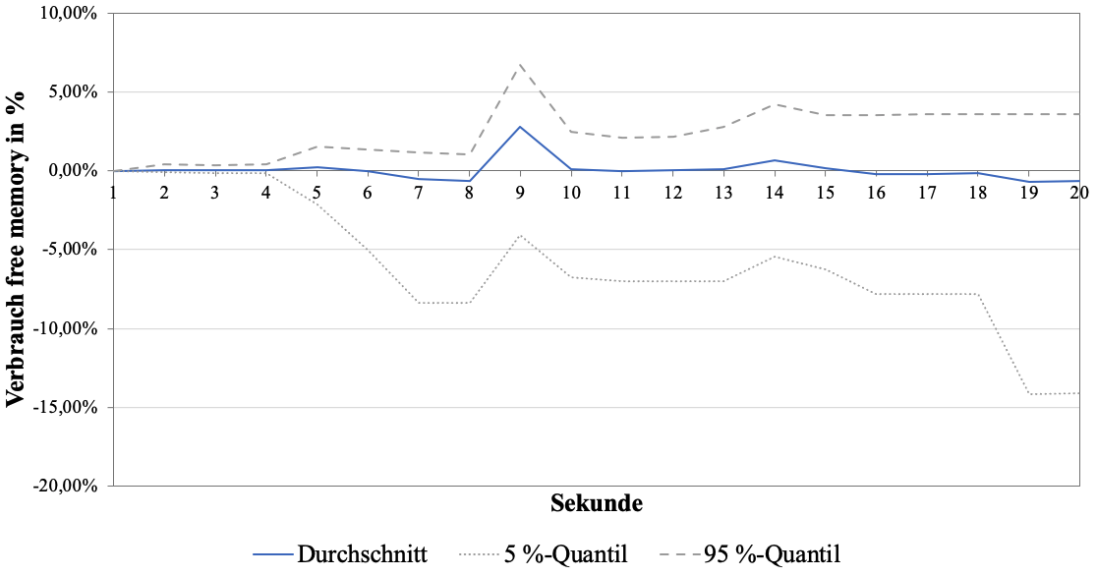

Abbildung 54: Memory - Android Native - Interaktion: Übergang zwischen zwei Bildschirmen

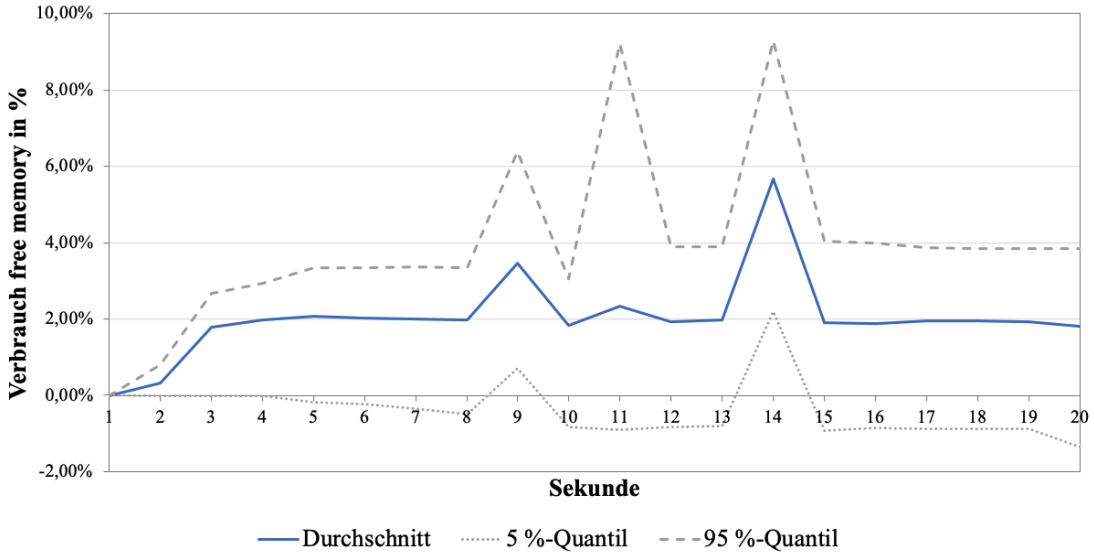

Abbildung 55: Memory - Android Native - Interaktion: Scrollen durch virtuelle Liste

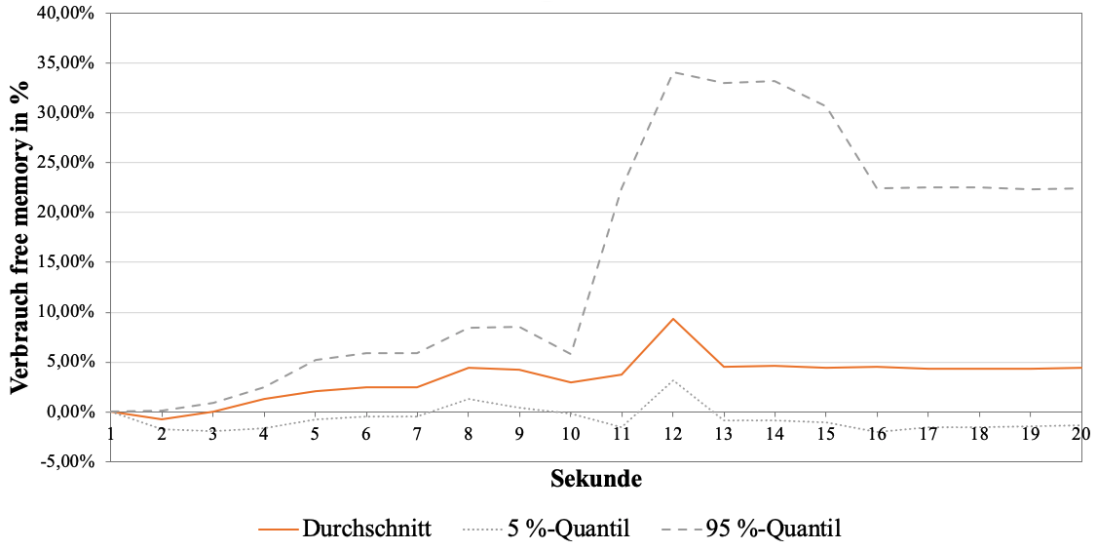

Abbildung 56: Memory - Android - React Native - Interaktion: Öffnen und Schließen des Navigation Drawers

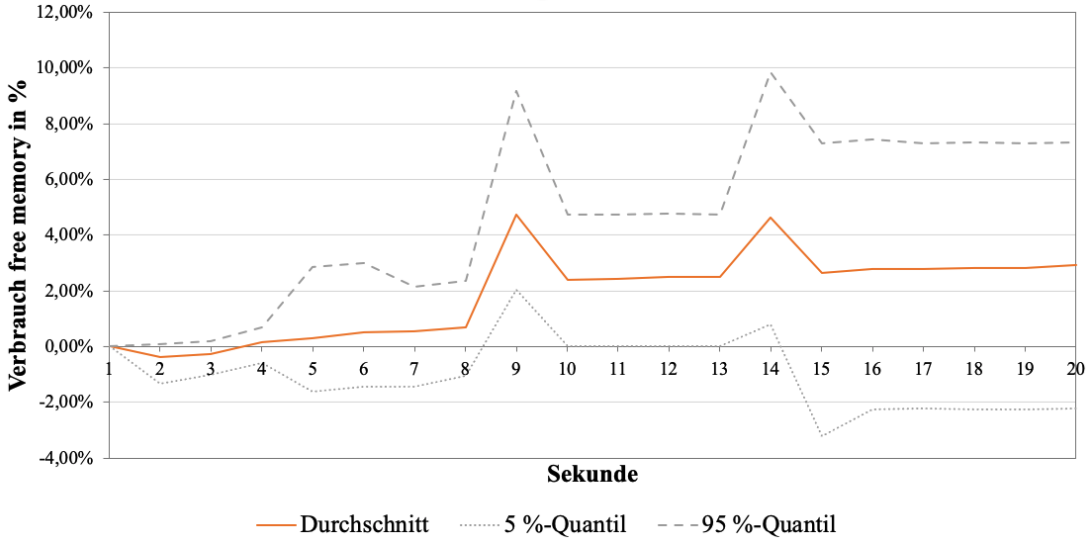

Abbildung 57: Memory - Android - React Native - Interaktion: Übergang zwischen zwei Bildschirmen

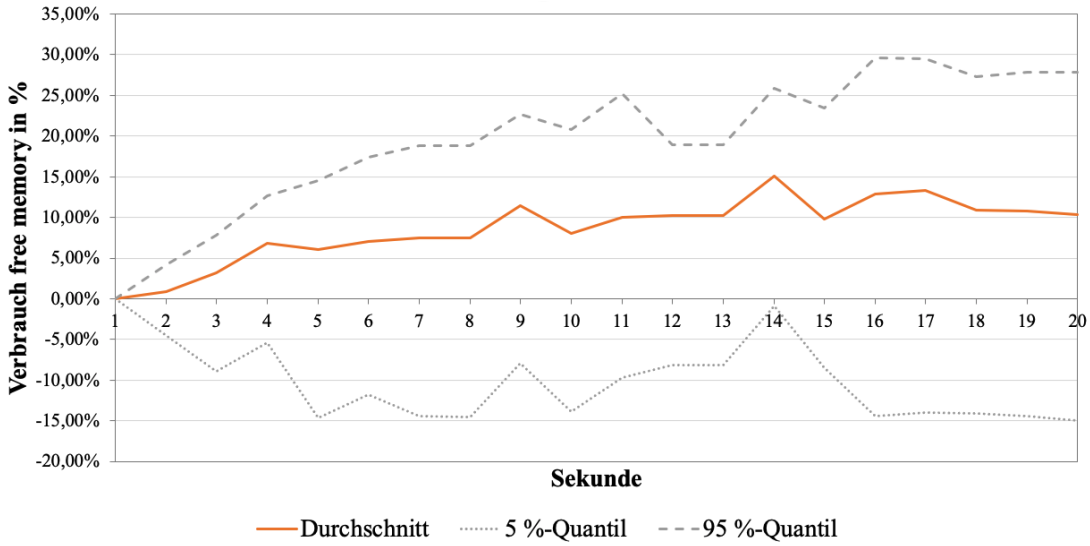

Abbildung 58: Memory - Android - React Native - Interaktion: Scrollen durch virtuelle Liste

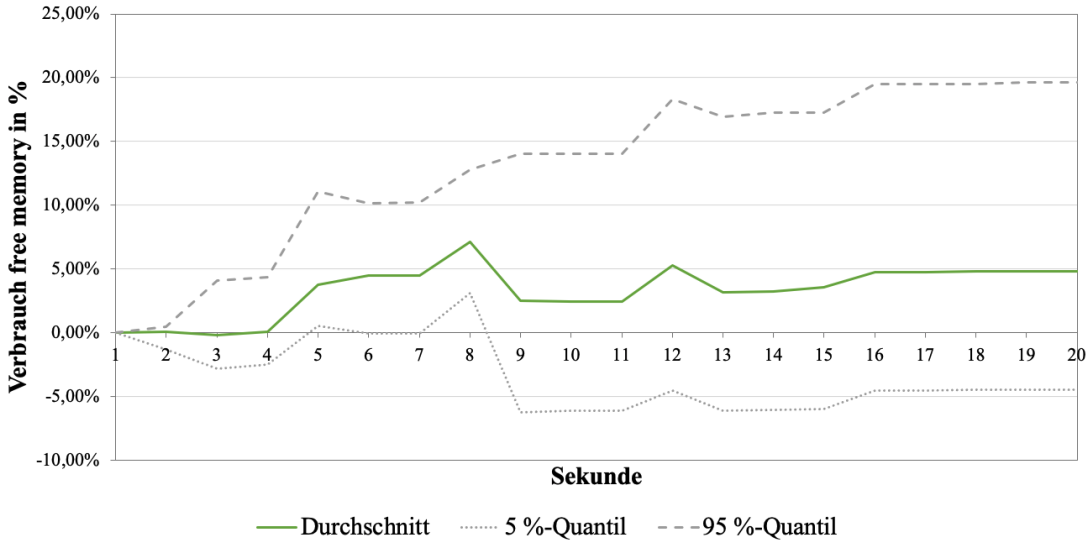

Abbildung 59: Memory - Android - Flutter - Interaktion: Öffnen und Schließen des Navigation Drawers

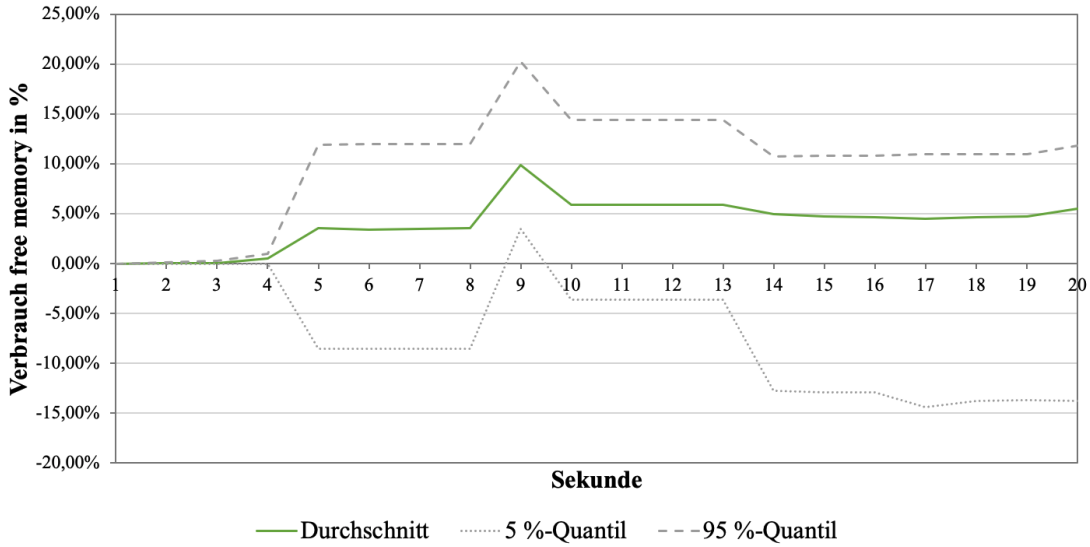

Abbildung 60: Memory - Android - Flutter - Interaktion: Übergang zwischen zwei Bildschirmen

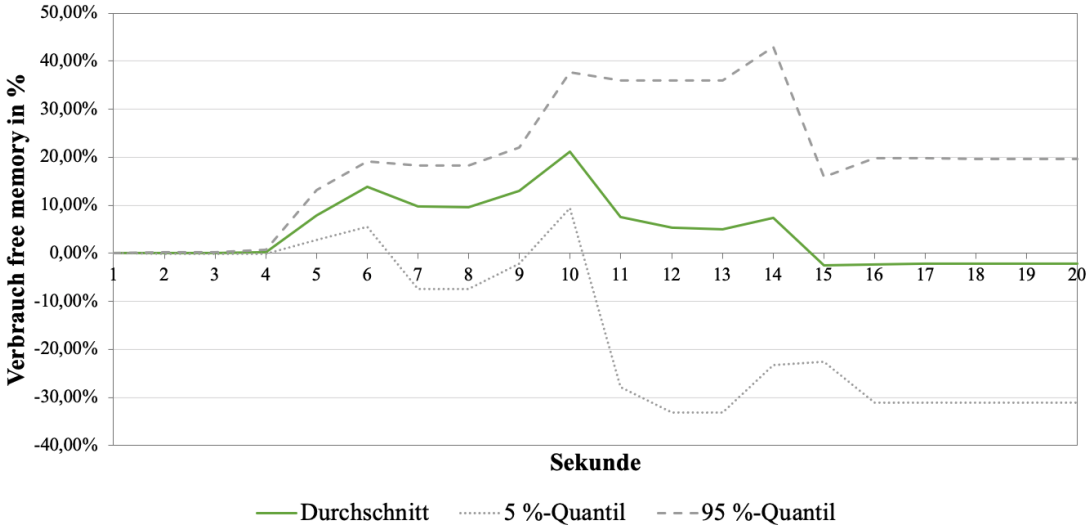

Abbildung 61: Memory - Android - Flutter - Interaktion: Scrollen durch virtuelle Liste

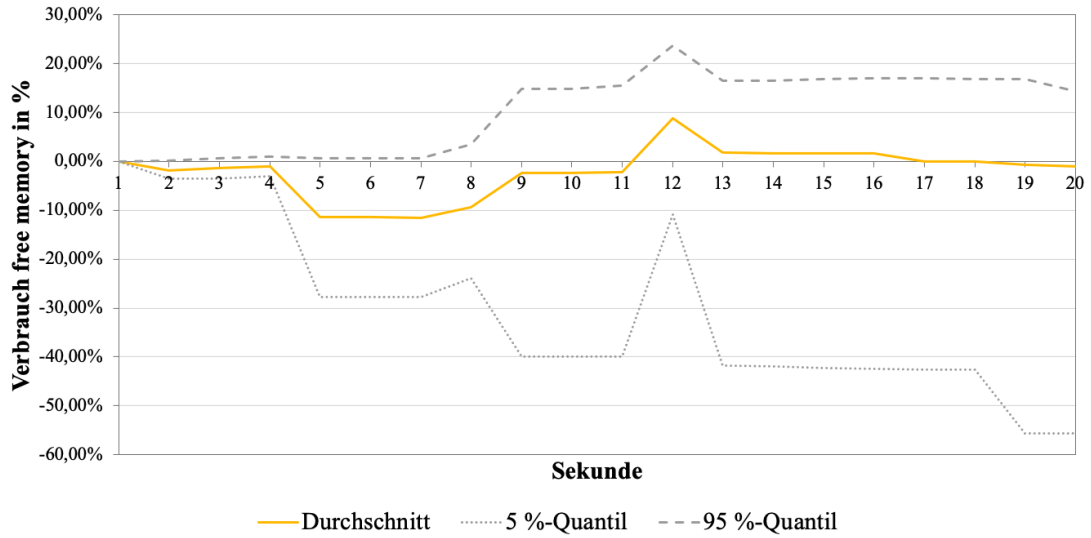

Abbildung 62: Memory - Android - Ionic/Capacitor - Interaktion: Öffnen und Schließen des Navigation Drawer

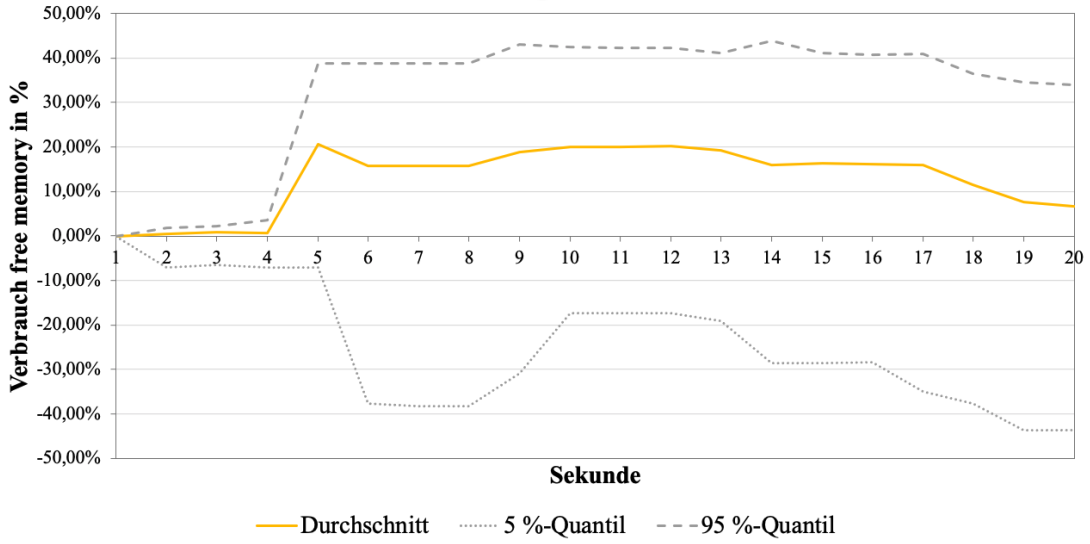

Abbildung 63: Memory - Android - Ionic/Capacitor - Interaktion: Übergang zwischen zwei Bildschirmen

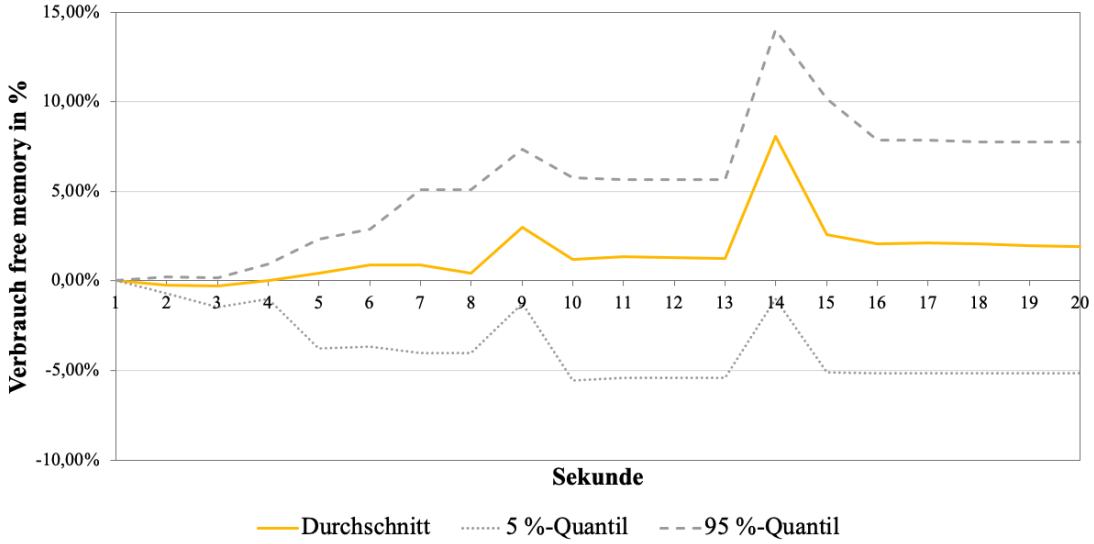

Abbildung 64: Memory - Android - Ionic/Capacitor - Interaktion: Scrollen durch virtuelle Liste

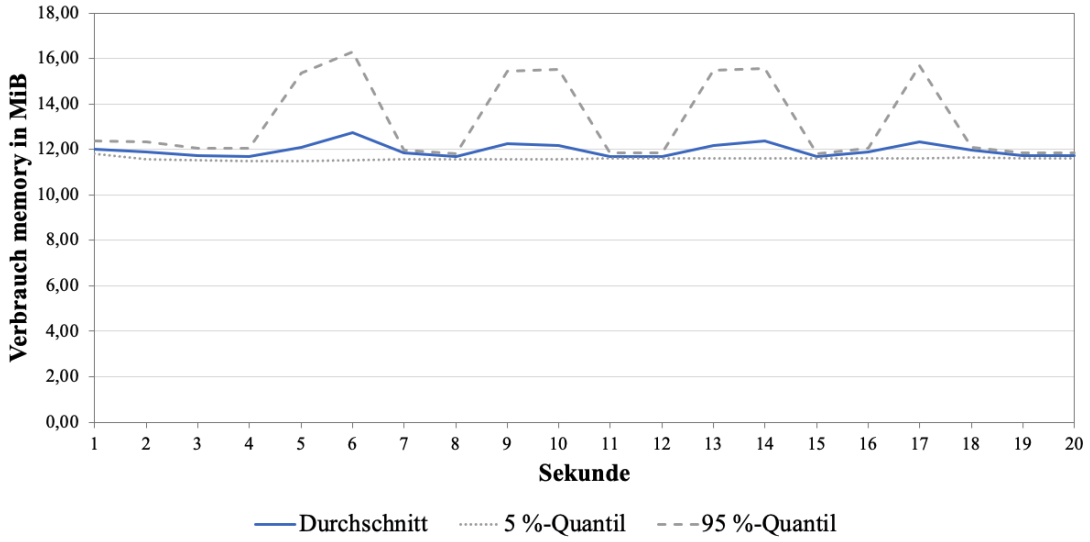

Abbildung 65: Memory - iOS Native - Interaktion: Öffnen und Schließen des Navigation Drawer

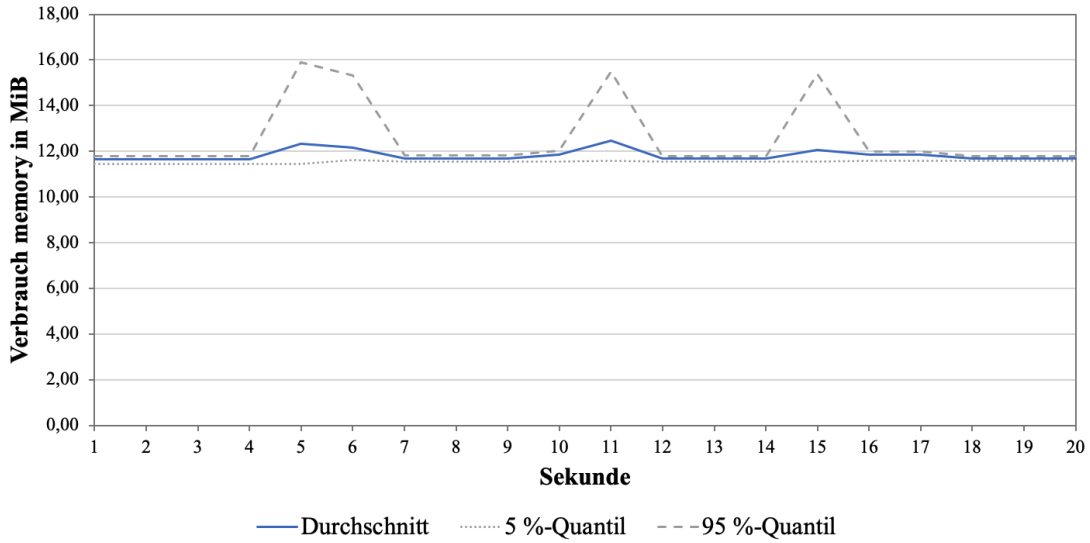

Abbildung 66: Memory - iOS Native - Interaktion: Übergang zwischen zwei Bildschirmen

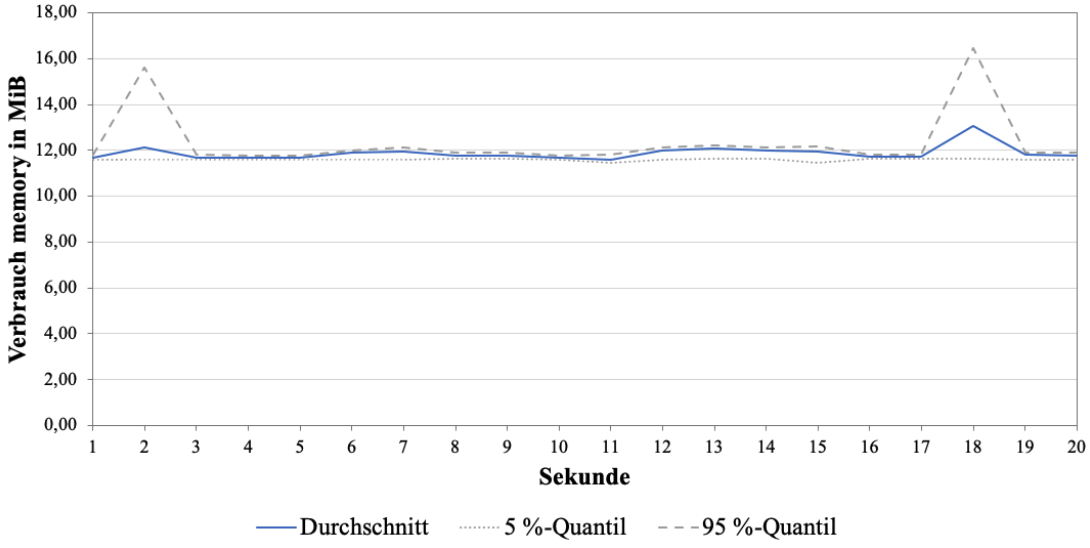

Abbildung 67: Memory - iOS Native - Interaktion: Scrollen durch virtuelle Liste

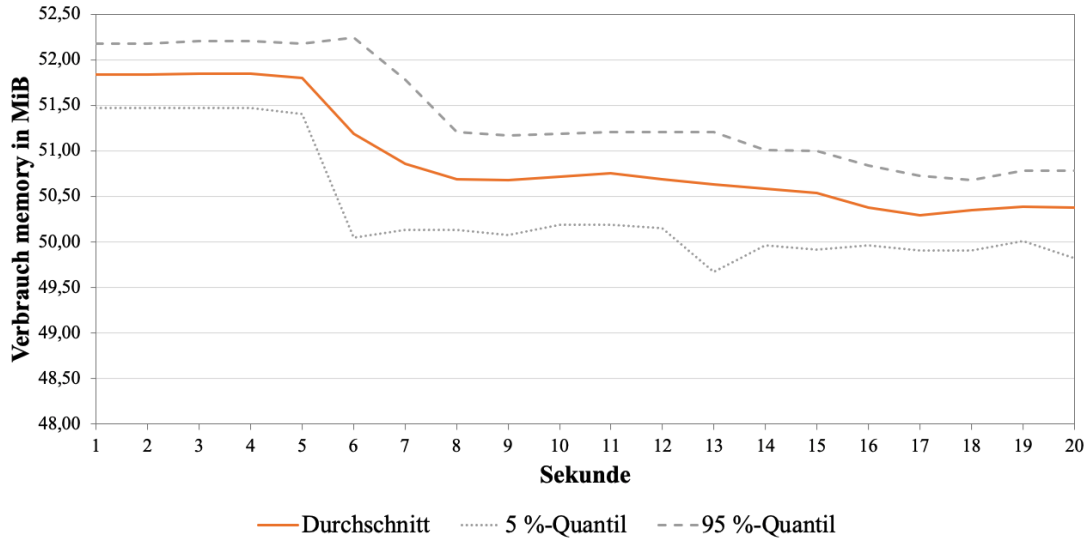

Abbildung 68: Memory - iOS - React Native - Interaktion: Öffnen und Schließen des Navigation Drawer

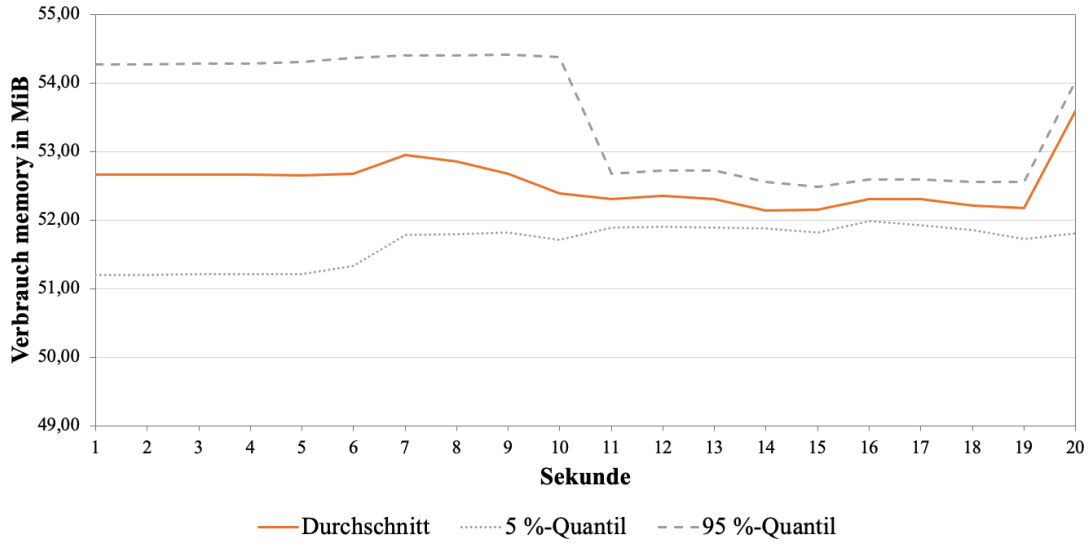

Abbildung 69: Memory - iOS - React Native - Interaktion: Übergang zwischen zwei Bildschirmen

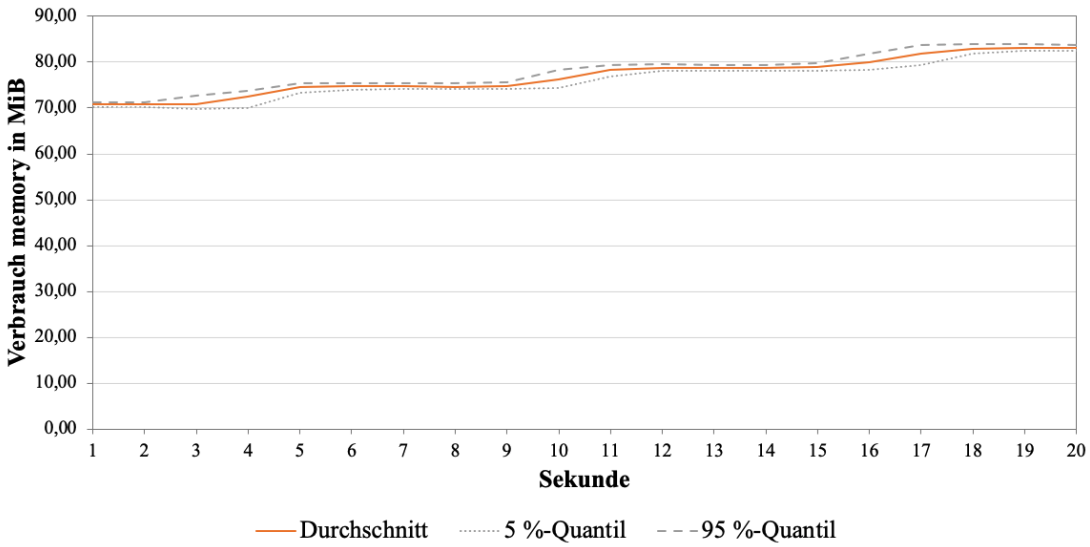

Abbildung 70: Memory - iOS - React Native - Interaktion: Scrollen durch virtuelle Liste

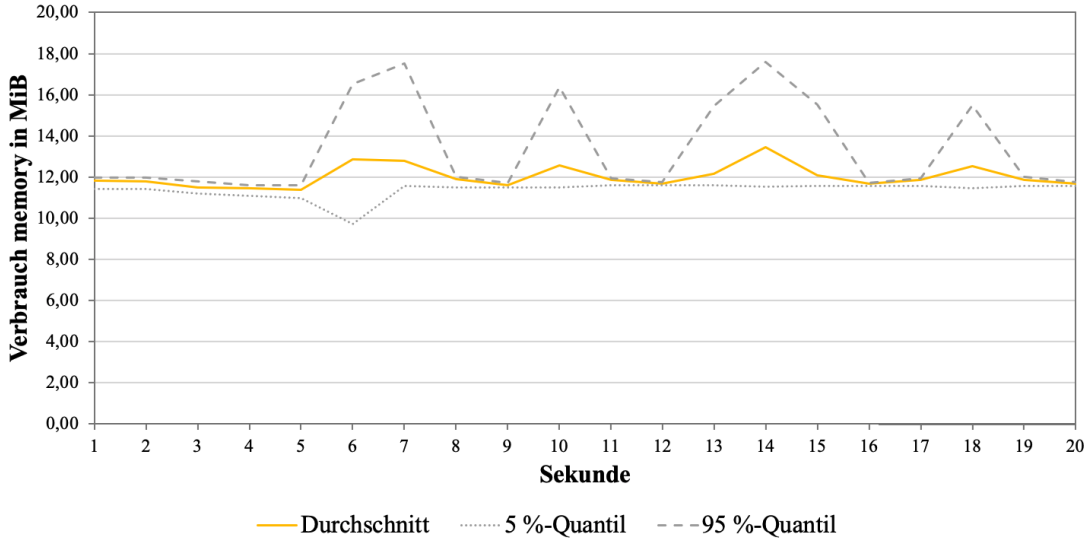

Abbildung 71: Memory - iOS - Ionic/Capacitor - Interaktion: Öffnen und Schließen des Navigation Drawer

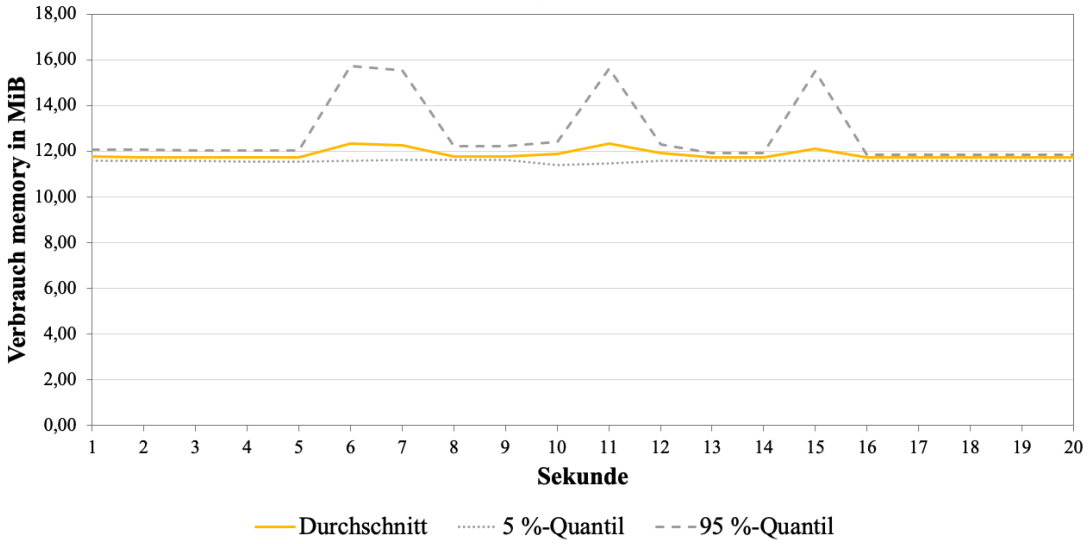

Abbildung 72: Memory - iOS - Ionic/Capacitor - Interaktion: Übergang zwischen zwei Bildschirmen

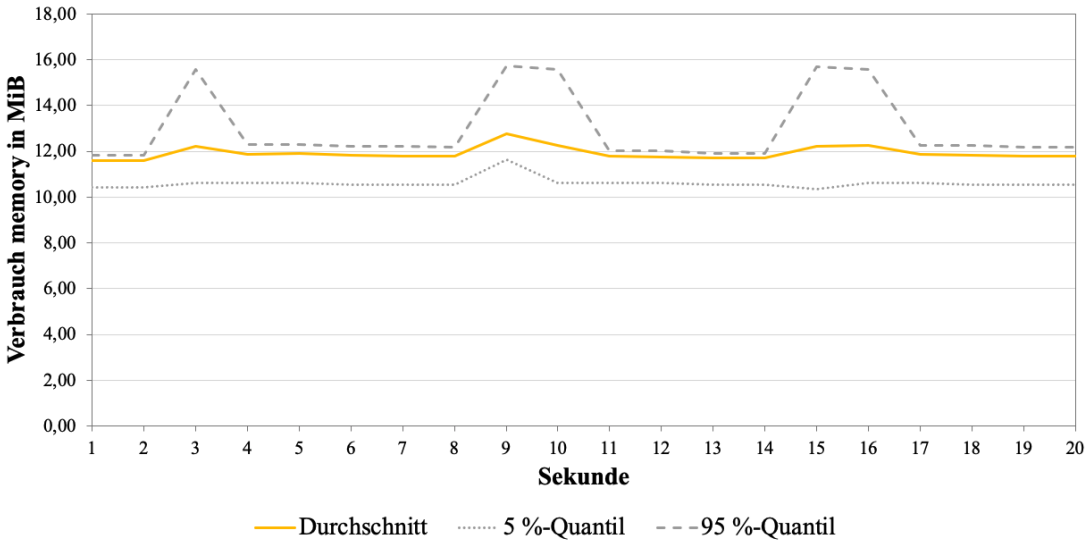

Abbildung 73: Memory - iOS - Ionic/Capacitor - Interaktion: Scrollen durch virtuelle Liste