Telematics
Computer Systems

Freie Universität Berlin

Computer Systems and Telematics — Dependable Systems

Masterarbeit

Design und Implementierung eines energieeffizienten Schachspiels mit mobilem Offloading

Christian Bruns

Matr. 4518546

Betreuer: Prof. Dr. Katinka Wolter
Betreuender Assistent: Dr. Huaming Wu

Institut für Informatik, Freie Universität Berlin, Deutschland

3. Februar 2016

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, sind als solche gekennzeichnet. Die Zeichnungen oder Abbildungen sind von mir selbst erstellt worden oder mit entsprechenden Quellennachweisen versehen. Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner Prüfungsbehörde eingereicht worden.

Berlin, den 3. Februar 2016

(Christian Bruns)

Zusammenfassung

Smartphones und Tablets sind schon seit einigen Jahren nicht mehr aus dem Alltag der Menschen wegzudenken. Während der allgemeine Trend von Hardware und Software in Richtung *höher, schneller, weiter* verläuft, bleibt die Entwicklung der Energieversorgung mobiler Geräte auf der Strecke. Trotz immer höherer Anforderungen durch Hardware, wie Sensoren oder größere Bildschirme, hat sich in den letzten Jahren nicht viel an der Kapazität der Akkus verändert. Mobiles Offloading greift dieses Problem auf und erreicht durch das Ausführen von rechenintensivem Code einer Anwendung auf einem leistungsstarken Server nicht nur kürzere Ausführungszeiten, sondern auch einen geringeren Energieverbrauch. Ermöglicht wird dies durch immer höhere Übertragungsgeschwindigkeiten und besserer Erreichbarkeit der kabellosen Netzwerke. Doch dadurch kommen auch neue Problemstellungen auf Forschung und Entwicklung zu: Welche Teile einer Anwendung sollen potentiell vom Server ausgeführt werden? Wann lohnt sich die dezentrale Ausführung? Wie wird mit der stark schwankenden Qualität drahtloser Netzwerke umgegangen?

Im Rahmen dieser Arbeit wurde ein Schachspiel für Smartphones entwickelt, dessen Energieverbrauch durch mobiles Offloading drastisch reduziert wird. Da die Suche nach dem nächsten Schachzug durch die Schach-KI hohe Anforderungen an den Prozessor des Mobilgerätes stellt, ist ein Schachspiel für die Verwendung von mobilem Offloading wie geschaffen. Durch das Auslagern der Suche an den Server spart das Mobilgerät Zeit und Energie. Die Entscheidung, eine Berechnung entfernt auszuführen, wird während der Ausführung anhand der geschätzten Laufzeiten getroffen.

In dieser Arbeit wird dem Leser ein Überblick des aktuellen Forschungsstandes zum Thema mobiles Offloading verschafft, die Entwicklung des Schachspiels nachvollziehbar erläutert und anhand geeigneter Experimente und deren Auswertung nachgewiesen, dass durch mobiles Offloading Zeit und Energie gespart wird.

Abstract

For some years smartphones and tablets have been an indispensable part of people's everyday lives. While hardware and software is trending to be better and faster the evolution of the battery of mobile devices is left behind. Despite higher demands on the battery due to new hardware like: built-in sensors and bigger displays, the capacity of the battery hasn't changed too much over time. At this point mobile offloading is saving time and energy by migrating heavy executions to a faster and stationary server. This technology is made possible by faster mobile connections and better availability of wireless networks. But with new technologies new problems arise: Which parts of an application should be executed by the server? When is a remote execution profitable? How to deal with continuously changing quality of wireless networks?

Within the scope of this thesis a chess game for smartphones with focus on reduced energy consumption was developed. Since the search for the next move by the chess-AI is a heavy task for the processor of a mobile device, the migration of this task to a faster server makes sense. By outsourcing the search algorithm the mobile device is saving time and energy. The decision to or not to offload will be made during runtime by estimating the execution time of the task.

In this thesis the reader will get an overview of the current state of research, a detailed explanation of the implementation of mobile offloading in the chess game and the thesis demonstrates the time and energy savings that are possible with the use of mobile offloading on the basis of suitable experiments.

Inhaltsverzeichnis

Stichwortverzeichnis

Abkürzungsverzeichnis

(k+1)-Algorithmus *Adaptive (k+1) Multi-Constraint Partitioning* Algorithmus. 11, 13

AESTET *Automated estimation system of task execution times*. 15, 16, 23, 33, 38, 39, 54, 55, 57, 61

AIDL *Android Interface Definition Language*. 22

AMDCOP *Android mobile devices computation offloading project*. 13, 23–25, 31

APK *Android Package*. 22

CIL *Common Intermediate Language*. 20

CLR *Common Language Runtime*. 20, 23, 24

CPU *Central Processing Unit*. 8, 13, 20, 31, 32, 38, 53, 55, 56

DMC Graph *dynamic multi-cost* Graph. 11

GPS *Global Positioning System*. 2, 18

HELVM-Algorithmus *heavy-edge and light-vertex matching* Algorithmus. 12

HTTP *Hypertext Transfer Protocol*. ix, 14

JAR Java-Archiv. 13, 14, 22, 59

KI Künstliche Intelligenz. 25, 27

KNN smoothing *k-Nearest Neighbor smoothing*. 16, 31

LTE *Long Term Evolution*. 39, 51

MCOP-Algorithmus *Min-Cost Offloading Partitioning* Algorithmus. 9, 12, 13

QR-Code *Quick Response Code*. 22

RAM *Random-Access Memory*. 56

sd Standardabweichung. 42, 61

VM Virtuelle Maschine. 17–19

WLAN *Wireless Local Area Network*. 2, 15, 23, 38–40, 44, 45, 50–52, 54, 56, 58, 60

XML *Extensible Markup Language*. 14

KAPITEL 1

Einführung

1.1 Motivation

Seit Jahren werden große Fortschritte in der Weiterentwicklung tragbarer Geräte erzielt. Smartphones und Tablets werden immer leistungsstärker, während, trotz des Trends zu einem größeren Display, die Geräte im Design immer flacher werden. Als Antwort auf die immer besser werdende Hardware bringen Softwareentwickler immer leistungsfähigere Anwendungen auf den Markt. Ein Trend, bei dem die Entwicklung des Akkus bisher nicht mithalten konnte.

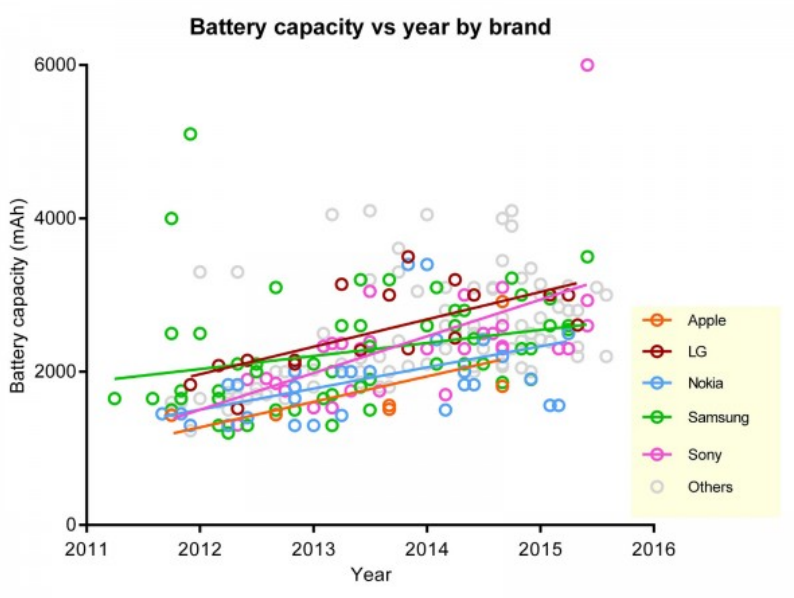


Abbildung 1.1: Entwicklung der Akkukapazität nach Hersteller [1]

[1] Abbildung von: `https://www.reddit.com/r/Android/comments/3jrajo/oc_the_increasing_battery_life_of_mobile_devices/` (Abruf: 18.01.2016)

Über die letzten dreißig Jahre entwickelten sich Akkus von Nickel Cadmium über Nickel-Metallhydrid bis zu Lithium und Lithium-Ionen-Akkus. Letztere mit einer durchschnittlichen Kapazität von 2000 - 2500 mAh. Moderne Smartphones können mit solchen Kapazitäten im Schnitt einen Tag auskommen. Auch durch moderate Nutzung kann die Zeit bis zum nächsten Ladevorgang nur leicht erhöht werden. Abbildung 1.1 zeigt, dass die Entwicklung des Akkus in puncto Kapazität in den letzten Jahren keine großen Fortschritte gemacht hat. Durch immer energiehungrigere Anwendungen und Hardware werden immer größere Anforderungen an den Akku gestellt, die jedoch nur durch größere und schwerere Akkus erfüllt werden könnten.[Wan14]

Ein Ansatz, diesem Problem entgegenzuwirken, ist die Reduzierung des Energieverbrauches. Jedes Jahr werden Transistoren kleiner, wodurch immer mehr Transistoren bei gleichbleibend großem Raum untergebracht werden können. Durch geringere Versorgungsspannung und geringere Abwärme kleinerer Transistoren wird der Energieverbrauch im Ganzen gesenkt. Auch ein besseres Verständnis der Solid State Physik brachte energieeffizientere Hardware hervor. [Wan14]

Ein weiterer wichtiger Aspekt der Energieversorgung liegt in den Anwendungen und den von ihnen verwendeten Ressourcen. Je nach Anwendung, kann das Mobilnetz, das *Wireless Local Area Network* (WLAN), das Bluetooth, das *Global Positioning System* (GPS), das Display und verschiedene Sensoren, den Verbrauch der im Akku gespeicherten Energie negativ beeinflussen. Aber auch komplizierte Berechnungen, die von einer Anwendung durchgeführt werden, tragen maßgeblich zum Energieverbrauch bei. Je länger der Prozessor zum Bearbeiten einer Aufgabe benötigt, um so mehr Energie wird verbraucht. [Wan14]

Hier setzt die Idee an, den Energieverbrauch mit Hilfe von Cloud-Computing zu reduzieren. Bei mobilem Offloading besteht die Möglichkeit, rechenintensive Aufgaben von einem leistungsstarken, an das Stromnetz angeschlossenen Server ausführen zu lassen. Dadurch wird der Energieverbrauch auf dem Mobilgerät selbst reduziert. Jedoch müssen dazu neue Aspekte berücksichtigt werden. Durch die Verwaltung und den Aufbau der Verbindung zum Server entstehen neue Kosten. So muss berücksichtigt werden, dass durch die mobile Nutzung eine gute Verbindung zum Server nicht stetig gewährleistet werden kann. Im schlechtesten Fall verbraucht die Auslagerung der Berechnung infolge langer Wartezeiten und das Aufrechterhalten der Verbindung nicht nur mehr Zeit, sondern auch mehr Energie als eine lokale Ausführung. In diesem Fall sollte das Mobilgerät die Berechnung selbst vornehmen. In der Forschung ist mobiles Offloading ein sehr aktuelles Thema mit großem Potenzial den Energieverbrauch mobiler Geräte zu reduzieren.

1.2 Ziele dieser Arbeit

Ziel dieser Arbeit ist es, ein energieeffizientes Schachspiel für mobile Smartphones zu entwickeln. Um das zu erreichen, sollen rechenintensive Berechnungen per Offloading an einen leistungsstärkeren und an das Stromnetz angebundenen Server ausgelagert werden. Im Rahmen dessen sind weitere Ziele die Verwendung und Evaluierung der von der Arbeitsgruppe Dependable Systems der Freien Universität entwickelten Technologien für das mobile Offloading und der Vergleich dieser mit weiteren aktuellen Technologien.

1.3 Kapitelübersicht

Es folgt eine kurze Übersicht aller Kapitel dieser Arbeit.

Kapitel 2 - Forschungsstand
Erläutert und vergleicht aktuelle Technologien miteinander, die im Zusammenhang mit mobilem Offloading stehen.

Kapitel 3 - Offloading Schach
In diesem Kapitel werden die Anforderungen an das zu entwickelnde Schachspiel gestellt, der Entwurf dokumentiert und die Implementierung im Detail erläutert. Anschließend werden Forschungsfragen aufgestellt und der Versuchsaufbau der Experimente, mit denen die Fragen beantwortet werden, festgelegt.

Kapitel 4 - Evaluation
Hier werden die Ergebnisse der im vorherigen Experimente visualisiert und interpretiert. Darauf aufbauend werden Schlussfolgerungen gezogen und Ansätze für mögliche weiterführende Forschung erläutert.

KAPITEL 2

Forschungsstand

2.1 Mobiles Offloading

Cloud Computing hat in der jüngeren Vergangenheit großes Interesse erfahren. Die primäre Motivation liegt dabei in der Bereitstellung scheinbar unbegrenzter, virtueller Ressourcen, genau zu dem Zeitpunkt, zu dem sie benötigt werden. Für mobile Geräte unterscheiden sich die Anforderungen an das Cloud Computing im Vergleich zu stationären, per Kabel ans Internet angebundenen Geräten. Während bei stationären Geräten die Transferzeit und die Kosten für das Übertragen von großen Mengen an Daten zu beachten sind, spielt bei mobilem Cloud Computing zusätzlich der Energieverbrauch eine sehr wichtige Rolle.

Wie schon in Kapitel 1.1 erwähnt, lässt sich mit mobilem Offloading, einer Form des Cloud Computing, der Energieverbrauch deutlich reduzieren. Unter bestimmten Bedingungen sind Energieersparnisse von 20% möglich [MN10]. Damit mobiles Offloading einen Vorteil bringt, muss die folgende Ungleichung erfüllt sein:

$$E_{cloud} < E_{local} \tag{2.1}$$

Dabei bezeichnet E_{cloud} die Energiekosten für den Transfer einer Berechnungseingabe und der daraus folgenden Berechnungsausgabe und E_{local} die Energiekosten für die lokale Ausführung einer Berechnung. [MN10]

Abbildung 2.1 zeigt den Ablauf eines Offloading-Vorgangs. Bevor eine Berechnung ausgeführt wird, die potentiell von einem Server ausgeführt werden kann um Energie oder Zeit zu sparen, muss entschieden werden, ob ein Offloading-Vorgang sinnvoll wäre oder nicht. Diese Entscheidung ist eine der Hauptproblemstellungen und wird in Kapitel 2.4 näher erläutert. Sollte ein Offloading-Vorgang durchgeführt werden, kann es immer noch zu einem Fehler kommen. Zum Beispiel wenn durch schwankende Qualität der Netzanbindung der Server nicht erreicht wird. In einem solchen Fall muss das System auf einen anderen Server ausweichen oder die Berechnung doch lokal ausführen.

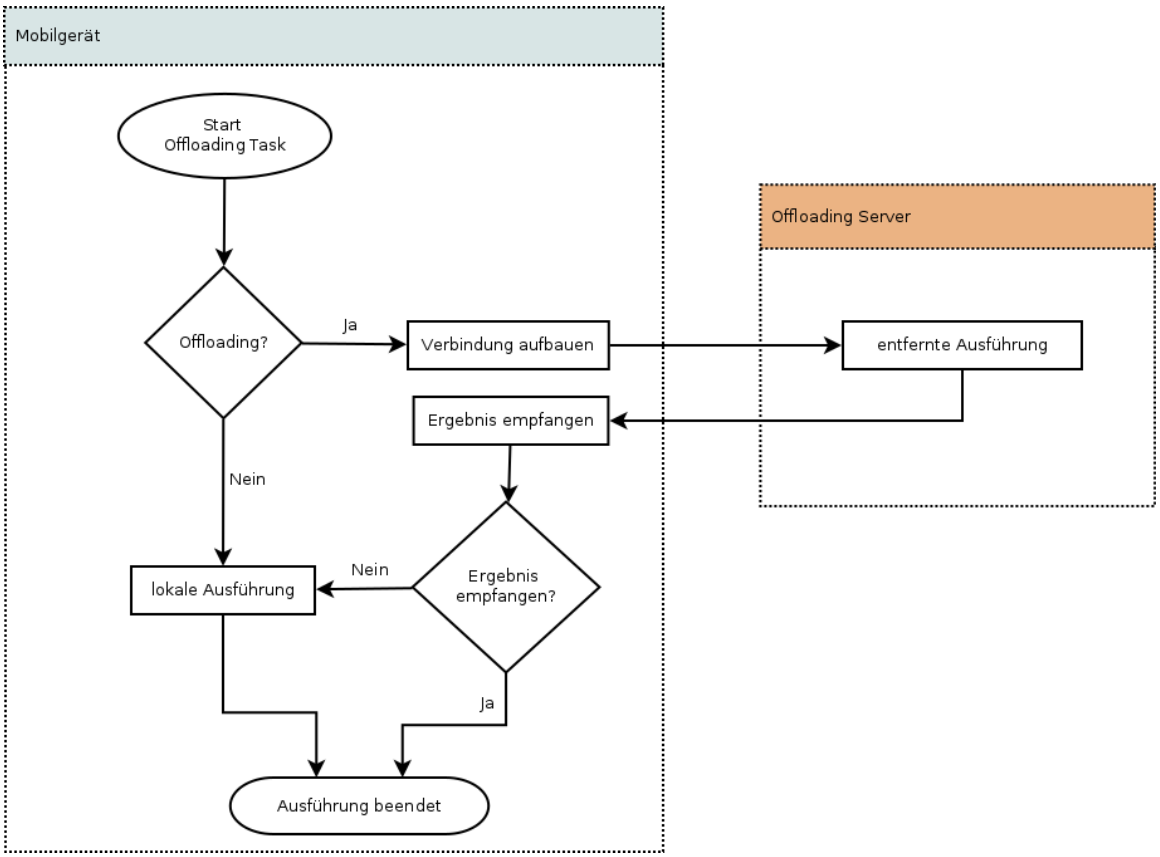


Abbildung 2.1: Offloading Flussdiagramm

2.2 Identifikation der Offloading-Tasks

Um eine mobiles Offloading zu ermöglichen, müssen innerhalb der Anwendung Aufgaben, die während der Laufzeit von einem Server ausgeführt werden sollen, anhand ihrer Funktionalität entsprechend von einander abgegrenzt und klare Schnittstellen identifiziert werden. Nur so kann eine Entscheidung getroffen werden, ob eine bestimmte Berechnung lokal oder auf einem Server durchgeführt werden soll. Diese Aufgaben werden hier und im Folgenden als (Offloading-)Tasks bezeichnet. Diese Partitionierung muss, je nach Offloading-System, manuell vom Entwickler erstellt werden oder wird vom Offloading-System selbständig ermittelt.

Bei der Partitionierung müssen verschiedene Abhängigkeiten beachtet werden. So greifen manche Tasks auf Sensoren, auf Daten des Mobilgerätes oder auf die grafische Benutzeroberfläche zu. Hierbei muss beachtet werden, dass der Server keinen Zugriff auf die nöten Schnitstellen hat. Einige Systeme übernehmen die Aufteilung einer Anwendung in die verschiedenen Offloading-Tasks automatisch. Solche Systeme gehen mit einer hohen Flexibilität einher. Sollte sich die Architektur der Anwendung durch neue oder geänderte Funktionalität verändern, kann sich das System ohne menschliches Eingreifen an die veränderten Umstände anpassen. [TS02]

Bei der manuellen Partitionierung werden Tasks, Schnittstellen und Abhängigkeiten zu erst

von einer menschlichen Instanz identifiziert. Dabei muss auf die selben Beschränkungen wie bei der automatischen Partitionierung geachtet werden. Danach muss das System so angepasst werden, dass Client und Server fähig sind, die für das Offloading vorgesehenen Tasks auszuführen und die entsprechende Task-Eingabe bzw. -Ausgabe zwischen Client und Server zu kommunizieren. Durch manuelle Partitionierung sind große Energieeinsparungen möglich, da der Entwickler selbst gezielt rechenintensive Tasks auswählen kann und so nur ein kleiner Overhead entsteht. Der Vorteil gegenüber einer automatischen Partitionierung liegt in der, gerade für kleinere Systeme mit wenigen Offloading-Tasks, einfacheren Implementierung.

2.3 Neustart in mobilem Offloading

Die schwankende Qualität drahtloser Netzwerke kann unter Umständen zu einer unzuverlässigen Kommunikation zwischen Mobilgerät und Server führen. Um durch Offloading Zeit und Energie sparen zu können, ist ein zuverlässiges Netzwerk jedoch unerlässlich. Sollte sich die Verbindung während eines Offloading-Vorgangs stark verschlechtern oder sogar komplett ausfallen, wird durch das Warten auf ein Ergebnis, das nie oder nur mit großer Verzögerung eintrifft, der Energieverbrauch und die Ausführungszeit stark erhöht. Abhilfe kann das Neustarten der Berechnung sein. Im Rahmen des mobilen Offloadings sind zwei Arten des Neustartens einer Berechnung verbreitet: Wird erkannt, dass die Qualität des Netzwerkes das Beenden des Offloading-Tasks nicht zulässt, wird auf eine Qualitätsverbesserung gewartet und die Anwendung daraufhin weiter ausgeführt. Sollte es sich nur um einen kurzen Ausfall handeln, funktioniert dieses Verfahren sehr gut. Wenn das Netzwerk es jedoch über einen längeren Zeitraum nicht zulässt den Offloading-Vorgang zu beenden, führt dieses Verfahren zu deutlich höherem Energie- und Zeitverbrauch. Alternativ kann sofort nach Einbrechen der Netzwerkqualität auf dem Mobiltelefon ein Neustart durchgeführt werden. Sollte das Netzwerk jedoch nur kurzzeitig die entfernte Ausführung eines Offloading-Tasks nicht zulassen, führt auch ein sofortiger, lokaler Neustart zu einem erhöhten Energie- und Zeitverbrauch, da sich ein neuer Offloading-Versuch möglicherweise nach der Verbesserung des Netzwerkes immer noch lohnen würde. Die Kombination dieser beiden Verfahren zu einem hybriden Verfahren ist ein vielversprechender Ansatz, die Vorteile beider Verfahren ohne die jeweiligen Nachteile zu nutzen. Dabei werden die Offloading-Tasks in bestimmten Intervallen erneut ausgeführt bis die Berechnung erfolgreich ist. Zuerst werden weitere entfernte Ausführungen durchgeführt. Sobald ein bestimmter Schwellenwert an Neustarts der entfernten Ausführung erreicht ist, wird die Berechnung lokal auf dem Mobilgerät durchgeführt. Anhand der erwarteten Ausführungszeit des Offloading-Tasks kann der optimale Schwellenwert ermittelt werden. Abbildung 2.2 zeigt diesen Prozess. [Wan15, S. 5f, 107f]

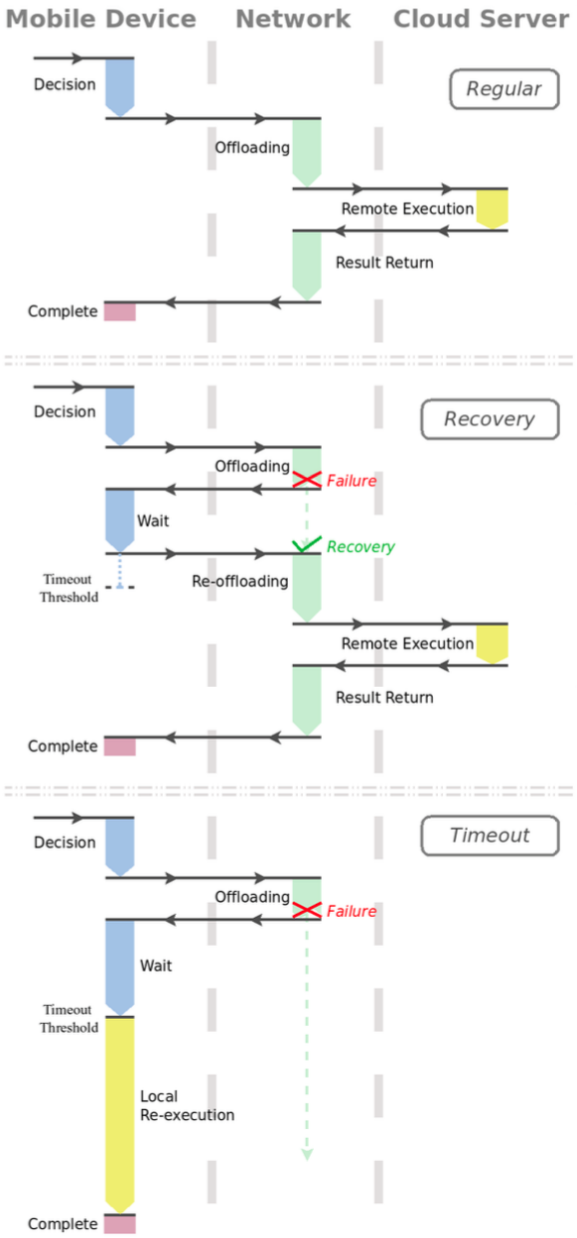


Abbildung 2.2: Neustart in mobilem Offloading [Wan15, S. 58]

2.4 Offloading-Entscheidung

Nicht immer ist das Auslagern eines Tasks an einen Server sinnvoll. Durch die Mobilität kommt es zu schwankender Qualität der Netzanbindung. Bei schlechter Verbindung zum Datennetz ist ein Offloading-Vorgang möglicherweise teurer im Sinne von Zeitersparnis und Energieverbrauch als eine lokale Ausführung. Zu diesem Zwecke muss für jeden Task vor der Ausführung oder zur Laufzeit eine Entscheidung getroffen werden. Wird die Entscheidung vor der Ausführung der Anwendung getroffen, hat das einen geringen Overhead zur Folge. Jedoch kann es auf Grund von unterschiedlichen Eingaben eines Tasks zu unterschiedlichen Laufzeiten kommen oder Ressourcen, wie Akkustand, verfügbarer Speicher und verschiedene Eigenschaften des Netzwerkes, verändern sich während der Ausführung und die Entscheidung muss angepasst werden. Um den optimalen Ausführungsort eines Tasks zu bestimmen, muss die Entscheidung daher dynamisch während der Laufzeit der Anwendung getroffen werden. [Wu15, S. 18]

Diese Entscheidung kann nicht für jeden Task unabhängig getroffen werden, jede Entscheidung beeinflusst weitere Entscheidungen. Die Problemstellung lässt sich als Graph beschreiben. Die Kosten für die lokale und entfernte Ausführung eines Tasks werden dabei als gewichtete Knoten und die Kommunikationskosten zwischen den Tasks als gewichtete Kanten zwischen den Knoten dargestellt [HK00]. Zur Lösung bieten sich Partitionierungsalgorithmen an, die den Graphen in zwei oder mehrere Partitionen teilen. Eine Partition beinhaltet alle Tasks, die lokal auf dem Mobilgerät ausgeführt werden und jede weitere Partition steht für einen Offloading-Server und beinhaltet die Tasks, die auf dem jeweiligen Server ausgeführt werden sollen. Traditionelle Partitionierungsalgorithmen stoßen hier allerdings an ihre Grenzen, da sie nur die Gewichtung an den Kanten aber nicht die der Knoten berücksichtigen und so die unterschiedlichen Kosten für die entfernte und die lokale Ausführung eines Tasks nicht berücksichtigen. Für einen Algorithmus, der die Offloading-Entscheidung trifft, sind folgende Faktoren von Relevanz:

- **Gewichtung**
  Für die Offloading-Entscheidung müssen die Gewichte der Knoten und Kanten bestimmt werden. Dafür müssen, je nach Anwendungsgebiet, Faktoren wie Speicherbedarf, *Central Processing Unit* (CPU)-Zeit und Bandbreite einbezogen werden.
- **Echtzeit**
  Durch die sich verändernde Bandbreite eines kabellosen Gerätes ist der Einsatz von statischen Partitionierungsalgorithmen nicht geeignet. Der Algorithmus muss sich den sich ständig verändernden Eigenschaften anpassen können.
- **Effizienz**
  Besonders für komplexere Anwendungen, die eine hohe Anzahl an Tasks mit der Möglichkeit zum Offloading besitzen, ist ein effizienter Partitionierungsalgorithmus nötig. Da die Partitionierung eines Graphen bekanntlich ein NP-schweres Problem ist, lässt es sich nicht in effizienter Zeit lösen. Um die Echtzeit-Fähigkeit zu gewährleisten, muss deshalb ein Approximationsalgorithmus zum Einsatz kommen.

[WSS+15]

Im folgenden werden zwei aktuelle Algorithmen näher erläutert. Beide Algorithmen partitionieren einen Graphen, dessen Knoten für Offloading-Tasks und dessen Kanten die Schnitt-

stellen zwischen den Tasks darstellen, so dass j,e nach Partition, ein Task auf einen bestimmten Server oder auf dem Mobilgerät ausgeführt wird.

2.4.1 *Min-Cost Offloading Partitioning* Algorithmus

Im folgenden wird der von Huaming Wu und Katinka Wolter et al. entwickelte, *Min-Cost Offloading Partitioning* Algorithmus (MCOP-Algorithmus) nach [WSS+15] näher erläutert:

Dieser Algorithmus benötigt als Eingabe einen Graphen, der die Tasks mit ihren oben genannten Gewichtungen repräsentiert und berechnet daraus zwei Partitionen. In einer Partition befinden sich alle durch Knoten repräsentierten Tasks, die auf dem Mobilgerät ausgeführt werden, und in der anderen Partition alle, die auf dem Server ausgeführt werden.

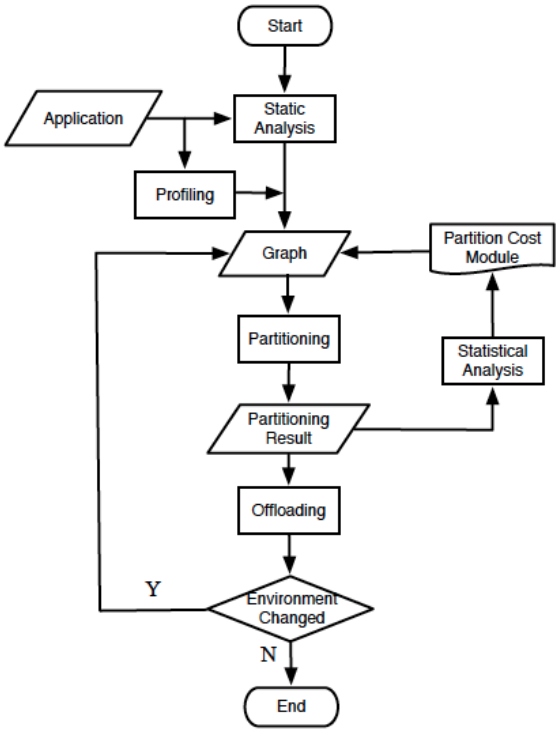


Abbildung 2.3: Flussdiagramm des Partitionierungsprozesses [WSS+15]

Abbildung 2.3 zeigt den Ablauf des Partitionierungsprozesses. Zuerst muss die zugrunde liegende Anwendung einer statischen Analyse unterzogen werden. In dieser Analyse werden die einzelnen Tasks mit den lokalen und entfernten Kosten identifiziert. Auch die Transferkosten zwischen den einzelnen Tasks müssen in diesem Arbeitsschritt definiert werden. Anhand der Resultate wird dann der Graph generiert, der als Eingabe für den eigentlichen Algorithmus dient. Da die Topologie des resultierenden Graphen je nach Anwendung unterschiedlich ausfallen kann, ist der Algorithmus in der Lage, mit allen in Abbildung 2.4 dargestellten Typen umzugehen.

Nun folgt der eigentliche Algorithmus, der den Graphen $G = (V, E)$, wobei V die Menge der Knoten und E die Menge der Kanten bezeichnet, in die zwei Partitionen teilt. Er besteht

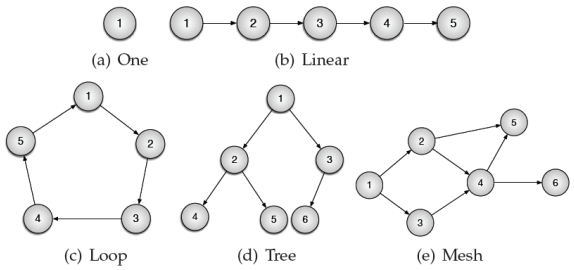


Abbildung 2.4: Verschiedene Topologien des Eingabe Graphen [WSS+15]

aus den folgenden zwei Schritten:

1. **Knoten vereinigen**
   Bestimmte Aufgaben können nicht außerhalb des Mobilgerätes ausgeführt werden. Zum Beispiel Aktionen, die in direkter Verbindung zur grafischen Benutzeroberfläche stehen. Aber auch andere Aktionen, von denen festgelegt wurde, dass sie nicht auf dem Server ausgeführt werden können oder sollen. Alle Knoten, die solche Tasks repräsentieren, werden zu einem Knoten zusammengefasst. Ihre Gewichte werden addiert und dieser Knoten dient dem Algorithmus als Einstiegspunkt.
2. **Schrumpfen des Graphen**
   In diesem Schritt wird der Graph nach und nach um einen Knoten verkleinert in dem zwei Knoten vereinigt werden. Dafür hat der Algorithmus $|V| - 1$ Phasen. In jeder Phase i (mit $1 \leq i \leq |V| - 1$) wird der Schnitt-Wert des Graphen $G_i = (V_i, E_i)$ mit der Formel 2.2 berechnet. Der Graph G_{i+1} für die nächste Phase entsteht dann durch das Vereinigen zweier Knoten in Graph G_i. Nach der letzten Phase wird das Minimum aller Schnitt-Werte jeder Phase i gesucht. Die dazugehörige Menge für die lokale und die entfernte Ausführung bilden das Ergebnis des Algorithmus.

Im Detail verläuft Phase 2 in den folgenden 5 Schritten:

1. Sei A die Menge der Knoten, die auf dem Mobilgerät ausgeführt wird. Beginne mit $A = \{a\}$, a ist dabei ein Knoten, der nicht auf dem Server ausgeführt werden kann.
2. Iterativ werden nun die Knoten, die A am nächsten sind zu A hinzugefügt.
3. s, t seien die zuletzt zu A hinzugefügten Knoten.
4. Der Schnitt von Phase i ist $(V_i \backslash \{t\}, \{t\})$
5. G_{i+1} entsteht aus G_i durch das Zusammenführen von Knoten s und t.

Seien $w^{local}(t)$ die lokalen Kosten des Knotens t und $w^{cloud}(t)$ die entfernten Kosten des Knotens t, dann lassen sich Kosten einer Phase bzw. einer möglichen Partitionierung $C_{cut(A-t,t)}$ mit der folgenden Formel berechnen:

$$C_{cut(A-t,t)} = C^{local} - \left[w^{local}(t) - w^{cloud}(t)\right] + \sum_{v \in A \backslash t} w(e(t,v)) \tag{2.2}$$

wobei $C^{local} = \sum_{v \in V} w^{local}(v)$ die lokalen Gesamtkosten, $w^{local}(t) - w^{cloud}(t)$ der Nutzen falls

Knoten t auf dem Server ausgeführt wird und $\sum_{v \in A \setminus t} w(e(t,v))$ die extra Transferkosten bei einer möglichen entfernten Ausführung sind.

2.4.2 *Adaptive (k+1) Multi-Constraint Partitioning* Algorithmus

Der *Adaptive (k+1) Multi-Constraint Partitioning* Algorithmus ((k+1)-Algorithmus) von Shumao Ou, Kun Yang und Antonio Liotta wird hier nach [OYL06] näher erläutert:

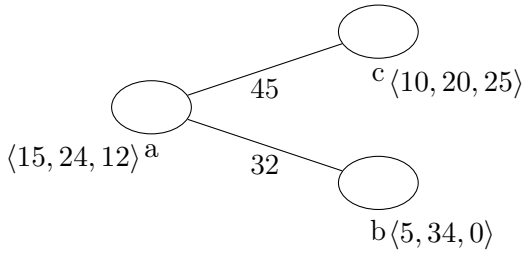
Dieser Algorithmus benötigt als Eingabe einen *dynamic multi-cost* Graph (DMC Graph) $G = (V, E)$ und einen nicht-negativen Integer k. Ziel ist es, den Graphen in $k + 1$ Partitionen aufzuteilen. Eine Partition beinhaltet alle Offloading-Tasks, die nicht auf einem entfernten Server ausgeführt werden können und k disjunkte Partitionen mit den Tasks, die auf k verschiedenen Servern ausgeführt werden. Dabei repräsentiert ein Knoten $v \in V$ ein Task der Anwendung, jeder Knoten v besitzt Gewichte in Form eines Tupels $\langle w_1, w_2, ..., w_n \rangle$ mit n Elementen. Das Gewichte-Tupel repräsentiert die Kosten, die einem Knoten zugeordnet werden. So können einem Knoten verschiedene Arten von Kosten zugeordnet werden. Zum Beispiel ein Tupel $\langle w_1, w_2, w_3 \rangle$, in dem w_1 für die Speichernutzung, w_2 für die akkumulierte Ausführungszeit und w_3 für die Bandbreitennutzung, falls die Komponente über das Netzwerk aufgerufen wird, steht. Um die Komplexität der Berechnung zu verringern, kann das Kosten-Tupel eines Knotens mit folgender Formel zusammengefasst werden:

$$w^v_{composite} = \sum_i \varepsilon_i w^v_i \tag{2.3}$$

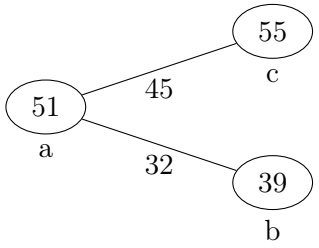
Dabei ist $i = 1, 2, ..., n$ die Anzahl an Kosten in einem Tupel und ε_i steht für einen Gewichtungsfaktor, mit dem die Kosten unterschiedlich gewichtet werden können.

Eine Kante $e \in E$ steht für die Häufigkeit, mit der eine Interaktion zwischen zwei Komponenten stattfindet. Wenn zum Beispiel Komponente v_i zwei verschiedene Methoden jeweils einmal in Komponente v_j aufruft und v_j auf eine Datenstruktur in v_i zugreift, dann beträgt das Gewicht der Kante zwischen den beiden Komponenten 3.

Abbildung 2.5(a) zeigt ein Beispiel für einen Graph ohne zusammengefasstes Gewichte-Tupel und Abbildung 2.5(b) denselben Graphen mit zusammengefasstem Gewichte-Tupel.



(a) 3-Tupel Gewichte



(b) Zusammengefasste Gewichte mit $\varepsilon_1 = \varepsilon_2 = \varepsilon_3 = 1$

Abbildung 2.5: Beispiele für *multi-cost graph*

Der Algorithmus selbst besteht aus drei Schritten:

1. **Knoten vereinigen**
   Genau wie in dem MCOP-Algorithmus werden auch in diesem Algorithmus die Tasks, die nicht auf einem Server ausgeführt werden können, im ersten Schritt vereinigt. Das Kosten-Tupel des neuen Knotens wird dabei aus der Summe aller Kosten-Tupel der vereinigten Knoten berechnet. Die Kanten aller vereinigten Knoten werden ebenfalls vereinigt, indem die Gewichte summiert werden.
2. **Schrumpfen des Graphen**
   Sei G_0 der original Graph nach Schritt 1 mit den vereinigten Knoten. In Schritt 2 wird der Graph mit Hilfe des unten beschriebenen *heavy-edge and light-vertex matching* Algorithmus (HELVM-Algorithmus) zu dem kleinstmöglichen Graph G_i geschrumpft, so dass $|G_i| = k + 1$ ist und die Summe aller Gewichte der Kanten, dessen adjazente Knoten in verschiedenen Partitionen liegt, minimal ist. Außerdem darf die Summe der Knoten-Gewichte bestimmte, vorher definierte Schwellenwerte nicht über- bzw. unterschreiten.
3. **Verfeinerung der Partitionen (optional)**
   Im dritten Schritt lassen sich die Partitionen noch verbessern. Die Idee dabei ist, Knoten, die an den Grenzen der Partitionen liegen, in die jeweils andere Partition zu verschieben. Wenn die Verschiebung die Summe aller Gewichte der Kanten, dessen adjazente Knoten in verschiedenen Partitionen liegt, verringert und die definierten Schwellenwerte nicht verletzt werden, wird sie durchgeführt. Um einen Overhead auf ressourcenarmen Mobilgeräten zu verhindern, kann dieser Schritt auch ausgelassen werden.

Der in Schritt 2 zur Anwendung kommende HELVM-Algorithmus sucht für den Graph G_i nach einem Matching. Ein Matching eines Graphen ist eine Teilmenge von Kanten, in der keine zwei Kanten inzident zu ein- und demselben Knoten sind. Basierend auf klassischen Lösungen dieser Problemstellung entsteht der Graph G_{i+1}, indem ein Match in G_i gesucht wird und die zugehörigen Knoten vereinigt werden. Die meisten mit Gewichten arbeitenden Algorithmen beachten bei der Suche nach einem Match jedoch nur die Kantengewichte. Für das Offloading-Szenario werden im HELVM-Algorithmus auch die Gewichte der Knoten berücksichtigt. Dazu werden Kanten- und Knotengewichte mit folgender Formel zusammengefasst:

$$CW(u, v) = \lambda_1 w(e_{uv}) + \frac{\lambda_2}{w^v} \tag{2.4}$$

Dabei ist $w(e_{uv})$ das Kantengewicht der Kante e_{uv} und w^v das Knotengewicht nach Formel 2.3. Mit Hilfe von λ_i (mit $i = 1, 2$; $0 \leq \lambda_1, \lambda_2 \leq 1$ und $\lambda_1 + \lambda_2 = 1$) können die Knoten- bzw. Kantengewichte unterschiedlich gewichtet werden. Ein Knoten v mit dem maximalen, zusammengefassten Gewicht $CW(u, v)$ wird mit dem Knoten u vereinigt. Sollte es mehr als ein Maximum geben, werden die Gewichte der Kanten zu den anderen Nachbarknoten von v betrachtet, die ihrerseits mit u verbunden sind. Ausgewählt wird dann der Knoten mit der maximalen Summe der Gewichte dieser Kanten.

2.4.3 Schlussfolgerung

Beide Algorithmen lösen das Offloading-Entscheidungsproblem durch das Schrumpfen des Graphen in vertretbarer Zeit. Auch zeigen die Experimente in den beiden zugrundeliegenden Papieren [WSS+15] und [OYL06], dass durch den Einsatz von Offloading mit dem jeweiligen Algorithmus eine Zeitersparnis und eine geringere CPU-Zeit und damit auch geringerer Energieverbrauch erreicht werden kann. Da beide Algorithmen mit verschiedenen Anwendungen evaluiert wurden, ist ein direkter Vergleich und damit eine Aussage, welcher der beiden Algorithmen bessere Ergebnisse erzielt, allerdings schwer möglich.

Der offensichtlichste Unterschied der beiden vorgestellten Algorithmen liegt in der Anzahl der Partitionen, die erstellt werden. Während der (k+1)-Algorithmus eine beliebige Anzahl an Partitionen für eine beliebige Anzahl an Offloading-Servern finden kann, berechnet der MCOP-Algorithmus genau zwei Partitionen. Für ein verteiltes System mit mehr als einem Server ist der MCOP-Algorithmus also nicht geeignet. Ein weiterer Unterschied liegt in der Modellierung der Kosten. Während der MCOP-Algorithmus drei Kostenarten (lokale Kosten, entfernte Kosten und Transferkosten) berücksichtigt, ist es mit dem (k+1)-Algorithmus durch die Verwendung von Kosten-Tupeln möglich, die Anzahl und Art der Kosten selbst zu bestimmen. Der (k+1)-Algorithmus ist in dieser Hinsicht also flexibler als der MCOP-Algorithmus.

Bis vergleichbare Werte zu Energie-, Zeit- und Speicherersparnissen vorliegen, hängt eine Entscheidung, welcher dieser beiden Algorithmen zum Einsatz kommen sollte, also in erster Linie von den Gegebenheiten der Server-Client-Architektur und der Modellierung der Kosten ab.

2.5 Mobile Offloading Systeme

In diesem Abschnitt werden verschiedene Offloading Systeme vorgestellt und miteinander verglichen.

2.5.1 *Android mobile devices computation offloading project*

Zu Forschungszwecken und um es Entwicklern zu ermöglichen, ihre Anwendungen ohne viel Aufwand um die Offload-Funktionalität erweitern zu können, wurde ein modulares und leicht erweiterbares Offloading-System entwickelt. Das in Java implementierte *Android mobile devices computation offloading project* (AMDCOP) besteht aus einem Framework für den Client und einem Server. Im folgenden soll dieses System nach [Gri13] und [Rip13] näher erläutert werden:

Der Entwickler muss seine Anwendung manuell partitionieren, in dem er rechenintensive Berechnungen identifiziert und diese in einem Java-Archiv (JAR) zur Verfügung stellt. Dieses JAR wird im Laufe der Entwicklung auf dem Server installiert. Eine Übertragung der Tasks zur Laufzeit der Anwendung würde einen unnötigen Overhead produzieren und wäre deshalb nicht sinnvoll. Allerdings bedeutet die manuelle Installation der Funktionalität auf dem Server auch einen Mehraufwand für die Entwickler.

Das System besteht aus einem Server und einem Client:

Server

Der Server bietet eine Weboberfläche mit der Entwickler Teile ihrer Anwendung, die potentiell von dem Server ausgeführt werden sollen, als JAR hochladen können. Das JAR muss in einer Klasse Methoden mit bestimmten Headern bereitstellen, die vom System erkannt werden und dann als Einstiegspunkt für die einzelnen Tasks dienen. Der Entwickler muss dabei die Partitionierung seiner Anwendung per Hand vornehmen. Da der Server auf einem Apache Tomcat Server läuft, lässt er sich, wie auch Android-Anwendungen, in Java programmieren. Das hat den Vorteil, dass der Entwickler keine zwei komplett verschiedenen Versionen seiner Offloading-Tasks programmieren muss, sondern ein und denselben Code für Server und Mobilgerät nutzen kann. Da für Android-Anwendungen jedoch Java-Bytecode in Dalvik-Bytecode übersetzt wird, während auf dem Server Java-Bytecode direkt ausgeführt wird, gilt es zu beachten, dass keine Android System-Bibliotheken auf dem Server verwendet werden können, sondern nur Java System-Bibliotheken.

Außerdem beinhaltet der Server ein System, um eine initiale Datenbank mit den Laufzeiten der Offloading-Tasks zu erstellen. Über die Weboberfläche lässt sich eine mit den Eingabeparametern eines Tasks vorher angefertigte csv-Datei hochladen. Daraus wird eine Datenbank generiert, die dann vom Entwickler in seine Android-Anwendung integriert wird und der Berechnung der durchschnittlichen Laufzeit eines Offloading-Tasks dient. Im weiteren Verlauf dieses Kapitels wird dieses System näher erläutert.

Die Anfrage des Clients an den Server erfolgt via HTTP. Eine Anfrage beinhaltet dabei den Namen und die Eingabeparameter des auszuführenden Tasks als String. Der Server schickt dann die Antwort im *Extensible Markup Language* (XML) Format zurück an den Client, welcher das Ergebnis auslesen kann.

Client

Der Client besteht aus einer kleinen Software-Bibliothek, bestehend aus drei Java-Klassen, die vom Entwickler in die Anwendung eingebunden wird. Nach einigen Anpassungen der Anwendung trifft diese Bibliothek eigenständig die Offloading-Entscheidung, verwaltet die Verbindung zum Server und behandelt die Antwort des Servers.

Offloading-Entscheidung

Die Entscheidung, ob ein Task auf dem Mobilgerät oder auf dem Server ausgeführt wird, trifft der Client während der Laufzeit. Da der Energieverbrauch auf Android-basierten Smartphones nicht erfasst werden kann, aber mit einer zeitlichen Entlastung des Prozessors auch eine Energieersparnis einhergeht [Gri13, S. 28f], basiert die Entscheidung auf folgender Ungleichung für Zeitersparnis:

$$\frac{I}{F \cdot M} + R + \frac{D_s}{B_u} + \frac{D_r}{B_d} < \frac{I}{M} \tag{2.5}$$

Dabei ist I die Anzahl an Instruktionen des potentiell vom Server auszuführenden Tasks, M die Anzahl der Instruktionen, die das Mobilgerät pro Sekunde ausführen kann, F der Faktor, um den der Server schneller als das Mobilgerät ist, R die Summe aus Round-Trip-Time und Zeit, die für Verbindungsaufbau benötigt wird, D_s die Größe der Daten die zur Berechnung versendet werden müsste, B_u die für den Upload zur Verfügung stehende Bandbreite, D_r die Größe der Daten, die nach der Berechnung empfangen werden würde und B_d die für den Download zur Verfügung stehende Bandbreite. Da es nicht möglich ist, vor dem Offloading-Vorgang zu erfassen wie groß D_r ist und angenommen wird, dass es sich immer um eine vernachlässigbar kleine Datenmenge handelt, wird die Formel zu folgender vereinfacht:

$$\frac{I}{F \cdot M} + R + \frac{D_s}{B_u} < \frac{I}{M} \tag{2.6}$$

Ein Offloading-Vorgang wird durchgeführt wenn die Formel nach der Auswertung wahr ist. Die Parameter werden dabei wie folgt berechnet:

- F**:**
  Beim ersten Start der Anwendung auf dem Mobilgerät wird ein einfacher Algorithmus ausgeführt. Die Ausführungszeit desselben Algorithmus für den Server ist bekannt. Daraus lässt sich F berechnen. Da F aufgrund der Ressourcenverwaltung des Betriebssystems je nach Zeitpunkt und Art des auszuführenden Tasks unterschiedlich ausfallen kann, werden zur Erhöhung der Genauigkeit im Laufe der Zeit für jeden Task mehrere Werte für F berechnet und der Durchschnittswert verwendet. Bei jeder Ausführung eines Tasks wird das dazugehörige Set an Werten für F aktualisiert.
- R**:**
  Bei jedem Start werden mehrere Anfragen an den Server geschickt. Der Mittelwert der gemessenen Round-Trip-Times ergibt R. Wenn die Verbindung zum Server unterbrochen wurde oder sich die Art der Netzanbindung ändert (zum Beispiel von 3G zu WLAN), wird der Vorgang wiederholt.
- B_u**:**
  Bei jedem Start der Anwendung wird eine kleine Datei zum Server geschickt. Daraus lässt sich die zur Verfügung stehende Bandbreite errechnen. Bei Veränderungen am Netz wird dieser Vorgang wiederholt.
- D_s**:**
  Sobald die Eingabe eines Tasks bekannt ist, kann die Größe der Eingabe direkt ausgelesen werden.
- I**:**
  Dieser Wert wird vom Entwickler durch eine Kostenfunktion bereitgestellt. Sollte das nachfolgend erläuterte *Automated estimation system of task execution times* (AESTET) zum Einsatz kommen, wird I nicht benötigt.

Automated estimation system of task execution times

Für viele potentielle Offoading-Tasks ist es auf Grund erhöhter Komplexität nicht möglich eine Kostenfunktion auf Basis von Anzahl der Instruktionen zu finden. Das AESTET er-

leichtert die Kostenberechnung, indem es die benötigte Zeit zur Bearbeitung eines Tasks als Kosten verwendet. Es beruht auf der Annahme, dass die Bearbeitung eines Tasks bei selber Eingabe die selbe Bearbeitungszeit benötigt und bei ähnlicher Eingabe eine ähnliche Bearbeitungszeit. Dafür muss vom Entwickler eine Funktion bereitgestellt werden, die die Eingabeparameter eines Tasks in einen numerischen Wert übersetzt. Mit Hilfe dieser Eingabe-ID wird eine Datenbank aufgebaut, in der vergangene Ausführungszeiten jedes Tasks der jeweiligen Eingabe zugeordnet werden. Bei einem neuen Offloading-Vorgang wird anhand der Eingabeparameter und mit Hilfe des unten beschriebenen *k-Nearest Neighbor smoothing* (KNN smoothing) eine durchschnittliche Ausführungszeit ermittelt.

Um das Mobilgerät zu entlasten, wird während der Entwicklung mit Hilfe der Weboberfläche des Servers eine initiale Datenbank mit vom Entwickler bereitgestellten Beispieleingaben für die Tasks aufgebaut. Diese Datenbank wird dann mit der fertigen Anwendung ausgeliefert und während der Ausführung der Anwendung um weitere Ausführungszeiten erweitert. Da die Datenbank zu Anfang nur einige initiale Einträge umfasst, durchläuft das System zu Beginn eine Trainingsphase. Mit der Zeit wird die Abschätzung immer genauer.

Die Datenbank ist wie folgt aufgebaut:

inputRepresentation	*executionTime*	*isLocal*

Tabelle 2.1: Datenbank des AESTET

inputRepresentation ist die oben erwähnte Eingabe-ID, *executionTime* die Zeit, die für die Ausführung des Tasks mit der jeweiligen Eingabe benötigt wurde und *isLocal* ein Boolean, der angibt, ob die Ausführungszeit für das Mobilgerät oder den Server gilt. Für jeden Offloading-Task gibt es eine eigene Tabelle 2.1.

Anhand der Eingabe-ID werden alle passenden Laufzeiten ausgelesen und mit Hilfe des KNN smoothing eine möglichst genaue Ausführungszeit ermittelt. Der Algorithmus verläuft dabei in folgenden Schritten:

1. Sobald ein Offloading-Task T ausgeführt werden soll, wird mit Hilfe der vom Entwickler bereitgestellten Funktion für die Eingabeparameter die Eingabe in eine numerische Eingabe-ID übersetzt.
2. In der zum Offloading-Task gehörende Tabelle wird nach den k nächsten Nachbarn der Eingabe-ID gesucht.
3. Für jedes dieser k Elemente werden die Attribute *executionTime* und *isLocal* ausgelesen. Sollte *isLocal* wahr sein, wird *executionTime* durch F dividiert, sodass ausschließlich die Ausführungszeiten des Servers vorliegen.
4. Die Ausführungszeiten werden gemittelt und Ausreißer eliminiert (*smoothing*). Falls ein oder mehrere Elemente mit der selben Eingabe-ID wie der von T vorhanden sind, wird aus diesen ein neuer Durchschnittswert *AvgSurrogate* errechnet und mit Schritt 6 fortgefahren.
5. Anhand des Datensatzes wird ein neuer Durchschnittswert *AvgSurrogate* errechnet. Dabei werden Elemente mit einer Eingabe-ID, die näher an der Eingabe-ID von T sind, höher gewichtet.

6. *AvgSurrogate* ist die für den Server ermittelte Ausführungszeit. Multipliziert man diesen Wert mit F, ergibt sich die Ausführungszeit für das Mobilgerät.
7. Nach der Ausführung von T wird die tatsächliche Ausführungszeit zur Datenbank hinzugefügt. Damit die Datenbank nicht zu groß wird, werden maximal 20 Einträge mit derselben Eingabe-ID behalten.

2.5.2 *CloneCloud*

Einen vollkommen anderen Ansatz verfolgt *CloneCloud*. Im folgenden wird das System nach [CIM+11] näher erläutert:

Die zugrunde liegende Idee besteht darin, eine vollständige, virtuelle Kopie des Mobilgerätes auf dem Server zu simulieren und dort Teile einer Anwendung ohne weitere Modifikationen auszuführen. Möglich wird das durch die betriebssystemunabhängige Ausführung von Bytecode in einer Virtuellen Maschine (VM). Java-, Android und Microsofts .Net laufen in einer VM, wodurch Anwendungen für diese Plattformen sich relativ leicht auf unterschiedlichen Systemen ausführen lassen.

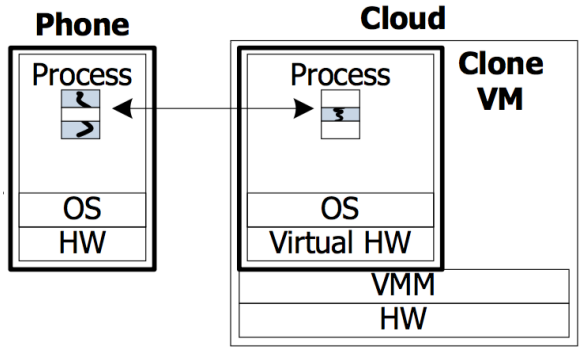


Abbildung 2.6: *CloneCloud* System Modell [CIM+11]

Identifikation der Offloading-Tasks

Anders als das zuvor vorgestellte System nimmt *CloneCloud* die Identifikation der Offloading-Tasks durch statische und dynamische Analyse automatisch vor. Die Granularität, in der eine Anwendung aufgeteilt wird, liegt dabei auf Thread-Ebene. Die statische Analyse identifiziert zulässige Partitionen anhand festgelegter Bedingungen und setzt im Code Aus- und Eingangspunkte für die entfernte Berechnung auf einem Server. Diese Punkte befinden sich an Methodenein- und Methodenausgängen und können nur an Methoden gesetzt werden, die zur Anwendung selbst gehören. Systembibliotheken werden ausgeschlossen. Dadurch wird die Implementierung während der Laufzeit vereinfacht. Des Weiteren dürfen die Ein- und Ausstiegspunkte nur an Methodenein- und Methodenausgängen gesetzt werden, die innerhalb der VM laufen. Nativer Code, der außerhalb der VM externe Prozesse, wie Datei Ein-/ Ausgabe, Netzwerkmanagement oder den Zugriff auf Sensoren steuert, darf auf der entfernten Maschine aufgerufen werden, ein Übergang vom Mobilgerät zum Server innerhalb

nativen Codes ist jedoch nicht zulässig. Eine zulässige Partition muss die folgenden Bedingungen erfüllen:

1. Methoden, die lokale Ressourcen, wie zum Beispiel GPS, Kamera oder Mikrophone nutzen, können nicht auf dem Server ausgeführt werden.
2. Verschiedene native Methoden, die auf native Zustände außerhalb der VM zugreifen, müssen auf der selben Maschine ausgeführt werden.
3. Keine verschachtelte Migration: Während ein Task auf dem Server ausgeführt wird, darf kein weiterer Task auf dem Server ausgeführt werden. Erst wenn das Ergebnis des Ersten empfangen wurde, darf der Zweite auf dem Server ausgeführt werden.

Automatische Kostenschätzung

Auch die Kosten werden vom System automatisch bestimmt. Dazu wird die Anwendung mehrfach ausgeführt und mit Hilfe eines Profilers bezüglich ihres Energieverbrauches und Zeitverbrauches untersucht. Das Resultat sind zwei *Profile Trees* T und T' pro Ausführung mit den Kosten für das Mobilgerät und den Server. Ein *Profile Tree* ist eine kompakte Repräsentation einer Ausführung auf einer Plattform. Jeder Knoten steht dabei für einen Methodenaufruf. Der Elternknoten eines Knotens gibt an, von wo die Methode aufgerufen wurde. Außerdem hat jeder Knoten $v \in \{T, T'\}$ ein Blatt v' das die Kosten für eine Ausführung des Knotens v ohne die Kosten der von v aufgerufenen Methoden repräsentiert. An den Kanten stehen die Kosten für eine Auslagerung an den Server. Abbildung 2.7 zeigt so einen *Profile Tree*. Mit Hilfe der beiden *Profile Trees* T und T' und den oben beschriebenen Einschränkungen wird dann ein hybrider Baum aus T und T' mit minimalen Kosten gebildet, woraus sich wiederum ergibt, welche Berechnungen auf dem Mobilgerät und welche auf dem Server ausgeführt werden.

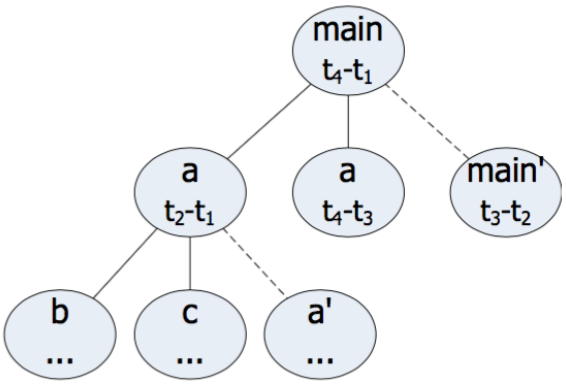


Abbildung 2.7: *CloneCloud Profile Tree* mit den Methoden *main, a, b* und *c*, ohne Kantenkosten [CIM+11]

Offloading-Entscheidung

Wird eine partitionierte Anwendung ausgeführt, wird anhand von Ausführungsbedingungen, wie die Erreichbarkeit des Servers und die Qualität der Netzanbindung, eine passende Partitionierungskonfiguration in einer Datenbank gesucht. Spezielle VM Instruktionen versehen die Methoden anhand der gewählten Partitionierungskonfiguration mit den Ein- und Ausgängen für die entfernte Ausführung. Wenn einer der Ausgangspunkte im Programmablauf erreicht wird, wird der ausführende Thread unterbrochen und sein Zustand inklusive Stack und Register an den Server geschickt. Dort wird ein neuer Thread mit den empfangenen Zuständen erstellt und ausgeführt. Nach Erreichen des Endpunktes wird der neue Zustand wieder an das Mobiltelefon migriert.

2.5.3 *MAUI*

Hier wird das Offloading-System *MAUI* nach [CBC+10] näher erläutert:

MAUI stellt Entwicklern eine Programmierumgebung zur Verfügung, in der Methoden, die potentiell auf einem Server ausgeführt werden sollen, markiert werden. Jedes Mal, wenn eine dieser Methoden aufgerufen wird, entscheidet *MAUIs* Optimierungs-Framework, ob sie auf dem Mobilgerät oder auf dem Server ausgeführt wird. Nach der Ausführung werden Daten gesammelt, die zukünftige Entscheidungen verbessern. Die Kosten, die für einen Offloading-Vorgang anfallen, werden durch eine Instrumentierung der Methoden und Profiling ermittelt. Außerdem wird die Qualität des Netzwerkes kontinuierlich überwacht, um Bandbreite und Latenz zu schätzen. Auf Grundlage dieser Daten entscheidet *MAUI*, ob eine Methode lokal oder auf dem Server ausgeführt wird. Sollte während eines Offloading-Vorgangs die Verbindung zum Server abbrechen, sucht sich das System nach einem Timeout einen neuen Server oder führt den Task lokal erneut durch.

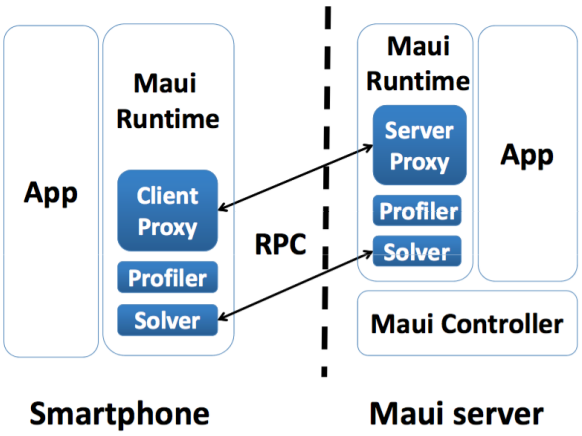


Abbildung 2.8: *MAUI* Architektur [CBC+10]

Abbildung 2.8 zeigt die Architektur von *MAUI*. Auf dem Mobilgerät finden sich drei Komponenten: 1. ein Interface zu der auf dem Server laufenden Entscheidungs-Engine, 2. ein Proxy, der die Verbindung zum Server verwaltet und 3. ein Profiler, der die Anwendung instrumentiert und Messungen zum Energieverbrauch und Datentransfer durchführt. Auf Seiten des

Servers gibt es vier Komponenten: Ein Profiler und ein Proxy, die ähnliche Aufgaben vollziehen wie ihre Gegenstücke auf dem Client, die Entscheidungs-Engine, die regelmäßig das Offloading-Entscheidungsproblem löst, und der *MAUI-coordinator*, der die Authentifizierung und die Ressourcen für eingehende Offloading-Anfragen verwaltet.

MAUI unterstützt Anwendungen, die für Microsofts *Common Language Runtime* (CLR) geschrieben wurden. CLR unterstützt die Programmierung in verschiedenen Programmiersprachen, die dann zu der *Common Intermediate Language* (CIL) kompiliert werden. Eine ausführbare Anwendung enthält CIL Instruktionen, die zur Ausführungszeit dynamisch, dem System entsprechend, kompiliert werden. Dadurch wird die für das Offloading notwendige Plattformunabhängigkeit erreicht.

Identifikation der Offloading-Tasks

Der Entwickler übernimmt die Identifikation der Offloading-Tasks selbst indem die Methoden und Klassen, die potentiell auf dem Server ausgeführt werden sollen, mit einem *remoteable* markiert werden. Dieses Attribut ist auch in der ausführbaren .NET CLR Anwendung enthalten und kann mit Hilfe der .NET *Reflection API* ausgelesen werden. Wie auch bei den anderen Offloading-Systemen darf bestimmter Code nicht auf dem Server ausgeführt werden: Code, der die Benutzeroberfläche implementiert, Code, der auf Sensoren des Smartphones zugreift und Code, der mit externen Komponenten interagiert, die bei wiederholter Ausführung fehlerhaftes Verhalten aufweisen könnten. Ein Beispiel für eine solche Komponente ist eine *E-Commerce*-Transaktion wie das Kaufen eines Artikels in einem Online Shop. Der Entwickler muss sich bei der *remoteable* Markierung keine großen Gedanken über den möglichen Vorteil einer entfernten Ausführung einer Klasse oder Methode machen. Viel mehr sollten alle Methoden und Klassen, die nicht zu eine der drei oben genannten Kategorien gehören, als *remoteable* markiert werden und *MAUI* entscheidet zur Laufzeit ob eine Auslagerung des Codes sinnvoll ist oder nicht.

Für jede Methode, die mit dem Attribut *remoteable* versehen wird, generiert *MAUI* ein Wrapper, der den Originalparametern zwei weitere Parameter hinzufügt: Ein Eingabeparameter, der den für die Ausführung auf dem Server nötigen Anwendungszustand erfasst und ein Ausgabeparameter, mit dessen Hilfe der Anwendungszustand wieder zurück zum Mobilgerät migriert wird.

Offloading-Entscheidung

Die Entscheidung, ob ein Task auf dem Mobilgerät oder auf dem Server ausgeführt wird, hängt von folgenden drei Faktoren ab:

1. Energieverbrauch des Mobilgerätes
   Die Entwickler von *MAUI* nutzten ein Hardware-Leistungsmessgerät, um den Energieverbrauch des Mobilgerätes zu protokollieren. So wurde ermittelt, wie viel Energie pro Prozessor Zyklus verbraucht wurde. Ein Prozessor-Zyklus entspricht der Zeit, die der Prozessor für eine einfache Operation, wie zum Beispiel einer Addition, benötigt. Anhand dieser Daten können Vorhersagen über den Energieverbrauch pro CPU-Zyklus getroffen werden.

2. Eingenschaften der Anwendung
   Mit einem Profiler wird die Ausführung eines Programmes überwacht. Die Zeit und die Prozessor-Zyklen, die eine Methode für die Ausführung benötigt, werden gemessen. Anhand dieser Daten wird der Energieverbrauch geschätzt.
3. Eigenschaften des Netzwerkes
   Mit Hilfe eines 10KB Datenpaketes, das zum Server geschickt wird, werden die aktuellen Netzwerkeigenschaften ermittelt. Bei jedem Offloading-Vorgang werden die Daten über das Netzwerk aktualisiert. Sollte in einem Zeitfenster von einer Minute kein Offloaing-Vorgang durchgeführt werden, wird erneut ein 10KB Datenpaket gesendet. Die gemittelten Daten von Bandbreite, Latenz und Paketverlust tragen zur Offloading-Entscheidung bei.

Mit Hilfe dieser Daten und einem Call Graph $G = (V, E)$, der den Call Stack der Anwendung repräsentiert, wird die Entscheidung getroffen. Dabei repräsentiert ein Knoten $v \in V$ die Methoden, die sich im Call Stack befinden und jede Kante $e = (u, v) \in E$ ein Aufruf der Methode v von u. Jeder Knoten v hat die Werte E_v^l, für den lokalen Energieverbrauch, T_v^l, für die Zeit, die die Ausführung lokal braucht und E_v^r für die Ausführungszeit auf dem Server. Jede Kante $e = (u, v)$ besitzt die Werte $B_{u,v}$ und $C_{u,v}$. Dabei steht $B_{u,v}$ für die benötigte Zeit und $C_{u,v}$ für den Energieverbrauch, um den Anwendungszustand zu übertragen. Jeder Knoten hat außerdem die Variable r_v, die angibt, ob der Knoten als *remoteable* markiert ist. Das folgende 0-1 Integer Lineares-Programmierungsproblem wird für eine Entscheidung gelöst:

$$\text{maximiere} \sum_{v \in V} I_v \times E_l^v - \sum_{(u,v) \in E} |I_u - I_v| \times C_{u,v} \tag{2.7}$$

$$\text{so dass} \sum_{v \in V} ((1 - I_v) \times T_v^l) + (I_v \times T_v^r) + \sum_{(u,v) \in E} |I_u - I_v| \times B_{u,v} \leq L \tag{2.8}$$

$$\text{und } I_v \leq rv, \forall v \in V \tag{2.9}$$

I_v ist dabei eine Indikatorvariable, die angibt, ob eine Methode lokal ($I_v = 0$) oder auf dem Server ($I_v = 1$) ausgeführt wird. Der erste Term der Zielfunktion ist die eingesparte Energie, falls die Methode auf dem Server ausgeführt wird. Der zweite Term erfasst die Energiekosten für den Transfer bei einer entfernten Ausführung. Die erste Bedingung schreibt vor, dass die maximale Zeit, die zur Ausführung benötigt wird, L nicht überschreiten darf. Die zweite Bedingung besagt, dass nur Methoden, die als *remoteable* markiert wurden, auf dem Server ausgeführt werden können.

2.5.4 *Cuckoo*

Das vierte zu untersuchende Offloading-System ist *Cuckoo*, benannt nach dem Kuckuck, der seine Eier in fremde Nester legt und somit das Ausbrüten an andere Vögel „auslagert“. Im folgenden wird das System nach [KPKB12] erläutert:

Cuckoo ist ein Offloading-Framework für Android. Der Server selbst hat zunächst keine Funktionalität und kann vom Benutzer auf jedem beliebigen Rechner gestartet werden. Erst zur Laufzeit der Anwendung wird die benötigte Funktionalität nachträglich installiert. Mit

Hilfe eines Ressourcenmanagers, der auf dem Android Mobilgerät läuft, wird die Adresse des Servers erfasst. Dazu wird nach dem Starten des Servers ein *Quick Response Code* (QR-Code) mit der codierten Adresse des Servers angezeigt. Dieser QR-Code wird vom Benutzer mit seinem Mobilgerät eingelesen.

Identifikation der Offloading-Tasks

Die Identifikation der Offloading-Tasks bleibt bei *Cuckoo* dem Entwickler überlassen. Für jeden Task muss eine Implementierung für das Mobilgerät und eine für die entfernte Ausführung vorliegen. Dafür werden rechenintensive und interaktive Teile der Anwendung durch die Schnittstellenbeschreibungssprache *Android Interface Definition Language* (AIDL) voneinander getrennt. Mit AIDL werden Schnittstellen zur Interprozesskommunikation definiert, so dass verschiedene Prozesse miteinander kommunizieren können. Der Entwickler implementiert diese AIDL-Schnittstellen in seiner Anwendung, damit *Cuckoo* mit Hilfe seines *Cuckoo Remote Service Drivers* Dummy-Methoden für die entfernte Ausführung generieren kann. Diese Dummys müssen daraufhin vom Entwickler mit der Implementierung für den Server ausgetauscht werden. Dabei obliegt es dem Entwickler, ob die beiden Versionen unterschiedlich oder identisch sind. Anschließend wird ein JAR, das die Implementierung für den Server beinhaltet, erstellt. Dieses JAR wiederum ist Teil des *Android Package* (APK) für das Mobilgerät.

Offloading-Entscheidung

Mit Hilfe des *Cuckoo Service Rewriter* wird im Laufe des Kompilierungsvorgangs jede, für die lokale Ausführung implementierte AIDL Schnittstelle so umgeschrieben, dass zur Laufzeit der Anwendung entschieden werden kann, ob eine lokale oder entfernte Ausführung vollzogen wird. Die Entscheidung selbst ist vorerst sehr einfach gehalten: Ist der Server erreichbar, wird ein Offloading-Vorgang eingeleitet. Sollte der Offloading-Task dem Server noch nicht bekannt sein, wird das JAR mit der benötigten Funktionalität vom Mobilgerät an den Server geschickt und ist daraufhin für die Zukunft verfügbar. Dabei gibt es zwei mögliche Zeitpunkte, zu denen eine Funktionalität installiert wird: 1. sobald Mobilgerät und Server das erste Mal miteinander kommunizieren und 2. zum Zeitpunkt des Methodenaufrufes. Ersteres bietet einen Zeitvorteil, da alle potentiell benötigten Tasks zum Zeitpunkt der Ausführung schon auf dem Server verfügbar sind. Letzteres bietet Energieeinsparungen, da zwischen dem Zeitpunkt der ersten Kommunikation und der Ausführung eines Tasks die Verbindung zum Server möglicherweise verloren gehen könnte und somit die Installation überflüssig wäre.

2.5.5 Schlussfolgerung

Alle Systeme bieten interessante Ansätze zum Thema Mobiles Offloading. Leider sind die meisten Systeme nicht öffentlich verfügbar und entziehen sich dadurch einer praktischen Untersuchung für diese Arbeit. Rückschlüsse sind ausschließlich auf Grundlage der jeweiligen wissenschaftlichen Papiere zu ziehen.

CloneCloud bietet durch die virtuelle Kopie des Mobilgerätes in der Cloud und das Transferieren des Threadzustandes eine Lösung, die ohne Modifikation der Anwendung funktioniert.

Aus dieser Vorgehensweise folgt jedoch, dass bei einem Offloading-Vorgang der gesamte Zustand eines Threads verschickt werden muss. Im Vergleich zu den anderen Systemen, die nur benötigte Zustände an den Server schicken, entsteht dadurch ein erhöhter Overhead.

Anwendungen die *Cuckoo* oder auch AMDCOP nutzen, haben durch die unterschiedlichen Virtualisierungen auf dem Mobilgerät und dem Server einen höheren Entwicklungsaufwand. Für jeden Task muss sowohl eine Implementierung für den Client als auch für den Server vorhanden sein. Da beide jedoch für das Betriebssystem Android entwickelt wurden und Anwendungen für Android in Java geschrieben werden, dürfte in den meisten Fällen ein ähnlicher, wenn nicht sogar der selbe Code für Mobilgerät und Server verwendbar sein. Trotz allem muss während der Entwicklung sichergestellt werden, dass der Code auf beiden Systemen lauffähig ist. Da bieten *CloneCloud* und *MAUI* einen klaren Vorteil. *MAUI* verlangt vom Entwickler lediglich die Markierung der Methoden, die für das Offloading vorgesehen werden. Durch die Verwendung der CLR auf dem Mobilgerät und dem Server ist die Implementierung eines Tasks auf beiden Systemen lauffähig.

Ein direkter Vergleich der Energie- und Zeitersparnisse der verschiedenen Systeme ist leider nicht möglich. Zu viele Faktoren beeinflussen das Ergebnis: So beeinflusst die Qualität des Netzwerkes die Ergebnisse, zum Beispiel ist der Energieverbrauch bei *MAUI* bis zu 2,5 mal höher, wenn Offloading über 3G statt über WLAN betrieben wird [CBC+10]. Auch kommen in allen Papieren unterschiedliche Anwendungen zum Einsatz. Je nach Art der Anwendung gibt es systemübergreifend aber auch innerhalb eines Systems erhebliche Unterschiede zwischen den Einsparungen. Des Weiteren ist das verwendete Mobilgerät von Bedeutung, ein leistungsschwaches Smartphone profitiert mehr vom Offloading als ein aktuelles Top Modell [Rip13, S. 61]. Ein Vergleich aller Systeme mit derselben Anwendung auf demselben Mobilgerät und den gleichen Netzwerkbedingungen wäre sehr interessant.

Allerdings beweisen alle Systeme, dass Offloading erhebliche Einsparungen bringt: *CloneClode* erreicht Zeit- und Energieersparnisse bis zu einem Faktor von 20 [CIM+11], *MAUI* schafft es, den Energieverbrauch um maximal 45% zu reduzieren und die Latenz einer Spieleanwendung um den Faktor 4,8 zu verbessern [CBC+10], *Cuckoo* verbessert die Ausführungszeit beispielsweise bei einer Anwendung, die Objekte auf Bildern erkennt, um den Faktor 60 und reduziert den Energieverbrauch um den Faktor 40. Gleichzeit wurde die Qualität der Erkennung verbessert [KPKB12]. Bei AMDCOP hängen die Ersparnisse stark von dem AESTET ab. Sollte eine identische Eingabe-ID gefunden werden, liefert das System auf Grundlage der Ausführungszeitabschätzung eine gute Offloading-Entscheidung. Ist das nicht der Fall, wird die Abschätzung jedoch ungenauer und führt dadurch eher zu Fehlentscheidungen. Da das System ständig dazulernt, sollten sich die Schätzungen und somit die Ergebnisse mit der Zeit kontinuierlich verbessern. Bei einem Schachspiel ist mit dem AESTET eine Zeitersparnis von bis zu 72 Sekunden bei der Suche nach den nächsten 20 Schachzügen möglich.[Rip13, S. 63]

Die folgende Tabelle stellt noch einmal die wichtigsten Eigenschaften der vier Systeme gegenüber:

	AMDCOP	***CloneCloud***	***MAUI***	***Cuckoo***
Partitionierung	manuell	automatisch	hybrid	manuell
Task Granularität	Methoden	Threads	Methoden	Methoden
Codemodifizierung	Ja	Nein	Nein[2]	Nein
Verfügbarkeit	verfügbar	nicht verfügbar	nicht verfügbar	nicht verfügbar
Unterschiedliche Implementierungen für Client und Server	Ja	Nein	Nein	Ja
Programmiersprache	Java	Java	.Net CLR kompatibel	Java
Zusätzlicher Entwicklungsaufwand	hoch	gering	mittel	hoch
Offloading Entscheidungskriterien	Netzwerk Profiling, Ausführungszeit, Transferkosten	Netzwerk Profiling, statische Partitionierungskonfiguration	Energieverbrauch, Ausführungszeit, Netzwerk Profiling	Serververfügbarkeit

Tabelle 2.2: Vergleich: Offloading Systeme

[2]Hinzufügen des Attributes *remoteable*

KAPITEL 3

Offloading Schach

3.1 Analyse

In diesem Kapitel wird kurz auf den Aufbau des zu erweiternden Schachspiels *CuckooChess* eingegangen. Im Anschluss werden die für die Verwendung der Offloading-Engine des AMD-COP nötigen Anforderungen an das Schachspiel gestellt.

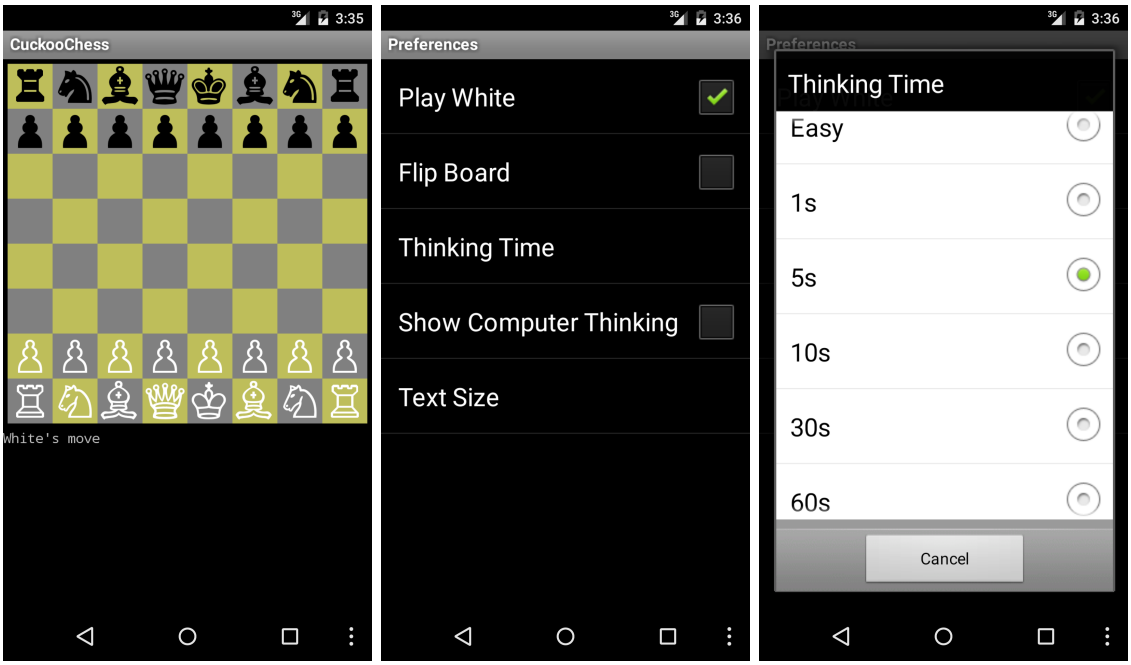
3.1.1 Auswahl des Schachspiels

Ein neues Schachspiel zu entwickeln, würde den Rahmen dieser Arbeit sprengen. Deshalb wird ein schon bestehendes Schachspiel um die Offloading-Funktionalität erweitert. Um diese Erweiterung umzusetzen, muss die Anwendung natürlich Open Source sein. Da das Offloading-Framework in Java für Android-Anwendungen entworfen wurde, muss auch das Schachspiel auf Android laufen. Es gibt zwar viele Schachspiele für Android, jedoch schränkt das Kriterium Open Source die Auswahl erheblich ein. Nach einiger Recherche fiel die Wahl auf das Java Schachprogramm *CuckooChess*, das eine Portierung auf Android mitbringt [Ös].

3.1.2 Übersicht von *CuckooChess*

Das Spiel *CuckooChess* ist in drei Schichten aufgeteilt.

Die erste Schicht *Android UI* stellt Methoden zur Verarbeitung der Eingabe des Benutzers und Methoden zur Darstellung von Daten auf der Benutzeroberfläche bereit. Die Schicht *Controller* ist das Bindeglied zwischen der Benutzeroberfläche und der eigentlichen Schach-Engine. In ihr werden die Benutzeraktionen ausgewertet und die jeweiligen Aktionen an die Schach-Engine weitergeleitet. In *Chess Engine* findet sich die Logik des Schachprogramms inklusive der Künstlichen Intelligenz (KI).



(a) Hauptfenster (b) Einstellungen (c) Schwierigkeitsgrad

Abbildung 3.1: Grafische Oberfläche von *CuckooChess*

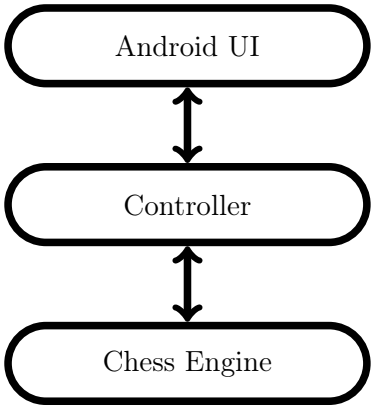


Abbildung 3.2: Schichtenmodell von *CuckooChess*

3.1.3 Anforderungsanalyse

Eine erfolgreiche Erweiterung des Schachspiels um die Offloading-Funktionalität beinhaltet die Erfüllung folgender Anforderungen:

- **Integrierung der Offloading-Engine**
  Das in 2.5.1 beschriebene Offloading-System soll für die Offloading-Funktionalität in das Schachspiel integriert werden.

- **Serialisierung**
  Das Offloading Framework ist so konzipiert, dass die Übertragung der Parameter zwischen Server und mobilem Endgerät als *String* stattfindet. Da einige der Parameter aus komplexen Datentypen bestehen, müssen diese für eine Übertragung zuvor serialisiert werden.
- **Schwierigkeitsgrad**
  In der originalen Version von *CuckooChess* wird der Schwierigkeitsgrad durch ein Zeitlimit umgesetzt. Dabei hat die KI, je nach Schwierigkeitsgrad, maximal 1, 5, 10, 30 oder 60 Sekunden Zeit, um den nächsten Zug zu finden. Sollte der Suchbaum bis zum Erreichen des Zeitlimits nicht komplett durchsucht sein, wird nach Ablauf der Zeit der Zug mit dem bis dahin höchsten Wert verwendet. Für das Offloading ist diese Methode nicht geeignet. Ein relativ schneller Server ist in der Lage, deutlich mehr Spielzüge zu analysieren als ein langsameres Mobilgerät in derselben Zeit. Wenn nun in einem Spiel ein Zug mit Hilfe von Offloading berechnet wird und für einen anderen nicht, dann verfälscht das die Stärke der KI. Wenn jeder Zug immer dieselbe Zeit benötigt, egal ob der Suchalgorithmus auf dem Server oder auf dem Mobilgerät ausgeführt wird, lässt sich kein Zeitvorteil durch Offloading erreichen. Abhilfe kann durch die Verwendung einer maximalen Suchtiefe anstatt der maximalen Suchzeit erreicht werden.
- **Identifikation der Eingabeparameter**
  Die Offloading-Engine verwendet für die Einträge in der Datenbank des *Automated estimation system of task execution times* eine eindeutige Identifikationsnummer, die aus den jeweiligen Eingabeparametern des auszuführenden Tasks berechnet werden muss. Diese ID wird als Java-Datentyp *long* gespeichert und darf deshalb nicht länger als 64 Bit sein.
- **Grafische Benutzeroberfläche anpassen**
  Das Schachspiel bietet die Möglichkeit, sich den „Denkprozess“ der KI anzeigen zu lassen. Diese Funktion soll zu Gunsten der Anzeige eines aktuell laufenden Offloading-Prozesses nicht mehr unterstützt werden. Auch die Einstellungen werden um einige Punkte, die das Offloading betreffen, erweitert.

3.2 Entwurf

Dieser Abschnitt beschreibt die Entwurfsentscheidungen, die für die Umsetzung relevant sind.

3.2.1 Identifikation der Offloading-Tasks

Um während der Laufzeit zu entscheiden, welche Teile der Anwendung überhaupt die Möglichkeit bekommen auf einem Server ausgeführt zu werden, müssen innerhalb der Anwendung die entsprechenden Offloading-Taks identifiziert werden. In früheren Arbeiten wurde dabei mit Tasks unterschiedlicher Granularität gearbeitet. Als Grundlage können dafür Klassen, Objekte, Methoden, Komponenten, und Threads dienen [WSS+15]. Eine erste Analyse des Schachspiels mit dem Tool *FlowDroid*[3] für statische Programmanalyse von Android-

[3] *FlowDroid*: `http://sseblog.ec-spride.de/tools/flowdroid/` (Abruf: 20.01.2016)

Anwendungen ergibt eine sehr hohe Anzahl an Methoden und gegenseitigen Aufrufen. Für jede Methode und für jede Schnittstelle zwischen den Methoden die Kosten zu bestimmen, ist auf manuellem Weg ein enormer Aufwand. Außerdem lohnt es sich nicht, jeder noch so kleinen Berechnung die Möglichkeit zu geben, auf dem Offloading-Server ausgeführt zu werden. Zum einen sind viele der Berechnungen von geringer Komplexität und ein guter Entscheidungsalgorithums würde sich nicht für die entfernte Ausführung entscheiden. Zum anderen beeinflussen die ständigen Entscheidungen, ob ein Task entfernt oder lokal ausgeführt wird, die Performanz des Schachspiels negativ. Eine Granularität auf Methodenebene ist demnach weder empfehlenswert noch umsetzbar. Stattdessen sollen die Offloading-Tasks anhand ihrer logischen Funktionalität ausgewählt werden.

Sind die Offloading-Tasks identifiziert, ergeben sich Schnittstellen, über die mit den Tasks kommuniziert wird. Bei einem Offloading-Vorgang findet die Kommunikation mit diesen Schnittstellen über ein Netzwerk statt. Damit es zur Laufzeit nicht zu inkonsistenten Daten kommt und das Programm fehlerhaft arbeitet oder sogar abstürzt, müssen diese Schnittstellen sehr klar definiert werden. Bei einer von Grund auf entwickelten Anwendung mit Offloading-Funktionalität können diese Schnittstellen schon im Rahmen der Systemarchitektur berücksichtigt werden. Bei der nachträglichen Erweiterung einer bereits existieren Java-Anwendung muss auf folgende Punkte geachtet werden:

- Welche globalen Variablen werden vom Offloading-Task verwendet?
- Welche Datenstrukturen werden auf dem Server benötigt?
- Welche weitere Funktionalität wird vom Offloading-Task verwendet und muss deshalb auf dem Server verfügbar sein?

Nicht für alle Teile des Schachspiels bietet es sich an, die Ausführung von einem leistungsstarken Server durchführen zu lassen. Wie schon im Rahmen von Kapitel 2.5 erwähnt, müssen einige Methoden zwingend auf dem Mobilgerät ausgeführt werden. Zum Beispiel die grafische Benutzeroberfläche: Sie steht in direkter Verbindung mit dem Benutzer, verarbeitet dessen Eingaben und gibt Rückmeldung. Aus diesem Grund ist es nicht möglich, Teile der Benutzeroberfläche des Schachspiels an einen Server auszulagern. Auch Tasks, die offensichtlich keine hohen Kosten verursachen, können von vornherein ausgeschlossen werden. Die Kosten für die Verbindungsverwaltung wären zu hoch, um solch triviale Tasks auf einem Server auszuführen. Da für jeden Task eine Implementierung für das Mobilgerät und eine für den Server vorhanden sein muss, steigt der Aufwand in der Entwicklung mit der Anzahl der Offloading-Tasks. Die Auswahl sollte also auch ein guter Kompromiss zwischen Nutzen und Entwicklungsaufwand sein. Es gilt herauszufinden, welches die komplexesten und somit kostenintensivsten Teile einer Anwendung sind und diese in einer zweiten, auf dem Server lauffähigen Implementierung bereitzustellen.

Laut Entwickler von *CuckooChess* sind folgende Standardmethoden für Schachcomputer implementiert [Ös]:

- *Iterative Deepening*
  Schrittweise Erhöhung der Tiefe des Suchbaumes in der Alpha-Beta-Suche. [Kop06]
- *NegaScout*-Suche
  Variante der Alpha-Beta-Suche, bei der Duplikate in der Implementierung vermieden werden. [Kop06]
- *Aspiration Windows*
  wird im Zusammenhang mit der iterativen Tiefensuche verwendet. Dabei wird das Suchfenster initial auf einen kleinen Bereich um den Ergebniswert der vorherigen Berechnung gesetzt. [Kop06]
- *Quiescence Search* mit *SEE Pruning* und *MVV/LVA Move Ordering*
  Nach der Alpha-Beta-Suche werden nur noch die wichtigsten der möglichen Züge betrachtet. [Kop06]
- *Hash Table*
  In einer Hashtabelle werden bereits untersuchte Stellungen und ihre Ergebnisse abgespeichert, damit sie, falls sie bei einer anderen Zugfolge auftaucht, wiederverwendet werden können. [Kop06]
- *History Heuristic*
  Sortiert den Suchbaum anhand vorangegangener Beobachtungen, so dass die voraussichtlich besten Zugfolgen als erstes durchsucht werden. [WWHU06]
- *Recursive Null Moves*
  Reduziert die Größe des Suchbaumes durch das Ausführen zweier Züge eines Spielers. Es wird davon ausgegangen, dass das Zugrecht einen Vorteil darstellt. Ist der Vorteil durch einen zweiten Zug nicht groß genug, war vermutlich der erste Zug minderwertig und der entsprechende Ast des Suchbaumes wird abgeschnitten. [Kop06]
- *Futility Pruning*
  Verwirft Züge ab einer Suchtiefe von eins, die kein Potential besitzen, den Alpha-Wert der Alpha-Beta-Suche noch zu erhöhen. [Hei98]
- *Late Move Reductions*
  Die Zugfolgen in einem Suchbaum werden nach der *History Heurisitc* so sortiert, dass die Zugfolgen mit den besten Aussichten als erstes und zur vollen Tiefe durchsucht werden. Sollte die Wertung dieser zuerst durchsuchten Züge gut sein, können viele der weiteren Zugfolgen in der Suchtiefe begrenzt weden. [Eir11]
- *Opening Book*
  Eröffnungszüge werden nicht durch einen Algorithmus gesucht und bewertet, sondern aus einem Datensatz mit Eröffnungszügen ausgewählt.
- *Magic Bitboards*
  Interne Repräsentation des Spielfeldes mit seinen Figuren durch die Benutzung von 64-bit Integers mit einer *Magic Hash Function*. [Reu09]

Viele dieser Methoden sind Erweiterungen der eigentlichen Suche im Spielbaum und eng miteinander verzahnt. Jede Methode als eigenständigen Task zu sehen und eine eigene, separate

Offloading-Entscheidung zu treffen, ist deshalb nicht sinnvoll. Auf Grundlage der verwendeten Schachcomputertechniken und bei näherer Betrachtung der Implementierung lassen sich die verschiedenen Technologien in zwei Hauptbereiche einteilen: Suche und Datenstrukturen und -verwaltung.

- Suche
  - *Iterative Deepening*
  - *NegaScout*-Suche
  - *Aspiration Windows*
  - *Quiescence*
  - *History Heuristic*
  - *Recursive Null Moves*
  - *Futility Pruning*
  - *Late Move Reductions*
- Datenstrukturen und Datenverwaltung
  - *Hash Table*
  - *Opening Book*
  - *Magic Bitboards*

Da der Zugriff und das Verwalten der Daten, wie die *Magic Bitboards*, *Opening Books* und die *Hash Tables*, auf vielen einfachen Operationen beruht, ist eine Ausführung auf einem Server nicht sinnvoll.

Bei einer durchschnittlichen Anzahl von 30 legalen Zügen [Sha88] entsteht sehr schnell ein großer Suchbaum. Trotz diverser Verbesserungen und Bemühungen, wie der Alpha-Beta-Suche, den Suchbaum durch Vorhersagen zu verkleinern, ist das Durchsuchen des Baumes und das Ermitteln des besten Zuges die eigentliche Herausforderung für den Prozessor. Um Zeit und Energie zu sparen, bietet es sich an, den Prozessor des Mobilgerätes zu entlasten indem der Suchvorgang von einem stationären Server durchgeführt wird.

Unter Berücksichtigung der oben beschriebenen Erkenntnisse und eingehender Code-Analyse lässt sich der folgende Offloading-Task identifizieren: *iterativeDeepening*. Dieser Task beinhaltet die iterative Tiefensuche nach dem nächsten Spielzug mit ihren Optimierungen. Aufgerufen wird er mit der Methode *iterativeDeepening()* in der Klasse *ComputerPlayer*. Damit der Server die Suche durchführen kann, benötigt er alle legalen Züge des aktuellen Spielfeldes, die maximale Suchtiefe und Anzahl von zu durchsuchenden Knoten, den Zustand des aktuellen Spielfeldes und die vergangenen Zustände des Spiels. Diese Daten müssen serialisiert und an den Server übertragen werden.

3.2.2 Bestimmung der Kosten

Um eine Offloading-Entscheidung zu treffen ist es erforderlich, die Kosten der auszulagernden Aufgaben und die im Falle eines Offloading-Vorgangs anfallenden Kosten für den Datentransfer zu bestimmen.

Lokale und entfernte Kosten

Um die Kosten für die lokale und entfernte Ausführung zu bestimmen, soll zunächst beschrieben werden, welche Kosten bei der Ausführung von Programmcode entstehen:

- Energieverbrauch
- Bytecode-Instruktionen
- CPU-Zeit

Geht es um das Reduzieren des Energieverbrauchs, sind die am geeignetesten Kosten offensichtlich der Energieverbrauch selbst. Da jedes Mobilgerät sich im Energieverbrauch unterscheidet, müssten die Daten für jedes Modell neu ermittelt werden. Das würde die Kombatibilität der Anwendung zunächst stark einschränken und ist deshalb nicht empfehlenswert. Für sehr einfache Offloading-Tasks, die nur aus wenigen simplen Anweisungen bestehen, bietet sich eine Kostenfunktion basierend auf den Bytecodeinstruktionen an [BH06]. Im Falle von Android-Anwendungen wäre das eine Analyse des Dalvik-Bytecodes. Wenn bekannt ist, wie viele Bytecodeinstruktionen von einer Maschine pro Sekunde ausgeführt werden und aus wievielen Bytecodeinstruktionen ein Offloading-Task besteht, erhält man eine sehr genaue Kostenabschätzung auf Basis der Laufzeit. Für komplexere Offloading-Tasks ist dieses Vorgehen leider nur schwer bis gar nicht umsetzbar. Die Analyse von Tasks mit viel Code, vielen Klassen und Datenstrukturen ist nur äußerst mühselig per Hand durchzuführen. Da der in Kapitel 3.2.1 identifizierte Task diese Komplexität bei weitem erreicht, ist eine Kostenabschätzung anhand manuell analysierter Instruktionen nicht empfehlenswert. Eine Möglichkeit wäre die automatische Analyse mit Hilfe eines Tools, welches Dalvik-Bytecode instrumentieren kann. Leider finden sich keine geeigneten freien Tools, die solch eine Instrumentierung beherrschen. Auch eine Untersuchung mit Hilfe von Tools, die den Java-Bytecode analysieren, bietet sich nicht an. Bei der Entwicklung von Android Anwendungen wird Java-Bytecode zwar mit Hilfe eines Cross-Compilers in Dalvik-Bytecode übersetz. Jedoch verwendet Java-Bytecode mit einer Stack-orientierten Arbeitsweise eine höhere Befehlszahl als der Register-orientierte Dalvik-Bytecode [RMS15]. So lässt sich aus der Anzahl an Java-Bytecodeinstruktionen nicht auf die Anzahl an Dalvik-Bytecodeinstruktionen schließen. Um Kompatibilität zu gewährleisten und den Entwicklungsaufwand gering zu halten, soll für das Schachspiel die CPU-Zeit als Kostenmaß genutzt werden. Die Kosten werden hierbei anhand der benötigten CPU-Zeit des Tasks bestimmt. Da die CPU eine Aufgabe nicht linear abarbeitet, sondern durch Prozess-Scheduling immer wieder Zeit für andere Aufgaben benötigt, ist diese Art der Kostenabschätzung ungenauer. Um diese Ungenauigkeit auszugleichen, verwendet das AMDCOP aus der Offloading-Engine das in Kapitel 2.5.1 erläuterte KNN smoothing.

Kommunikationskosten

Sollte ein Task nicht auf dem mobilem Endgerät ausgeführt werden sondern auf einem Server, entstehen durch den Transfer der Daten extra Kosten, die bei einer lokalen Ausführung entfallen. Auch diese Kosten müssen für eine korrekte Offloading-Entscheidung berücksichtigt werden. Diese Kosten setzen sich aus den folgenden drei Größen zusammen:

1. Die Größe der über das Netzwerk zu verschickenden Daten.
2. Die aktuelle Bandbreite.
3. CPU-Zeit für die Verwaltung der Verbindung.

Um die Anfrage eines Mobilgerätes zu bearbeiten, benötigt der Server Daten. Beispielsweise werden für die Berechnung des nächsten Zuges mindestens die Positionen der aktuellen Spielfiguren benötigt. Wenn der Server diese Aufgabe erledigt hat, muss er das Ergebnis wieder zurück an das Mobilgerät schicken. Beides verursacht im Vergleich zu der lokalen Ausführung zusätzliche Kosten. Diese Kosten müssen bei der Offloading-Entscheidung mit einbezogen werden. Die CPU-Zeit für die Verwaltung der Verbindung ist im Vergleich zu der Größe der Daten und der aktuellen Bandbreite weniger ausschlaggebend und kann somit vernachlässigt werden. Die aktuelle Bandbreite zu bestimmen ist ohne weiteres möglich. In der verwendeten Offloading-Engine wird diese zur Bestimmung der Übertragungszeit ermittelt. Eine Bestimmung der Größe der Daten, die an den Server geschickt werden sollen, ist möglich und wird von der Offloading-Engine unterstützt. Das vorherbestimmen der Größe der Daten, die als Antwort vom Server zurück kommen, ist jedoch nicht möglich. Wie schon in Kapitel 2.5.1 beschrieben, wird diese Größe unter der Annahme, dass sie vernachlässigbar ist, deswegen nicht berücksichtigt.

3.2.3 Offloading-Entscheidung

Für eine Anwendung mit vielen Offloading-Tasks bietet sich einer der in 2.4 vorgestellten Partitionierungs-Algorithmen an. Da im Schachspiel nur ein einziger Offloading-Task existiert, macht eine Partitionierung nach Server bzw. Servern und Mobilgerät mit Hilfe eines der in Kapitel 2.4 vorgestellten Algorithmen wenig Sinn.

Für die Entscheidung, ob ein Task lokal oder entfernt ausgeführt wird, stellt die Offloading-Engine einen einfachen Mechanismus bereit: Wenn der Server erreichbar ist und die Kosten für die lokale Ausführung sowie die entfernte Ausführung und die Kommunikation bei einem Offloading-Vorgang bekannt sind, wird anhand eines logischen Ausdrucks ermittelt, ob die lokale Ausführung oder die Summe aus Kommunikation und entfernter Ausführung günstiger ist. Die Engine unterscheidet dabei nach Zeitersparnis, Energieersparnis oder Zeit- und Energieersparnis. Leider sind für eine Entscheidung anhand des Kriteriums Energieersparnis Daten über den Energieverbrauch des Mobilgerätes nötig. Um diese zu ermitteln ist zusätzliche Hardware nötig, die im Rahmen dieser Arbeit nicht zur Verfügung steht. Außerdem würde das die Kompatibilität der Anwendung auf Mobilgeräte beschränken, dessen Werte zum Energieverbrauch bekannt sind. Deshalb liegt der Fokus in dieser Arbeit auf Energieersparnis durch das Einsparen von CPU-Zeit. Solange das Mobilgerät während der entfernten Ausführung eines Tasks nicht in einen Ruhemodus wechselt, verhält sich der Energieverbrauch proportional zu der Ausführungszeit [CIM+11]. Die Entscheidung anhand

des Kriteriums Energieersparnis soll aber trotzdem innerhalb der Einstellungen des Schachspiels auswählbar bleiben. Falls die benötigten Werte in der Zukunft bekannt werden, lassen sie sich leicht nachträglich verwenden.

3.2.4 Eingabeidentifikation der Offloading-Tasks

Für die Datenbank des AESTET wird für jede Eingabe eines Tasks eine ID benötigt. Wie in Kapitel 2.5.1 beschrieben, geht das System davon aus, dass ein Task bei gleichen Eingabeparametern auch dieselbe Zeit zur Bearbeitung und bei einer ähnlichen Eingabe eine ähnliche Bearbeitungszeit benötigt. Auf dieser Grundlage muss jede Eingabe eines Tasks in eine ID überführt werden, die in der Datenbank mit der benötigten Bearbeitungszeit abgespeichert wird.

Bei einem Schachspiel bietet es sich an, den numerischen Hashwert, der das Spielfeld mit den Positionen der Figuren repräsentiert, zu verwenden. Dieser Wert wird in *CuckooChess* bereits für die Verwendung der Hash-Tabellen berechnet. Es wäre also leicht, diesen als ID zu verwenden. Jedoch ist dadurch nicht garantiert, dass zwei IDs, die einen geringen Abstand zu einander haben, auch eine ähnliche Berechnungszeit bei der Durchsuchung des Spielbaumes haben. Die Zeit, die zum Durchsuchen benötigt wird, hängt von der Anzahl an Knoten ab, die in dem Baum durchsucht werden muss. Die genaue Anzahl der Knoten lässt sich allerdings erst bestimmen, wenn die Suche bereits abgeschlossen ist, also nachdem der Tasks ausgeführt wurde. Ein Indikator dafür, wie groß der Suchbaum wird, ergibt sich aus der Anzahl der legalen Züge zu Beginn der Suche im Baum. Dieser Wert lässt sich leicht auslesen und kann deshalb als ID verwendet werden. Zusätzlich bestimmt auch die Tiefe des Suchbaumes, wie viel Zeit für die Suche benötigt wird. Da die Tiefe durch den eingestellten Schwierigkeitsgrad beschränkt ist, ist sie schon vor der Ausführung des Suchalgorithmus bekannt und wird zusätzlich zur Bildung der ID hinzugezogen. Eine Beispielhafte ID sieht wie folgt aus:

$$\text{ID: } \underbrace{14}_{\text{Suchtiefe}} \quad \underbrace{32}_{\text{Figuren}} \quad \underbrace{20}_{\text{legale Züge}} \tag{3.1}$$

3.3 Implementierung

Dieses Kapitel beschreibt die Implementierung der Anwendung auf Grundlage der in Kapitel 3.1.3 aufgestellten Anforderungen und der in Kapitel 3.2 getroffenen Entscheidungen inklusive einer Übersicht über das komplette System. Dabei werden nur die für das Offloading benötigten Mechanismen im Detail erläutert.

3.3.1 Übersicht des Systems

Das System ist in vier Schichten aufgeteilt. Jede Schicht hat nur Zugriff auf die ihr unter- und übergeordnete Schicht. Die in *CuckooChess* vorhandenen Schichten *Android UI*, *Controller* und *Chess Engine* wurden um die Schicht *Offloading Engine* erweitert.

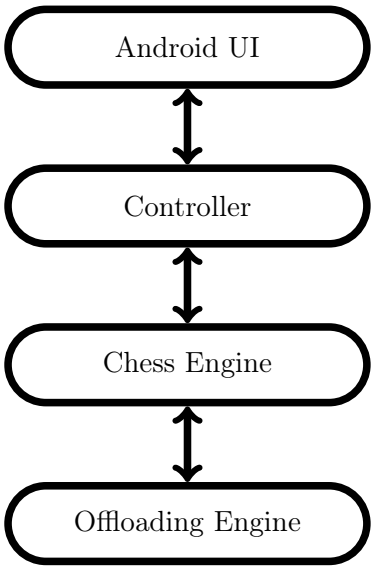


Abbildung 3.3: Schichtenmodell von *OffloadingChess*

3.3.2 Serialisierung

Um die in derAbschnitt 3.2.1 beschriebenen Daten zwischen Server und Mobilgerät verschicken zu können, wird das Offloading-Engine um die Klasse *Serializer* erweitert. Diese Klasse stellt die zwei statischen Methoden *serialize()* und *deserialize()* zum Serialisieren und Deserialisieren von Objekten, die das Java Interface *Serializable* implementieren, bereit. Um unterschiedlichen Kodierungen zwischen Client und Server vorzubeugen, wird server- und clientseitig das *base64*-Verfahren angewandt. Dafür kann auf Seiten von Android auf die Systembibliothek *android.util.Base64* zurückgegriffen werden, während serverseitig die von Java bereitgestellte *javax.xml.bind.DatatypeConverter* Bibliothek verwendet wird.

3.3.3 Schwierigkeitsgrad

Zusätzlich zu den Schwierigkeitseinstellungen durch eine maximale Suchzeit wird im leichtesten Schwierigkeitsgrad des ursprünglichen *CukooChess* der Suchbaum nur bis zum ersten Halbzug, also bis zu einer Suchtiefe von eins, durchsucht. Dieser Mechanismus kann in *OffloadingChess* genutzt werden, um die Suche nach einer bestimmten Suchtiefe abzubrechen. Die Werte in Tabelle 3.1 haben sich dabei unter Berücksichtigung von [Fer13] als geeignet herausgestellt, um verschiedene Schwierigkeitsgrade im Spiel einzustellen.

Schwierigkeitsgrad	**Suchtiefe**
Very Easy	1
Easy	4
Moderate	6
Hard	14

Tabelle 3.1: Schwierigkeitsgrad und ihre jeweilige Suchtiefe im Suchbaum

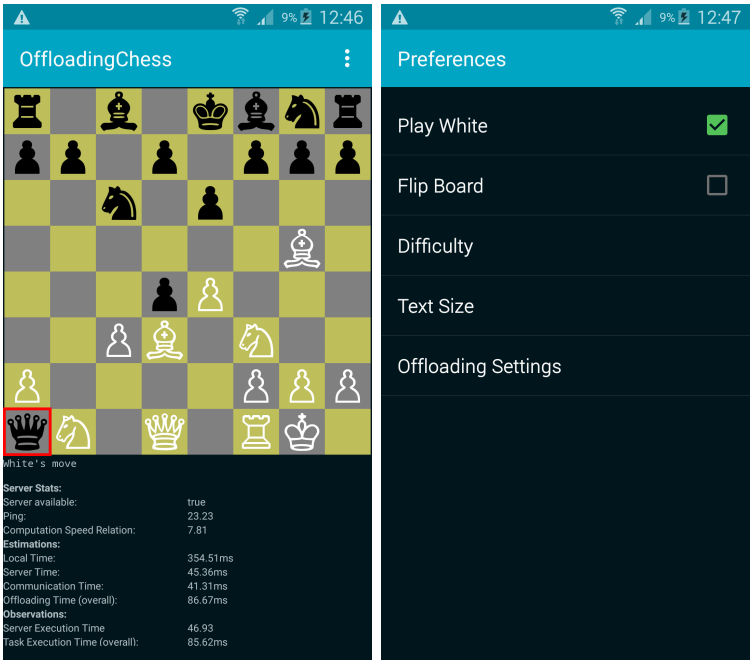
3.3.4 Eingabeidentifikation der Offloading-Tasks

Die in der Datenbank gespeicherten Kosten der einzelnen Tasks werden mit einer ID gespeichert. Diese setzt sich aus drei Teilen zusammen: Der aktuelle Schwierigkeitsgrad, die Anzahl der Figuren auf dem Schachbrett und die Anzahl der legalen Züge zu Beginn der Suche. Der Schwierigkeitsgrad wird vom Spieler im Optionsmenü des Schachspiels angegeben und kann somit leicht ausgelesen werden. Die Anzahl der Spielfiguren wird durch Iteration über das Spielfeld ermittelt und die Anzahl der legalen Züge wird zur Ausführung des Tasks *iterativeDeepening* benötigt und kann deshalb ausgelesen werden.

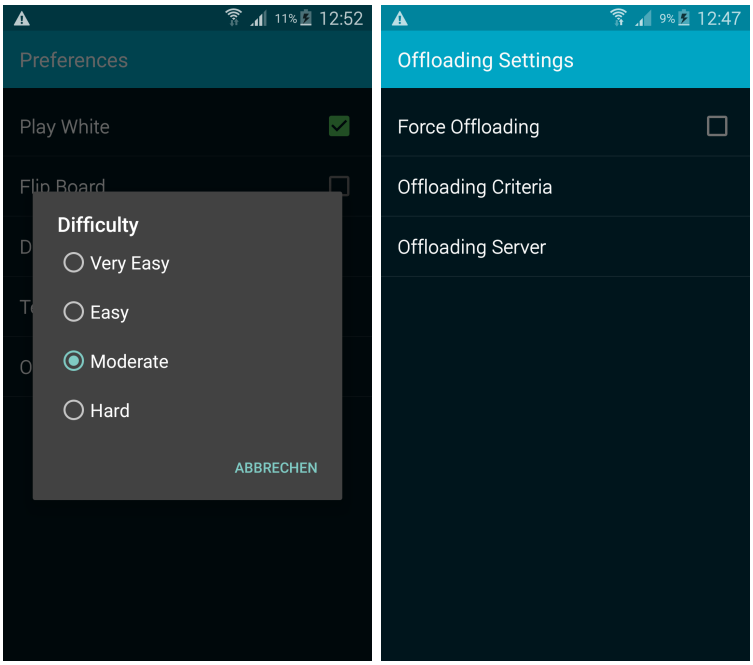
3.3.5 Grafische Oberfläche

Durch die Verwendung von Offloading wurde die Oberfläche des Schachspiel der neuen Funktionalität angepasst.

Abbildung 3.4(a) zeigt das Hauptfenster samt Spielfeld. Nach jedem Offloading-Vorgang werden detaillierte Daten, wie die Schätzungen und tatsächliche Ausführungszeit und Daten zu der Netzqualität, wie Ping und Bandbreite, angezeigt. Abbildung 3.4(b) zeigt das Einstellungsmenü. Aus Gründen, die in 3.1.3 erläutert wurden, wird der Schwierigkeitsgrad nicht mehr mit einer vorgegebenen maximalen Suchzeit, sondern mit einer maximalen Suchtiefe eingestellt. Abbildung 3.4(c) zeigt das dazugehörige Auswahlmenü. Abbildung 3.4(d) zeigt das Menü, in dem Einstellungen für das Offloading getätigt werden.



(a) Hauptfenster (b) Einstellungen



(c) Einstellungen: Schwierigkeitsgrad (d) Einstellungen: Offloading

Abbildung 3.4: Grafische Oberfläche von *OffloadingChess*

3.3.6 Änderungen an der Offloading-Engine

Während der Implementierungsarbeiten wurde festgestellt, dass noch folgende, außerplanmäßige Änderung an der Offloading-Engine vorgenommen werden muss.

Das Offloading-Framework kann Kosten, wie in Kapitel 3.2.2 beschrieben, auf der Grundlage von Bytecode Instruktionen oder mit Hilfe der durchschnittlichen CPU-Zeit berechnen. Da im Falle des hier entwickelten Offloading-Schachs die Kosten durch die CPU-Zeit gegeben sind, führt folgender Code in der Klasse *Engine* in der Methode *decide()* zu einer verfälschten Kostenabschätzung:

```
[...]
estServerRuntime = Algorithms.getCost(algName, parameters) /
    SERVER_INST_MS; //Estimated server execution time
[...]
```

Quellcode 3.1: Kostenberechnung mit Hilfe von Instruktionen pro Millisekunde

Durch das Weglassen des konstanten Divisors $SERVER\_INST\_MS$, der für Instruktionen pro Millisekunde steht, wird das Problem behoben.

3.3.7 Implementierung für den Server

Die Implementierung für den Server kann eins zu eins von der Android Implementierung übernommen werden. Zusätzlich zu den für den Task *iterativeDeepening* benötigten Java-Klassen beinhaltet die Implementierung noch die Klassen *ChessController* und *Serializer*. *ChessController* stellt eine Methode als Einstiegspunkt für das Offloading bereit, deserialisiert die empfangenen Parameter, um sie dem Suchalgorithmus zu übergeben und schickt das serialisierte Ergebnis des Algorithmus zurück an das Mobilgerät. Die Klasse *Serializer* ist analog zu der Implementierung für das Mobilgerät und stellt Methoden zum Serialisieren und Deserialisieren bereit.

3.4 Experimente

Dieses Kapitel beschreibt den Aufbau und die Durchführung von Experimenten, die mit dem implementierten Schachspiel durchgeführt wurden.

3.4.1 Forschungsfragen

Mit den Experimenten sollen die folgenden Forschungsfragen beantwortet werden:

- **Wird durch mobiles Offloading der Energieverbrauch des Mobilgerätes beim Spielen eines Schachspiels reduziert?**
  Auf Grundlage der vorhandenen Daten wird untersucht, ob Offloading bei einem Schachspiel die CPU entlastet und somit Energie eingespart wird.
- **Wurden fehlerhafte Offloading-Entscheidungen getroffen?**
  Bildet man aus Laufzeit des Servers und Kommunikationszeit die Summe und vergleicht diese mit der lokalen Laufzeit, lässt sich hieraus schließen, ob Offloading durchgeführt wurde obwohl eine lokale Ausführung effizienter gewesen wäre.
- **Ist mit Hilfe des AESTET und der verwendeten Eingabe-ID eine genaue Vorhersage der Laufzeit möglich?**
  Durch den Vergleich von realer Laufzeit und geschätzter Laufzeit lässt sich herausfinden, wie genau die Vorhersagen sind.
- **Beeinflusst der im Spiel eingestellte Schwierigkeitsgrad die Offloading-Entscheidung und die Laufzeitschätzung?**
  Anhand der gesammelten Daten wird ersichtlich, ob die Laufzeitschätzung in den unterschiedlichen Schwierigkeitseinstellungen besser oder schlechter wird.
- **Werden bei einer Verbindung zum WLAN mehr Offloading-Vorgänge ausgeführt als bei einer Verbindung zum mobilen Netz?**
  Vermutlich werden auf Grund der geringeren Bandbreite und des höheren Pings im mobilen Netz deutlich weniger Offloading-Vorgänge durchgeführt als im WLAN.
- **Beeinflusst die Leistung des Mobilgerätes die Offloading-Entscheidung?** Ein leistungsstarkes Mobilgerät sollte für eine Berechnung weniger Zeit benötigen als ein leistungsschwaches. Dadurch verringert sich der Gewinn an Zeit, die für die Berechnung benötigt wird. Das leistungsschwächere Mobilgerät müsste daher mehr Offloading-Vorgänge durchführen.
- **Kann das Schachspiel und auch die Offloading-Engine im Allgemeinen zukünftig noch verbessert werden?**
  Auf Grundlage der Ergebnisse im Bezug auf die Forschungsfragen lassen sich möglicherweise Schlüsse ziehen, wie in der Zukunft das Offloading noch weiter verbessern werden könnte.

3.4.2 Versuchsaufbau

Als Server steht ein Rechner mit einem Intel Core 2 Duo mit zwei Kernen á 3 GHz und 8 GB Arbeitsspeicher zur Verfügung. Dieser ist mit bis zu 32000 kbit/s Downlink und 2000

kbit/s Uplink an das Internet angebunden. Als Betriebssystem kommt auf dem Server openSUSE 42.1 zum Einsatz. Als Mobilgerät wird zum einen ein Samsung Galaxy S5 mit einem Qualcomm Snapdragon 801 Prozessor mit vier Kernen á 2,5 GHz und 2 GB Arbeitsspeicher verwendet. Zum anderen ein Samsung Galaxy S6 mit Samsung Exynos 7420 Prozessor mit acht Kernen, von denen vier mit 2,1 GHz und vier mit 1,5 GHz laufen, und 3 GB Arbeitsspeicher.

Die initiale, mit dem Spiel ausgelieferte Datenbank des AESTET beinhaltet 386 Datensätze. Das entspricht zwei Schachpartien pro Schwierigkeitsgrad. Damit das AESTET möglichst genaue Vorhersagen treffen kann, wurden auf beiden Smartphones weitere Partien Schach gespielt, bis die Datenbank eine Größe von insgesamt 2400 Einträgen erreichte. Pro Schwierigkeitsgrad befinden sich 600 Einträge in der Datenbank des jeweiligen Smartphones.

Die Experimente wurden zum einen mit vorhandener WLAN Verbindung (bis zu 145000 kbit/s) und Server im selben Netzwerk und zum anderen mit Verbindung zum mobilen Datennetz durchgeführt. Um möglichst realistische Bedingungen und sich kontinuierlich ändernde Qualität der Anbindung ans Netz zu schaffen, wird die ausführende Person mit dem Mobilgeräten während der Durchführung in Bewegung sein.

Des weiteren werden folgende Datensätze gesondert betrachtet:

- Samsung Galaxy S5 im mobilen Netz
- Samsung Galaxy S5 im WLAN
- Samsung Galaxy S6 im mobilen Netz
- Samsung Galaxy S6 im WLAN

Um die Daten besser vergleichen zu können, wird die Aktualisierung der Datenbank des jeweiligen Smartphones für die Dauer der Experimente deaktiviert. Andernfalls würde sich die Datenbank kontinuierlich mit Daten füllen und somit die Schätzungen der Laufzeiten immer genauer werden.

Nach jedem Schachzug werden die benötigten Daten in einer csv-Datei auf der SD-Karte des Smartphones gespeichert und können so leicht mit einer Statistik-Software ausgelesen werden. Folgende Variablen werden nach jedem Zug gespeichert:

- *GameID* (Datum): Das aktuelle Datum und die aktuelle Uhrzeit zu Beginn eines Spiels werden als ID gespeichert. So kann nachvollzogen werden welche Datensätze zu welchem Spiel gehören.
- *InputID* (numerisch): Die in 3.2.4 beschriebene Eingabe ID.
- *NetworkType* (Zeichenkette): Die Art der Netzanbindung des Smartphones (zum Beispiel WLAN, *Long Term Evolution* (LTE), ...).
- *ServerAvailable* (Wahrheitswert): Erreichbarkeit des Servers.
- *Ping* (numerisch): Ping zum Server.
- *CSR* (numerisch): Faktor, um den der Server schneller ist als das Mobilgerät.
- *DecisionIndication* (Aufzählung): Offloading Entscheidung auf Grund Zeitersparnis und / oder Energieersparnis.
- *LocalTimeEstimation* (numerisch): Geschätzte Laufzeit auf dem Mobilgerät.

- *ServerTimeEstimation* (numerisch): Geschätzte Laufzeit auf dem Server.
- *CommunicationTimeEstimation* (numerisch): Geschätze Zeit, die für die Kommunikation benötigt wird.
- *OverallOffloadingTimeEstimation* (numerisch): *ServerTimeEstimation* + *CommunicationTimeEstimation*.
- *CommunicationEnergyEstimation* (numerisch): Geschätzter Energieverbrauch für die Kommunikation mit dem Server.
- *LocalEnergyEstimation* (numerisch): Geschätzter Energieverbrauch für die lokale Ausführung.
- *ServerEnergyCommunication* (numerisch): Geschätzter Energieverbrauch für die Ausführung auf dem Server.
- *RealServerTime* (numerisch): Tatsächliche Ausführung auf dem Server.
- *RealExcutionTime* (numerisch): *RealServerTime* + Zeit für Kommunikation.
- *OffloadingDone* (Wahrheitswert): *True* wenn Offloading durchgeführt wurde, sonst *False*.
- *ExecutedTask* (Zeichenkette): Name des Offloading-Tasks.
- *ParameterSizeKB* (numerisch): Größe, der zum Offloading-Server verschickten Daten in KB.
- *UplinkKBs* (numerisch): Bandbreite für den Upload in KB/s.
- *Result* (Zeichenkette): Das Ergebnis des Offloading-Tasks.

Nicht alle dieser Variablen werden für die Beantwortung der oben aufgestellten Forschungsfragen benötigt. Jedoch könnte diese, falls nötig,für zukünftige Datenanalyse von Relevanz sein.

Die folgenden zwei Experimente werden durchgeführt:

1. Je Schwierigkeitsgrad werden mit dem Samsung Galaxy S6 160 mal ein und derselbe Zug wiederholt. Jeweils 80 mal wird die entfernte Ausführung auf dem Server erzwungen und 80 mal die lokale Ausführung auf dem Mobilgerät. Da der Offloading-Task *iterativeDeepening* pro Spielzug genau einmal ausgeführt wird, ergibt das 640 Datensätze, die im folgenden *Dataframe A* genannt werden. Anhand dieser Daten soll untersucht werden, wie weit sich die Laufzeiten eines Tasks bei selber Eingabe unterscheiden.
2. Es werden 8000 Datensätze während des normalen Spielverlaufs gesammelt. Davon 2000 mit dem Samsung Galaxy S5 im mobilen Netz, 2000 im WLAN, 2000 mit dem Samsung Galaxy S6 im mobilen Netz und weitere 2000 im WLAN. In jedem dieser Szenarien werden pro Schwierigkeitsgrad 500 Datensätze gesammelt. Zur besseren Unterscheidung wird dieser Datensatz im Folgenden *Dataframe B* genannt. Anhand dieser Daten soll ermittelt werden, wie genau die Schätzungen der Laufzeit sind.

KAPITEL 4

Evaluation

4.1 Resultate der Experimente

In diesem Abschnitt werden die Resultate der Experimente visualisiert und interpretiert.

4.1.1 Schwankungen der Laufzeit

Die folgenden Daten stammen aus *Dataframe A*. Ziel ist es, die lokalen und entfernten Ausführungszeiten auf ihre Stabilität hin zu untersuchen. Abbildung 4.1 zeigt den Zug, der für jeden der Datensätze ausgeführt wurde.

Abbildung 4.1: Spielzug, der für das Sammeln von *Dataframe A* ausgeführt wurde.

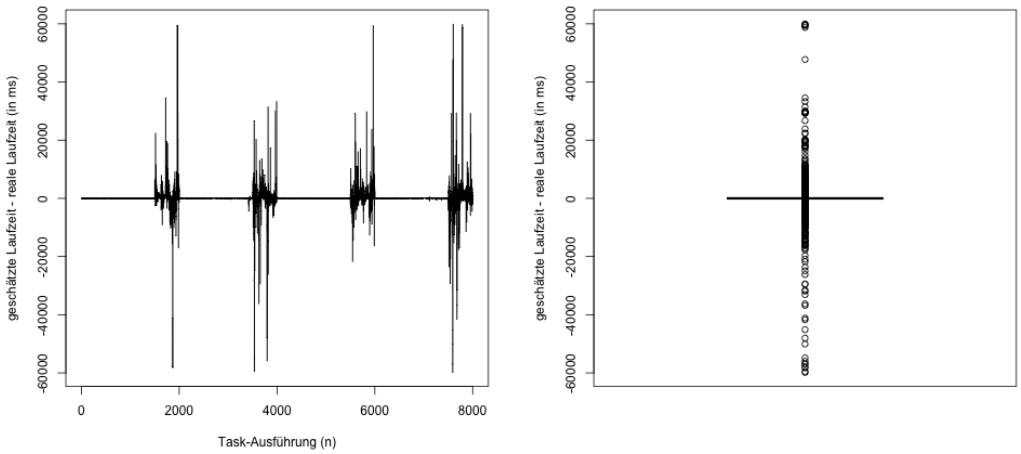
Schwierigkeitsgrad	sd (lokale Laufzeit)	sd (entfernte Laufzeit)
Very Easy	3,651ms	36,80ms
Easy	4,81ms	9,11ms
Moderate	9,66ms	17,07ms
Hard	7810,07ms	412,75ms

Tabelle 4.1: Standardabweichung (sd) bei der mehrfachen Ausführung eines Spielzuges.

Tabelle 4.1 zeigt die Standardabweichung der Laufzeit bei der Ausführung desselben Spielzuges. Mit steigendem Schwierigkeitsgrad steigt auch die Standardabweichung. Das heißt, die Laufzeit schwankt trotz Ausführung desselben Zuges immer stärker. Auffallend ist auch, dass bei den Schwierigkeitsgraden *Very Easy*, *Easy* und *Moderate* die Standardabweichung bei der entfernten Ausführung größer ist als bei der lokalen Ausführung, während bei *Hard* die Laufzeit auf dem Server deutlich stabiler als auf dem Mobilgerät ist.

4.1.2 Schätzung der Laufzeiten

Dieser Abschnitt beschäftigt sich mit der Schätzung der Laufzeiten. Es werden die Schätzungen mit den realen Werten verglichen und die Qualität der Schätzungen bewertet. Die dafür verwendeten Daten stammen aus *Dataframe B*.



(a) als Graph (b) als Boxplot

Abbildung 4.2: Differenz geschätzter und realer Ausführungszeit beider Smartphones

In Abbildung 4.2 wird die Differenz zwischen der Schätzung und der realen Ausführungszeit beider Smartphones dargestellt. Je näher die Differenz den Wert 0 erreichte, desto besser war die Schätzung der Laufzeit. Auf den ersten Blick sieht man in Abbildung 4.2(a) vier Bereiche in denen die Kurve stark nach oben und unten ausschlägt. In diesen Bereichen war die Schätzung sehr ungenau mit teilweise erheblichen Abweichungen. Im weiteren Verlauf der Analyse werden auch diese Bereiche näher untersucht. Von den vier Bereichen abgesehen,

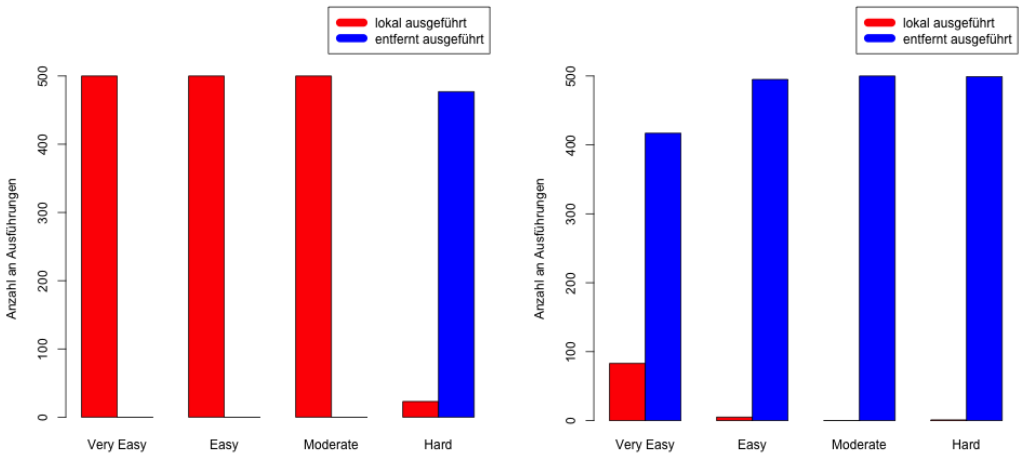
waren die Schätzungen im Allgemeinen nah an den eigentlichen Laufzeiten, wie auch die folgenden Daten belegen:

- Task-Ausführung (n): 8000
- Durchschnittliche Laufzeitabweichung (Schätzung - Laufzeit): 67,87ms
- Median der Laufzeitabweichung (Schätzung - Laufzeit): 1,85ms
- Anzahl der Schätzungen die nicht mehr als 500ms abweichen: 6484 (81,05%)
- Anzahl der Schätzungen die nicht mehr als 250ms abweichen: 6299 (78.74%)
- Anzahl der Schätzungen die nicht mehr als 10ms abweichen: 3571 (44,64%)
- Anzahl der Schätzungen die nicht mehr als 1ms abweichen: 519 (6,49%)

Anhand des Medians von 1,85ms wird deutlich, dass die Schätzungen im Ganzen keine eindeutige Tendenz zum Unter- oder Überschätzen der Laufzeit aufweist. Da der Wert relativ nah bei 0 ist, weicht die Schätzung ungefähr im gleichen Ausmaß positiv und negativ von der realen Laufzeit ab.

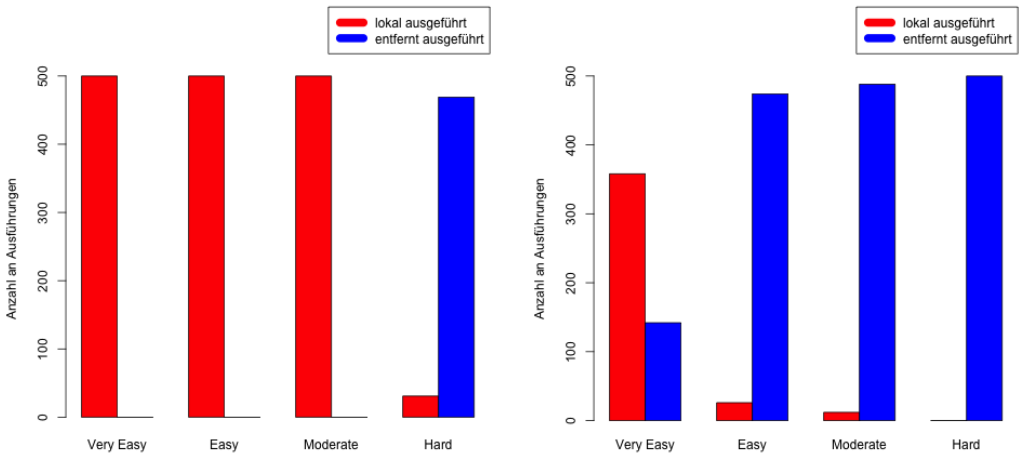
4.1.3 Einfluss des Schwierigkeitsgrades auf die Laufzeitschätzung

In diesem Abschnitt wird untersucht, wie sich der im Schachspiel eingestellte Schwierigkeitsgrad auf das Offloading auswirkt. Dazu werden die Daten der beiden Smartphones aus *Dataframe B* je nach Schwierigkeitsgrad gesondert betrachtet.



(a) Samsung Galaxy S5: mobiles Netz

(b) Samsung Galaxy S5: WLAN



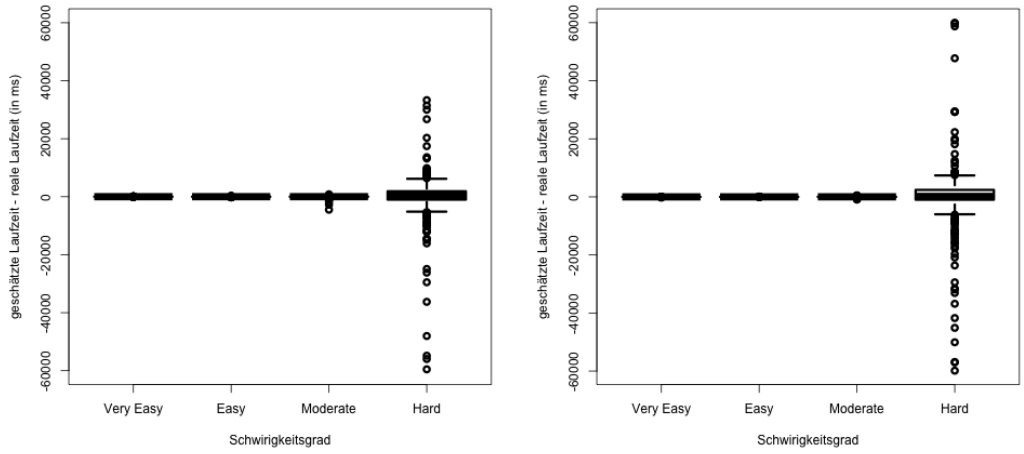
(c) Samsung Galaxy S6: mobiles Netz

(d) Samsung Galaxy S6: WLAN

Abbildung 4.3: Offloading-Vorgänge anhand des Schwierigkeitsgrads bei erreichbarem Server

In Abbildung 4.3 wird deutlich, dass nicht nur der eingestellte Schwierigkeitsgrad erheblichen Einfluss auf die Offloading-Entscheidung hat, sondern auch, dass die Leistungsfähigkeit des Mobilgerätes eine wichtige Rolle spielt. Der Server war im Schnitt 4,9 mal schneller als das Samsung Galaxy S6 und 7,8 mal schneller als das Samsung Galaxy S5. Für das leistungsfähigere Samsung Galaxy S6 lohnt sich daher ein Offloading-Vorgang erst ab deutlich rechen-

intensiveren Vorgängen als für das Samsung Galaxy S5. Auch die Art der Netzanbindung hat starken Einfluss auf die Offloading-Entscheidung. So werden auf beiden Smartphones in den Schwierigkeitsgraden *Very Easy*, *Easy*, und *Moderate* bei einer Verbindung zum mobilen Netz keine Offloading-Vorgänge durchgeführt, während bei einer WLAN-Verbindung auch in diesen Schwierigkeitsgraden Schachzüge vom Offloading-Server berechnet werden. Bei *Hard* wird dagegen auf Grund des hohen Berechnungsaufwandes auch im mobilen Netz fast ausschließlich Offloading durchgeführt.



(a) Samsung Galaxy S5 (b) Samsung Galaxy S6

Abbildung 4.4: Differenz der geschätzten und realen Ausführungszeit anhand des Schwierigkeitsgrads

Abbildung 4.4 zeigt die Differenz der geschätzten und der realen Laufzeit anhand des eingestellten Schwierigkeitsgrads. Auffallend ist, dass die Anzahl der Fehlschätzungen mit steigendem Schwierigkeitsgrad zunimmt. Bei *Hard* sind die meisten Fehlschätzungen zu finden, was auch die starken Ausbrecher in Abbildung 4.2 erklärt.

In den folgenden Abschnitten werden die Daten in Bezug auf den eingestellten Schwierigkeitsgrad noch einmal näher betrachtet. Da die Art der Netzanbindung die Schätzung der Ausführungszeit nicht beeinflusst, wird auch nicht unterschieden, ob eine Verbindung zum mobilem Netz oder zum WLAN vorliegt. Für jedes Smartphone wird nach lokaler und entfernter Ausführung unterschieden. Die Größe der jeweilige Datensätze hängt von der Offloading-Entscheidung ab und ist daher nicht gleich verteilt.

Schwierigkeitsgrad: *Very Easy*

	S5 entfernt	**S5 lokal**	**S6 entfernt**	**S6 lokal**
Task-Ausführung (n)	417	583	142	858
Durchschnittliche Laufzeitabweichung (Schätzung - Laufzeit)	-2,87ms	3,01ms	1.55ms	1,08ms
Median der Laufzeitabweichung (Schätzung - Laufzeit)	-2,05ms	2,28ms	1,63ms	1,74ms
Standardabweichung (Laufzeit)	5,37ms	19,26ms	3,15ms	20,24ms
$\leq$ 500ms Abweichung	417 (100%)	583 (100%)	142 (100%)	858 (100%)
$\leq$ 250ms Abweichung	417 (100%)	583 (100%)	142 (100%)	858 (100%)
$\leq$ 10ms Abweichung	381 (91,37%)	308 (52,83%)	136 (95,77%)	738 (86,01%)
$\leq$ 1ms Abweichung	78 (18,7%)	36 (6,17%)	26 (18,3%)	141(16,43%)
Zeitersparnis	6246,39ms		768,36ms	

Tabelle 4.2: Daten im Schwierigkeitsgrad *Very Easy*

Wie die Werte aus Tabelle 4.2 zeigen, sind die Schätzungen in diesem Schwierigkeitsgrad sehr genau. Die durchschnittliche Abweichung der Schätzung von der realen Laufzeit liegt zwischen -2,87ms und 3,01ms. Keine Schätzung lag um mehr als 250ms daneben. Auffallend ist allerdings der Unterschied in der Qualität zwischen lokaler und entfernter Laufzeit. Während nur 91,37% bzw. 95,77% der Schätzungen für den Server mehr als 10ms vom Realwert abweichen, sind es bei der Schätzung für die lokale Ausführung 52,83% bzw. 86,01%. Die Standardabweichung der Laufzeit bei der lokalen Ausführung ist größer als bei der entfernten. Das heißt, die Laufzeit des Tasks streut auf dem Mobilgerät mehr als auf dem Server. Unter der Annahme, dass die geschätzten Laufzeiten für die lokale Ausführung korrekt sind, wurde durch Offloading auf dem Samsung Galaxy S5 bei 1000 Spielzügen 6246,39ms durch 417 Offloading-Vorgänge und auf dem Samsung Galaxy S6 bei 1000 Spielzügen 768,36ms durch 142 Offloading-Vorgänge gespart.

Schwierigkeitsgrad: *Easy*

	S5 entfernt	**S5 lokal**	**S6 entfernt**	**S6 lokal**
Task-Ausführung (n)	495	505	474	526
Durchschnittliche Laufzeitabweichung (Schätzung - Laufzeit)	-1,81ms	6,83ms	0,1ms	6,49ms
Median der Laufzeitabweichung (Schätzung - Laufzeit)	-0,88ms	8,18ms	-0,05ms	5,99ms
Standardabweichung (Laufzeit)	11,11ms	44,13ms	7,89ms	21,86ms
$\leq$ 500ms Abweichung	495 (100%)	505 (100%)	474 (100%)	526 (100%)
$\leq$ 250ms Abweichung	495 (100%)	504 (99,8%)	474 (100%)	526 (100%)
$\leq$ 10ms Abweichung	374 (75,56%)	139 (27,52%)	423 (89,24%)	200 (38,02%)
$\leq$ 1ms Abweichung	49 (9,9%)	12 (2,38%)	53 (11,18%)	19 (3,61%)
Zeitersparnis	16596,32ms		22178,31ms	

Tabelle 4.3: Daten im Schwierigkeitsgrad *Easy*

Wie die Werte aus Tabelle 4.3 zeigen, sind auch diese Schätzungen noch gut getroffen. Fast 100% der Schätzungen haben keine größere Abweichung als 250ms. Auch hier unterscheidet sich die Qualität der Schätzung zwischen lokaler und entfernter Laufzeit. Auf dem Samsung Galaxy S5 wurden bei 1000 Spielzügen durch 495 Offloading-Vorgänge insgesamt 16596,32ms und auf dem Samsung Galaxy S6 bei 1000 Spielzügen durch 474 Offloading-Vorgänge 22178,31ms Zeit gespart.

Schwierigkeitsgrad: *Moderate*

	S5 entfernt	**S5 lokal**	**S6 entfernt**	**S6 lokal**
Task-Ausführung (n)	500	500	488	512
Durchschnittliche Laufzeitabweichung (Schätzung - Laufzeit)	-3ms	-11,54ms	-4,11ms	21,23ms
Median der Laufzeitabweichung (Schätzung - Laufzeit)	-1,04ms	13,37ms	-2,21ms	17,91ms
Standardabweichung (Laufzeit)	21,86ms	273,13ms	21,13ms	98,1ms
$\leq$ 500ms Abweichung	500 (100%)	491 (98,2%)	488 (100%)	504 (98,44%)
$\leq$ 250ms Abweichung	500 (100%)	482 (96,4%)	488 (100%)	492 (96,09%)
$\leq$ 10ms Abweichung	323 (64,6%)	81 (16,2%)	327 (67%)	107 (20,9%)
$\leq$ 1ms Abweichung	53 (10,6%)	6 (1,2%)	29 (5,94%)	10 (1,95%)
Zeitersparnis	74352,83ms		33526,17ms	

Tabelle 4.4: Daten im Schwierigkeitsgrad *Moderate*

Anhand der Werte aus Tabelle 4.4 lässt sich ein deutlicher Trend hin zu einer schlechteren Schätzung bei höherem Schwierigkeitsgrad ausmachen. Nur noch 64,6% bzw. 67% der Schätzungen lagen nicht mehr als um 10ms daneben und auch die durchschnittliche Laufzeitabweichung ist gestiegen. Die Schätzungen für die lokale Ausführung sind hier mit 16,2% und 20,9% noch schlechter getroffen. Trotz schlechterer Schätzungen wurde hier beim Samsung Galaxy S5 die Laufzeit bei 1000 Spielzügen um 74352,83ms durch 500 Offloading-Vorgänge und beim Samsung Galaxy S6 bei 1000 Spielzügen um 33526,17ms durch 488 Offloading-Vorgänge reduziert.

Schwierigkeitsgrad: *Hard*

	S5 entfernt	**S5 lokal**	**S6 entfernt**	**S6 lokal**
Task-Ausführung (n)	976	24	969	31
Durchschnittliche Laufzeitabweichung (Schätzung - Laufzeit)	211,44ms	-263,33ms	480,44ms	-4257,22ms
Median der Laufzeitabweichung (Schätzung - Laufzeit)	389,2ms	196,39ms	229,72ms	-132,60ms
Standardabweichung (Laufzeit)	5990,98ms	1060,84ms	7005,92ms	13709,98ms
$\leq$ 500ms Abweichung	236 (24,18%)	8 (33,34%)	238 (24,56%)	19 (61,29%)
$\leq$ 250ms Abweichung	142 (14,55%)	5 (20,83%)	175 (18,06%)	16 (51,61%)
$\leq$ 10ms Abweichung	19 (1,95%)	1 (4,17%)	11 (1,14%)	3 (6,68%)
$\leq$ 1ms Abweichung	5 (0,51%)	1 (4,17%)	1 (0,1%)	0 (0%)
Zeitersparnis	26054302ms		32181994ms	

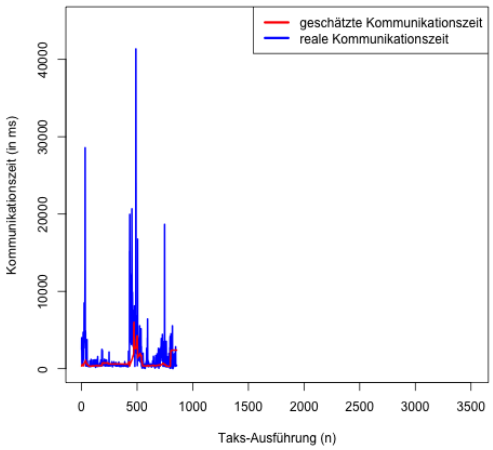
Tabelle 4.5: Daten im Schwierigkeitsgrad *Hard*

Auch hier setzt sich der Trend der schlechter werdenden Schätzungen fort: Die Werte aus Tabelle 4.5 zeigen, dass nur noch ca. 24% der Schätzungen eine Abweichung um weniger als eine halbe Sekunde vom realen Wert haben. Da es bei diesem Schwierigkeitsgrad zu wenigen lokalen Berechnungen kam und dadurch nur wenige reale Laufzeiten für die lokale Berechnung vorlagen, sind die Werte für die lokalen Schätzungen nicht repräsentativ. Durch das Offloading wurden auf dem Samsung Galaxy S5 bei 1000 Spielzügen 26054302ms und auf dem Galaxy S6 bei 1000 Spielzügen 32181994ms eingespart, das entspricht ca. 7,24 bzw. 8,94 Stunden.

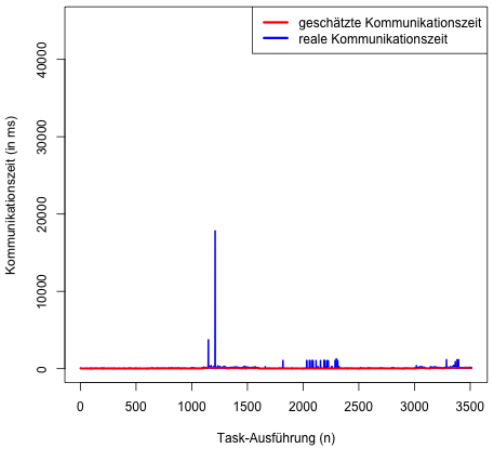
Die Offloading-Engine ist so konzipiert, dass die Schätzung für die lokale Ausführungszeit durch das Dividieren der Schätzung für die entfernte Ausführungszeit mit dem Faktor, um den der Server schneller ist als das Mobilgerät, berechnet wird. Da es im Schwierigkeitsgrad *Hard* in seltenen Fällen zu Laufzeiten des Servers von annähernd 60 Sekunden kommen kann und der Server je nach Leistungsfähigkeit des Mobilgerätes deutlich schneller ist, entstehen so schnell lokale Laufzeitschätzungen von mehreren Minuten. Das wiederum führt zu den exorbitanten Einsparungen von mehren Stunden.

4.1.4 Schätzung der Kommunikationszeit

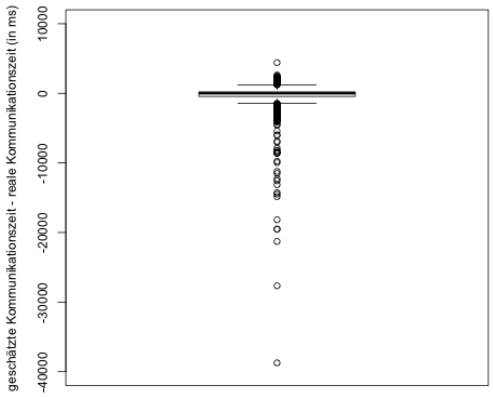
In diesem Abschnitt wird der Einfluss der Netzwerkdaten auf das Offloading-Verhalten untersucht. Dazu werden die Daten aus *Dataframe B* verwendet. Da im Mobilnetz die Bandbreite und der Ping deutlich schlechter als im WLAN waren, kam es im Mobilnetz zu deutlich weniger Offloading-Vorgängen als im WLAN. Daraus resultieren unterschiedlich große Datensätze. Um trotzdem vergleichbare Werte zwischen den verschiedenen Netzanbindungen darzustellen, werden in Tabelle 4.6 zusätzlich die Daten eines gekürzten WLAN-Datensatzes aufgelistet.



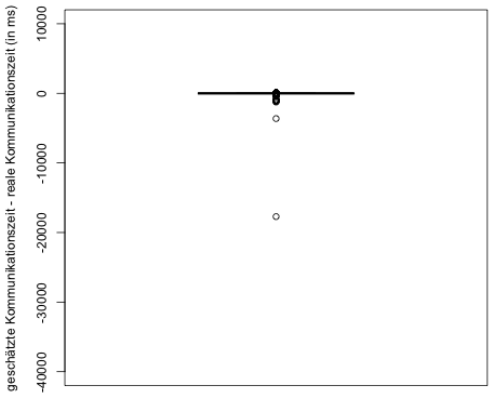
(a) Mobiles Netz



(b) WLAN



(c) Mobiles Netz



(d) WLAN

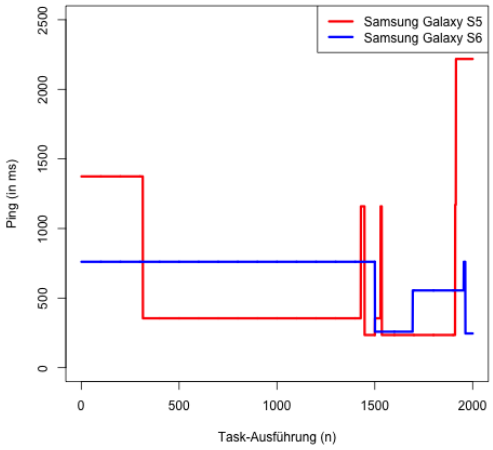
Abbildung 4.5: Geschätzte und reale Kommunikationszeit

	Mobiles Netz	**WLAN (vergleichbar)**	**WLAN (gesamt)**
Task-Ausführung (n)	946	946	3515
Durchschnittliche Abweichung (Schätzung - Laufzeit)	57,22ms	22,23ms	15,4ms
Median der Abweichung (Schätzung - Laufzeit)	-37,46ms	0,45ms	1,69ms
Standardabweichung (Kommunikationszeit)	4285.18ms	135,72ms	545.02ms
$\leq$ **500ms Abweichung**	584 (61,73%)	930 (98,31%)	3489 (99,26%)
$\leq$ **250ms Abweichung**	426 (45,03%)	928 (98,1%)	3482 (99,06%)
$\leq$ **10ms Abweichung**	29 (3,07%)	763 (80,66%)	2060 (58,61%)
$\leq$ **1ms Abweichung**	5 (0,53%)	108 (11,42%)	269 (7,65%)

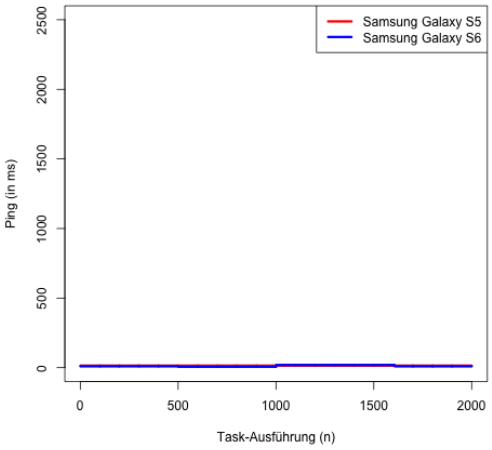
Tabelle 4.6: Kommunikationszeitschätzungen

Abbildung 4.5(a) und 4.5(b) zeigen die geschätzte und die reale Kommunikationszeit zwischen Smartphone und Server beider Smartphones bei einem durchgeführten Offloading-Vorgang. Abbildung 4.5(c) und 4.5(d) zeigen die Differenz aus geschätzter Kommunikationszeit und realer Kommunikationszeit. Auffallend sind die häufiger vorkommenden Fehlschätzungen im mobilen Netz in Abbildung 4.5(c).

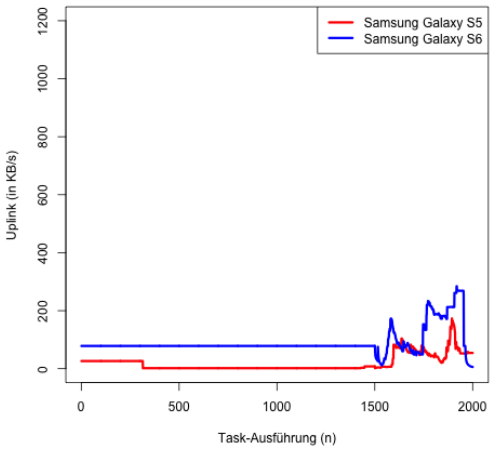
Abbildung 4.6 zeigt den Verlauf von Ping und Bandbreite während der Experimente. Auffällig ist das unrealistische Verhalten des Pings. Die Offloading-Engine stellt lediglich bei einer Veränderung des Netzwerktyps eine erneute Berechnung des Pings an. Wenn zum Beispiel die Anbindung des Mobilgerätes von 3G zu LTE wechselt, wird der Ping neu berechnet. Das führt dazu, dass bei gleichbleibenden Netzwerktyps der Ping auf dem Wert der ersten Berechnung verbleibt. Auch bei der Bandbreite zeigt sich ein ähnliches Problem: Zwar wird die Bandbreite, anders als der Ping, bei jedem Offloading-Vorgang aktualisiert, jedoch wird keine neue Messung vorgenommen, solange kein Offloading-Vorgang durchgeführt wird. Sollte nun bei einer ersten Messung eine sehr schlechte Bandbreite ermittelt werden und auf Grund dessen keine Offloading-Vorgänge ausgeführt werden, wird auch nicht erkannt, wenn sich die Netzwerkqualität verbessert. Als Folge der geringen Aktualisierungsrate und der im mobilen Netz deutlich stärker schwankenden Qualität ergeben sich für das mobile Netz schlechtere Schätzungen als für das WLAN. Die Daten in Tabelle 4.6 bestätigt diese Schlussfolgerung.



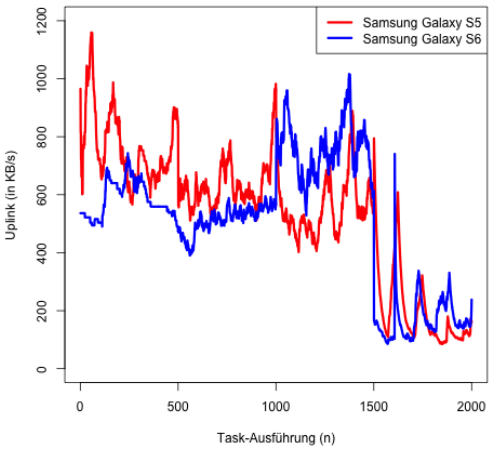
(a) Ping im mobilen Netz



(b) Ping im WLAN



(c) Bandbreite im mobilen Netz



(d) Bandbreite im WLAN

Abbildung 4.6: Ping und Bandbreite im mobilen Netz und im WLAN

4.1.5 Offloading-Entscheidungen

Die Beurteilung, ob ein Offloading-Vorgang gerechtfertigt war oder nicht, gestaltet sich als sehr schwierig. Theoretisch müsste jeder Zug unter exakt gleichen Bedingungen lokal auf dem Mobilgerät und entfernt auf dem Server ausgeführt werden. Wenn daraufhin die Summe aus Laufzeit des Servers und Kommunikationszeit geringer ausfällt als die Laufzeit für das Mobilgerät, war der Offloading-Vorgang gerechtfertigt und hat dem Mobilgerät CPU-Zeit eingespart. In der Praxis ist dieser Vergleich leider nicht möglich. Sollte es zu einem Offloading-Vorgang kommen, liegt keine lokale Laufzeit zum Vergleichen vor. Natürlich könnten Spielzüge, für die eine entfernte Ausführung vorgesehen sind, zusätzlich auf dem Mobilgerät ausgeführt werden und somit im Nachhinein beurteilt werden, ob die Entscheidung gerechtfertigt war oder nicht. Jedoch fließen durch das drahtlose Netzwerk und durch das Prozess-Scheduling der jeweiligen Betriebssysteme Variablen in das Experiment mit ein, die selbst unter Laborbedingungen nur schwer zu kontrollieren sind. Um trotzdem eine Aussage treffen zu können, wird die reale Server-Ausführungszeit aus *Dataframe B* mit der geschätzten, lokalen Laufzeit aus *Dataframe B* verglichen. Unter der Annahme, dass die Schätzungen der lokalen Laufzeit korrekt sind, kann daraufhin beurteilt werden, ob ein Offloading-Vorgang CPU-Zeit eingespart hat oder nicht. Da jedoch, wie oben gezeigt, die Schätzungen nur annähernd korrekt sind, ist die Aussage darüber, wie viele richtige und falsche Entscheidungen getroffen wurden, nur unter Vorbehalt aussagekräftig.

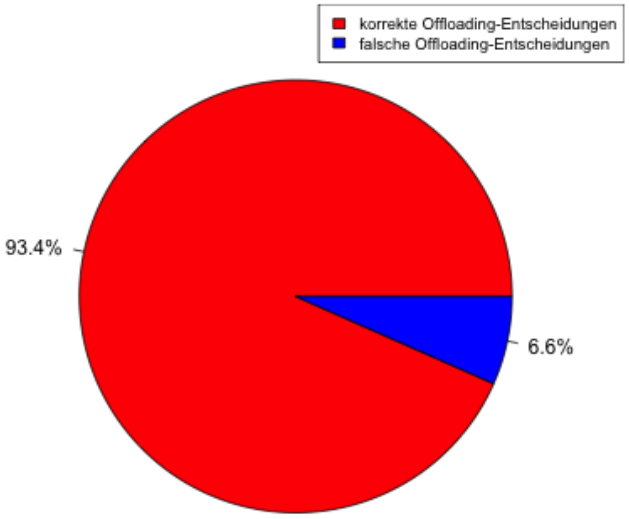


Abbildung 4.7: Offloading-Entscheidungen unter der Annahme, dass die lokalen Laufzeitschätzungen korrekt sind

Unter der oben genannten Annahme sind nur 6,6% aller Offloading-Vorgänge Fehlentscheidungen gewesen. Bei einer Gesamtzahl von 4461 Offloading-Vorgängen ergibt das 295 Fehlentscheidungen.

Eine Fehlentscheidung resultiert aus falschen Schätzungen der Laufzeit: Sollten die Schätzungen besagen, dass die lokale Ausführungszeit größer als die Summe aus entfernter Ausführungszeit und Kommunikationszeit ist, wird ein Offloading-Vorgang durchgeführt. Ist nun die tatsächliche Summe aus entfernter Ausführungs- und Kommunikationszeit jedoch größer als die lokale Ausführungszeit, liegt eine Fehlentscheidung vor. Eine korrekte Entscheidung hängt also maßgeblich von der Qualität der Schätzungen ab.

4.2 Schlussfolgerung

Die Fragen **Beeinflusst der im Spiel eingestellte Schwierigkeitsgrad die Offloading-Entscheidung und die Laufzeitschätzung?** und **Werden bei einer Verbindung zum WLAN mehr Offloading-Vorgänge ausgeführt als bei einer Verbindung zum mobilen Netz?** können mit einem Ja beantwortet werden. Die in Abbildung 4.3 dargestellten Daten zeigen eindeutig mehr Offloading-Vorgänge bei höherem Schwierigkeitsgrad und mehr Offloading-Vorgänge bei einer Verbindung zum WLAN als bei einer Verbindung zum mobilen Datennetz. Beides war zu erwarten und liegt zum einen an dem geringeren Ping, der höheren Bandbreite und der damit geringeren Kommunikationszeit zwischen Server und Mobilgerät im WLAN und zum anderen an der höheren Berechnungskomplexität bei steigendem Schwierigkeitsgrad. Bei einer Verbindung zum mobilen Datennetz entscheidet sich das System auf Grund einer höheren Kommunikationszeit eher gegen einen Offloading-Vorgang als bei einer WLAN Verbindung. Die Daten in Abbildung 4.4 und in Kapitel 4.1.3 zeigen, dass die Laufzeitschätzung mit steigendem Schwierigkeitsgrad ungenauer wird. Offensichtlich korreliert die Qualität der Schätzung mit der Standardabweichung der Laufzeit. Da Tabelle 4.1 zeigt, dass die Ausführungszeit desselben Spielzuges streut, müssen äußere Einflüsse wie das Prozess-Scheduling starken Einfluss auf die Ausführungszeit haben. Offloading-Tasks, die im Durschnitt eine hohe Ausführungszeit haben, bieten dem Betriebssystem wiederum mehr Möglichkeiten, durch Prozess-Scheduling den Ablauf zu beeinflussen. Daraus lässt sich schließen, dass im Allgemeinen die Schätzung der Laufzeit eines Offloading-Tasks mit steigender Laufzeit schlechter wird.

Die Antwort auf die Frage **Ist mit Hilfe des AESTET und der verwendeten Eingabe-ID eine genaue Vorhersage der Laufzeit möglich?** kann nicht mit einem eindeutigen ja oder nein beantwortet werden. Die in 4.1.3 dargestellten Daten belegen die Abweichungen der Laufzeitschätzungen. Jedoch halten sich diese Abweichungen zumindest für die ersten drei Schwierigkeitsgrade im Rahmen. Für eine Bewertung der Schätzungen werden folgende Kategorien definiert:

- Sehr gute Schätzung: weniger als 1ms Abweichung
- Gute Schätzung: zwischen 1ms und 10ms Abweichung
- Befriedigende Schätzung: zwischen 10ms und 250ms Abweichung
- Unbefriedigende Schätzung: zwischen 250ms und 500ms Abweichung
- Schlechte Schätzung: mehr als 500ms Abweichung

Auf alle Schätzungen angewendet, bekommen 6,49% der Schätzungen die Bewertung sehr gut, 38,15% gut, 34,1% befriedigend, 2,31% unbefriedigend und 18,95% schlecht. Die Daten in Kapitel 4.1.1 zeigen, dass die reale Laufzeit trotz gleicher Eingabe eine, je nach Schwierigkeitsgrad, leichte bis starke Streuung aufweist. Daher ist eine Laufzeitschätzung mit Hilfe einer Eingabe-ID und dem AESTET nur mit Abweichung möglich. Wie in Kapitel 4.1.5 erläutert, wurden trotz alledem wenig falsche Entscheidungen getroffen. Nur 6,6% der Entscheidungen wurden als falsch eingestuft. Somit muss die Frage **Wurden fehlerhafte Entscheidungen getroffen?** mit ja, es wurden einige wenige Fehlentscheidungen getroffen beantwortet werden. Jedoch ist der Anteil an Fehlentscheidungen sehr gering.

Die Schätzungen für die Kommunikationszeit sind durch die seltene Aktualisierung von Ping und Bandbreite noch verbesserungswürdig. Besonders im mobilem Netz wäre durch eine häufigere Messung der beiden Werte eine bessere Schätzung der Kommunikationszeit möglich. Nach den oben definierten Kategorien sind 6,14% der Schätzungen als sehr gut, 40,69% als gut, 40,78% als befriedigend, 3,7% als unbefriedigend und 8,7% als schlecht zu bewerten.

Die Beantwortung der Forschungsfrage **Wird durch mobiles Offloading der Energieverbrauch des Mobilgerätes beim Spielen eines Schachspiels reduziert?** lässt sich auf Grund der schon erwähnten fehlenden Möglichkeit den direkten Energieverbrauch von Android Smartphones mittels Software zu messen, leider nicht direkt beantworten. Wenn die Offloading-Engine die Entscheidung trifft, eine Berechnung an den Server auszulagern, spart das Smartphone im optimalen Fall CPU-Zeit. Sobald die Summe aus realer Kommunikationszeit und realer Serverausführungszeit geringer ist als es die reale Ausführungszeit auf dem Mobilgerät wäre, wird dadurch die CPU entlastet und Energie gespart. Natürlich verbraucht das Warten auf die Antwort des Servers beim Mobilgerät deutlich weniger Energie als der Energieverbrauch, der anfallen würde, wenn das Mobilgerät in derselben Zeit die Berechnung durchführen würde. Daher müsste die Zeit, die benötigt wird, um auf die Antwort des Server zu warten geringer gewichtet werden als die Zeit, die benötigt wird um die Berechnung selbst durchzuführen. Diese Gewichtung lässt sich jedoch nur ermitteln, wenn genaue Daten zum Energieverbrauch in den verschiedenen Zuständen vorliegen. Da diese nicht verfügbar sind, wurde hier von einer unterschiedlichen Gewichtung abgesehen und die Offloading-Entscheidungen werden ausschließlich auf Grundlage der Laufzeiten getroffen. Da eine gute Offloading-Entscheidung maßgeblich von den Schätzungen der Laufzeit und der Kommunikationszeit abhängt, wird die Qualität dieser als Indikator für einen geringeren Energieverbrauch herangezogen. Zusammenfassend ist also zu sagen, dass eine gute Abschätzung der Laufzeiten und ein Server, der in der Berechnung der Offloading-Tasks deutlich schneller ist als das Mobilgerät, zu korrekten Offloading-Entscheidungen und somit zu häufigen Offloading-Vorgängen und einem geringeren Energieverbrauch führt. Die Experimente haben gezeigt, dass die meisten Schätzungen keine zu großen Abweichungen zu den Reallaufzeiten aufweisen und unter der in 4.1.5 aufgestellten Annahme, dass die geschätzten Laufzeiten des Mobilgerätes korrekt sind, wurde auch gezeigt, dass die meisten der Offloading-Entscheidungen richtig getroffen wurden. Deshalb kann die Forschungsfrage nach der Energieeinsparung mit ja beantwortet werden.

In Abschnitt 4.1.3 zeigen die durchschnittlichen Laufzeitabweichungen, dass die Schätzungen der Laufzeit für den Server genauer sind als die Schätzungen für die Laufzeit des Mobilgerätes. Über die Gründe kann hier ohne weitere Experimente nur spekuliert werden. Auffallend

ist, dass dieser Unterschied bei beiden Smartphones auftritt. Möglicherweise ist das Prozess-Scheduling des Betriebssystems Android dafür verantwortlich und erschwert so eine genaue Vorhersage. Dagegen sprechen die in Tabelle 4.1 gezeigten Daten. Bei der wiederholten Ausführung eines einzigen Zuges ist die lokale Laufzeit in den ersten drei Schwierigkeitsgraden stabiler als die entfernte. Eine mögliche Ursache könnte ein unterschiedliches *Random-Access Memory* (RAM)-Management der beiden Betriebssysteme sein. Um abschließend zu klären, was die genauen Gründe sind, bedarf es weiterer Experimente in denen RAM-Management und Prozess-Scheduling kontrolliert werden.

Abbildung 4.3 zeigt, dass die Leistungsfähigkeit des Mobilgerätes starken Einfluss auf die Offloading-Entscheidung hat. Da ein leistungsstärkeres Mobilgerät eine Berechnung schneller durchführen kann als ein leistungsschwächeres und die Offloading-Entscheidung unter anderem von der Laufzeit der Berechnung abhängt, war dieses Ergebnis zu erwarten. Besonders auffällig ist der Unterschied im Schwierigkeitsgrad *Very Easy* bei Verbindung zum WLAN. Während das Samsung Galaxy S5 die meisten der Berechnungen vom Offloading-Server durchführen lassen hat, berechnet das leistungsstärkere Samsung Galaxy S6 die meisten Berechnungen selbst. Mit zunehmenden Schwierigkeitsgrad werden auch die lokalen Berechnungen der beiden Smartphones weniger. Die Frage **Beeinflusst die Leistung des Mobilgerätes die Offloading-Entscheidung?** kann also mit ja beantwortet werden.

Auch wenn es an einigen Stellen noch Verbesserungspotenzial hinsichtlich besserer Laufzeit- und Kommunikationszeitschätzungen gibt, um so noch mehr CPU-Zeit und damit Energie zu sparen, kann die Entwicklung eines energieeffizienten Schachspiels für Android Smartphones im großen und ganzen als erfolgreich angesehen werden.

4.3 Zusammenfassung

In dieser Arbeit wurde dem Leser zuerst eine allgemeine Einführung in das Thema mobiles Offloading verschafft. Darauf folgte eine detaillierte Auseinandersetzung mit den aktuellen Forschungsgebieten: Neustart in mobilem Offloading, Offloading-Entscheidung und der Untersuchung verschiedener Offloading-Systeme. Im zweiten Teil wurde die Entwicklung und Implementierung eines Schachspiels mit Unterstützung für mobiles Offloading beschrieben. Darauf aufbauend wurden Forschungsfragen gestellt und der Versuchsaufbau der Experimente beschrieben. Nach der Beantwortung der Forschungsfragen folgte im letzten Teil der Arbeit die Visualisierung und Interpretation der durch die Experimente erhaltenen Daten und die daraus resultierenden Schlussfolgerungen.

4.4 Ausblick

In diesem Abschnitt werden Anregungen zur weiteren Forschung und Entwicklung in Bezug auf die Offloading-Engine und das Offloading-Schachspiel im speziellen gemacht. Auch wird hier die Forschungsfrage nach den Verbesserungen der Offloading-Engine und des Schachspiels beantwortet.

Wie schon in Abschnitt 4.1.4 und 4.2 beschrieben, wird Bandbreite und Ping nur unzureichend oft aktualisiert. Um die Schätzung der Kommunikationszeit weiter zu verbessern, könnten beide Werte zusätzlich in regelmäßigen Abstand aktualisiert werden. Dabei ist es

wichtig, einen guten Kompromiss zwischen Aktualisierungsintervall und zusätzlichen Overhead durch die Messungen zu finden. Anhand von Experimenten könnte in Erfahrung gebracht werden, welcher Zeitintervall für eine Aktualisierung optimal ist.

Die aus maximaler Suchtiefe, Anzahl legaler Züge und Anzahl der verbleibenden Figuren zusammengesetzte Eingabe-ID für den Offloading-Task kann noch verbessert werden. Die wichtigste Größe bei der Einschätzung der Laufzeit ist die durch den Schwierigkeitsgrad begrenzte Suchtiefe. Die Anzahl der legalen Züge zu Beginn eines Zuges bestimmt, wie viele Kinder die Wurzel des Suchbaums hat und somit den weiteren Aufbau des Baumes. Ansetzen könnte man bei der Anzahl der verbleibenden Figuren auf dem Schachbrett. Wenn dieser Wert nicht nur eine Aussagekraft darüber hätte, welche Figuren noch auf dem Brett stehen, sondern auch was für Figuren, dann könnte die Schätzung der Laufzeit mit Hilfe des AESTET möglicherweise noch verbessert werden. Wenn zum Beispiel zwei Könige und vier Bauern auf dem Brett stehen, ist der daraus resultierende Suchbaum durch die geringere Anzahl an legalen Zügen auch nach dem ersten Halbzug deutlich kleiner als würden zwei Könige, zwei Damen und zwei Türme auf dem Spielfeld stehen. Interessant wäre ein Experiment, in dem die Eingabe-ID auf unterschiedliche Art berechnet werden würde und danach ein Vergleich der Qualität der Schätzungen, je nach Art der Berechnung der Eingabe-ID, erfolgen würde. Dadurch könnte auf experimentelle Weise die beste Version ermittelt werden.

Um den tatsächlichen Energieverbrauch zu bestimmen, existieren zwei Anwendungen für Android: *PowerTutor*[4] und *Trepn Profiler*[5]. Leider unterstützt keine der beiden Anwendungen die für die Experimente zur Verfügung stehenden Smartphones. Daher gibt es keine Garantie für die Korrektheit der Daten und eine Analyse kann wissenschaftlichen Standards nicht gerecht werden. Eine zukünftige Untersuchung des Schachspiels mit einem von den Analyseanwendungen unterstützten Mobilgeräten würde eine genauere Aussage über den tatsächlichen Energieverbrauch zulassen.

Da äußere Einflüsse auf Netzwerke, speziell drahtlose Netzwerke, und das Prozess-Scheduling eines Betriebssystems in Experimenten schwer zu kontrollierende Variablen sind, wäre es spannend, die hier durchgeführten Experimente in einem Simulator zu wiederholen. Einzelne Offloading-Vorgänge könnten unter exakt den selben Bedingungen durchgeführt werden und so eindeutig bestimmt werden, ob eine entfernte Ausführung Zeit- bzw. Energieersparnisse erbracht hat oder nicht.

Wie schon in Abschnitt 4.2 beschrieben, müsste die Zeit, die das Mobilgerät auf eine Antwort des Offloading-Servers wartet, anders gewichtet werden als die Zeit, die das Mobilgerät selbst für die Berechnung benötigt. Auf Grund der fehlenden Möglichkeit, den exakten Energieverbrauch per Software zu bestimmen, bietet es sich an, diesen Verbrauch mit Hilfe externer Hardware zu ermitteln. Daraus lässt sich schließen, um welchen Faktor beim Warten oder Transferieren der Daten weniger bzw. mehr Energie verbraucht wird als bei der eigentlichen Berechnung. Da diese Werte sich bei verschiedenen Mobilgeräten unterscheiden werden, müsste so eine Messung für jedes Modell neu durchgeführt werden. Hilfreich wäre eine Datenbank, in der diese Werte festgehalten werden und dann in Anwendungen mit mobilem Offloading zum Einsatz kommen könnte.

Die Offloading-Engine kann um das in Kapitel 2.3 beschriebene Neustarten eines Tasks

[4] *PowerTutor*: `http://ziyang.eecs.umich.edu/projects/powertutor/index.html` (Abruf: 23.01.2016)
[5] *Trepn Profiler*: `https://developer.qualcomm.com/software/trepn-power-profiler` (Abruf: 23.01.2016)

erweitert werden. Bisher ist eine recht simple Art des lokalen Neustarts in der Offloading-Engine implementiert: Sollte innerhalb einer bestimmten Zeit vom Server keine Antwort eingehen, wird der Tasks lokal noch einmal ausgeführt. Sollte dieser Fall eintreten, erhöht sich der Energieverbrauch und die Ausführungszeit drastisch. Durch das entfernte oder lokale Neustarten eines Tasks anhand bestimmter Kriterien wird das ganze System weniger anfällig für Fehler, die auf schwankender Netzwerkqualität beruhen.

Der Offloading-Server ist nicht für die Bearbeitung von Anfragen mehrere Clients ausgerichtet. Für die Beantwortung der im Rahmen dieser Arbeit gestellten Forschungsfragen genügt dieser Ansatz. Für den praktischen Betrieb müsste eine Warteschlange die Abarbeitung der ankommenden Tasks regulieren.

Die Analyse der Schätzungen im Schwierigkeitsgrad *Hard* in Kapitel 4.1.3 zeigt, dass die Berechnung einiger Züge bei nicht erreichbarem Server teilweise mehrere Minuten auf dem Mobilgerät dauern würden. Was die Benutzerfreundlichkeit des Spiels angeht, ist das nicht tolerierbar. Abhilfe kann durch eine zeitliche Begrenzung bei der Suche geschafft werden. Wie in der Anforderungsanalyse in Kapitel 3.1.3 beschrieben, wurde der Schwierigkeitsgrad im ursprünglichen *CuckooChess* durch eine unterschiedliche Zeitbegrenzungen der Suche gesetzt. Die Funktionalität ist demnach noch vorhanden und kann einfach für eine maximale Suchzeit genutzt werden.

Abschließend ist zu sagen, dass mobiles Offloading großes Potential hat, die hohen Anforderungen aktueller Hard- und Software an den Akku aufzufangen. Auf Grund immer besserer Netzabdeckung des mobilen Datennetzes und schnellerer WLAN-Netzwerke lohnt sich Offloading zunehmend für viele Berechnungen und reduziert dadurch den Energieverbrauch moderner Mobilgeräte. Die daraus resultierende längere Zeitspanne zwischen den Aufladevorgängen erhöht den Komfort und die Mobilität des Benutzers. Man darf gespannt sein, wie sich mobiles Offloading aus wissenschaftlicher Sicht weiterentwickelt und wie es sich auf dem freien Markt etablieren wird.

Anhang A: *OffloadingChess* und Funktionalität für den Server

Das Schachspiel und die Implementierung für den Server ist unter folgender Adresse auf *GitHub* zu finden:

`https://github.com/chbruns/OffloadingChess.git`

Die Implementierung des Schachspiels befindet sich als Android Studio Projekt im Verzeichnis *OffloadingChess*. Die Implementierung der Funktionalität für den Server befindet sich als JAR im Verzeichnis *ServerJAR*. Der Webinterface des Servers selbst befindet sich unter folgender Adresse:

`https://www.mi.fu-berlin.de/offload/`

Über den passwortgeschützten Managementbereich des Webinterfaces kann das JAR auf dem Server installiert werden. Die Klasse *ChessController* beinhaltet die vom Server benötigte Methode. Alternativ kann die Implementierung des Servers unter Downloads heruntergeladen werden und der Server selbst betrieben werden. Anschließend muss in den Einstellungen von *OffloadingChess* die Adresse des Servers eingegeben werden.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[BH06] Binder, Walter ; Hulaas, Jarle: Using Bytecode Instruction Counting As Portable CPU Consumption Metric. In: *Electron. Notes Theor. Comput. Sci.* 153 (2006), Mai, Nr. 2, S. 57–77

[CBC+10] Cuervo, Eduardo ; Balasubramanian, Aruna ; Cho, Dae-ki ; Wolman, Alec ; Saroiu, Stefan ; Chandra, Ranveer ; Bahl, Paramvir: MAUI: Making Smartphones Last Longer with Code Offload. In: *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*. New York, NY, USA : ACM, 2010 (MobiSys '10). – ISBN 978–1–60558–985–5, S. 49–62

[CIM+11] Chun, Byung-Gon ; Ihm, Sunghwan ; Maniatis, Petros ; Naik, Mayur ; Patti, Ashwin: CloneCloud: Elastic Execution Between Mobile Device and Cloud. In: *Proceedings of the Sixth Conference on Computer Systems*. New York, NY, USA : ACM, 2011 (EuroSys '11). – ISBN 978–1–4503–0634–8, S. 301–314

[Eir11] Eiríksson, Hrafn: *Investigation of Multi-Cut Pruning in Game-Tree Search*. Reykjavik University, 2011

[Fer13] Ferreira, Diogo R.: The Impact of Search Depth on Chess Playing Strength. In: *ICGA Journal* (2013)

[Gri13] Griera, Martí: *Improving the reliability of an offloading engine for Android mobile devices and testing its performance with interactive applications*, Freie Universität Berlin, Masterarbeit, Oktober 2013

[Hei98] Heinz, Ernst A.: Extended Futility Pruning. In: *ICCA Journal* 21 (1998), S. 75 – 83

[HK00] Hendrickson, Bruce ; Kolda, Tamara G.: Graph Partitioning Models for Parallel Computing. In: *Parallel Comput.* 26 (2000), November, Nr. 12, S. 1519–1534

[Kop06] Koppitz, Stefan: *Implementierung eines Schachprogramms*. Technische Universität Dresden, Seminar: Neuronale Netze und Fallbasiertes Schließen, August 2006

[KPKB12] Kemp, Roelof ; Palmer, Nicholas ; Kielmann, Thilo ; Bal, Henri: Cuckoo: A Computation Offloading Framework for Smartphones. In: Gris, Martin (Hrsg.) ; Yang, Guang (Hrsg.): *Mobile Computing, Applications, and Services* Bd. 76, Springer Berlin Heidelberg, 2012 (Lecture Notes of the Institute for Computer

Sciences, Social Informatics and Telecommunications Engineering). – ISBN 978–3–642–29335–1, S. 59–79

[MN10] Miettinen, Antti P. ; Nurminen, Jukka K.: Energy Efficiency of Mobile Clients in Cloud Computing. In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. Berkeley, CA, USA : USENIX Association, 2010 (HotCloud'10), S. 4–4

[OYL06] Ou, Shumao ; Yang, Kun ; Liotta, Antonio: An Adaptive Multi-Constraint Partitioning Algorithm for Offloading in Pervasive Systems. In: *Fourth Annual IEEE International Conference on Pervasive Computing and Communications* (2006)

[Reu09] Reul, F.M.H.: *New architectures in computer chess*, Tilburg University, Diss., 2009

[Rip13] Ripoll, Joan M.: *Improving the performance and usability of an offloading engine for Android mobile devices with application to a chess game*, Freie Universität Berlin, Masterarbeit, Oktober 2013

[RMS15] Riddle, Bryan ; Manalo-Schwarz, Sydney: *Java und Dalvik Bytecode - Ein Vergleich*. Fachhochschule Köln - Fakultät für Informatik und Ingenieurwissenschaften, 2015

[Sha88] Shannon, C. E.: Computer Chess Compendium. New York, NY, USA : Springer-Verlag New York, Inc., 1988. – ISBN 0–387–91331–9, Kapitel Programming a Computer for Playing Chess, S. 2–13

[TS02] Tilevich, Eli ; Smaragdakis, Yannis: J-Orchestra: Automatic Java Application Partitioning. In: *Proceedings of the 16th European Conference on Object-Oriented Programming*. London, UK, UK : Springer-Verlag, 2002 (ECOOP '02). – ISBN 3–540–43759–2, S. 178–204

[Wan14] Want, Roy: The Power of Smartphones. In: *IEEE Pervasive Computing, July - September* (2014), S. 76 – 77

[Wan15] Wang, Qiushi: *Restart in Mobile Offloading*. Berlin, Freie Universität Berlin, Diss., September 2015

[WSS+15] Wu, Huaming ; Seidenstücker, Daniel ; Sun, Yi ; Nieto, Carlos M. ; Knottenbelt, William J. ; Wolter, Katinka: An Optimal Offloading Partitioning Algorithm in Mobile Cloud Computing. In: *CoRR* abs/1510.07986 (2015)

[Wu15] Wu, Huaming: *Analysis of Offloading Decision Making in Mobile Cloud Computing*. Berlin, Freie Universität Berlin, Diss., September 2015

[WWHU06] Winands, Mark H. ; Werf, Erik C. d. ; Herik, H. J. d. ; Uiterwijk, Jos W.: The Relative History Heuristic. In: Herik, H.Jaap van d. (Hrsg.) ; Björnsson, Yngvi (Hrsg.) ; Netanyahu, NathanS. (Hrsg.): *Computers and Games* Bd. 3846. Springer Berlin Heidelberg, 2006. – ISBN 978–3–540–32488–1, S. 262–272

[Ös] Österlund, Peter: *CuckooChess 1.12 - A Java Chess Program*. – `http://web.comhem.se/petero2home/javachess/` (abgerufen am 08.10.2015)