View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Online Publikationen der Universität Stuttgart

Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3662

Konzepte und Methoden für die Verwendung eines Caches in mobilen Code Offloading Systemen

Stefan Schweizer

Studiengang: Informatik

Prüfer/in: Prof. Dr. rer. nat. Kurt Rothermel

Betreuer/in: Dipl.-Inf. Florian Berg

Beginn am: 5. Mai 2014

Beendet am: 4. November 2014

CR-Nummer: C.2.4

Kurzfassung

In dieser Arbeit wird der Frage nachgegangen, ob eine Caching Komponente ein Code Offloading System im mobilen Umfeld sinnvoll ergänzen kann. Dazu wird zunächst die Caching Komponente, in Form eines über das Netzwerk erreichbaren Server, konzipiert und implementiert. Zur Anbindung an das Code Offloading System wird ein Netzwerkprotokoll entworfen und in einem zuvor schon vorhandenen Offloading System implementiert. Zuletzt werden die Komponenten in unterschiedlichen Szenarien evaluiert.

Inhaltsverzeichnis

1 Einleitung 9
1.1 Motivation . . . 9
1.2 Ziel . . . 11

2 Grundlagen 13
2.1 Code Offloading . . . 13
2.2 Caching . . . 18

3 Verwandte Arbeiten 21

4 Konzept 25
4.1 Ziele und Rahmenbedingungen . . . 25
4.2 Offloading System . . . 26
4.3 Architektur . . . 28
4.4 Anwendungsspezifische Optimierungen . . . 30
4.5 Kodierung . . . 32
4.6 Kommunikationsprotokoll . . . 34

5 Implementierung 39
5.1 Offloading System . . . 39
5.2 Caching Server . . . 44

6 Evaluation 49
6.1 Testumgebung . . . 49
6.2 Getestete Anwendungen . . . 50
6.3 Szenarien . . . 51
6.4 Testergebnisse . . . 55

7 Zusammenfassung und Ausblick 67

Literaturverzeichnis 69

Abbildungsverzeichnis

1.1 Leistungsverbesserungen einiger Smartphonekomponenten seit 1990 [PS05] . . . 10

2.1 Kommunikation zwischen Endgerät und Offloading Instanzen . . . 14
2.2 Offloading im Spannungsfeld zwischen Kommunikationsaufwand D und Berechnungsaufwand C [KL10] . . . 16

4.1 Methoden Offloading in der Jikes RVM . . . 27
4.2 Netzwerkkommunikation zwischen Cache und Offloading Client und Server . . . 32

5.1 Datenstruktur zum Speichern von Methoden, deren Parameter und Rückgabewert . . . 47

6.1 Beispiel Gesichtserkennung . . . 51
6.2 Evaluationsaufbau . . . 52
6.3 Ausführungszeiten für 5 lokale Schachzüge . . . 56
6.4 Ausführungszeiten für 5 Schachzüge auf dem Netbook als Offloading Client . . . 57
6.5 Ausführungszeiten für 5 Schachzüge auf dem Laptop als Offloading Client . . . 58
6.6 Ausführungszeiten für 5 Schachzüge auf dem Netbook mit Offloading und Cache . . . 59
6.7 Ausführungszeiten für 5 Schachzüge auf dem Laptop mit Offloading Cache . . . 60
6.8 Netzwerkverkehr des Offloading Clients für 5 Schachzüge mit Offloading und Caching . . . 61
6.9 Ausführungszeiten für die Gesichtserkennung lokal . . . 62
6.10 Ausführungszeiten für die Gesichtserkennung von fünf Bildern mit dem Netbook als Offloading Client . . . 63

6.11 Ausführungszeiten für die Gesichtserkennung von fünf Bildern mit dem Laptop als Offloading Client 64
6.12 Ausführungszeiten für die Gesichtserkennung auf dem Netbook mit Offloading und Caching 65
6.13 Ausführungszeiten für die Gesichtserkennung auf dem Laptop mit Offloading und Caching 66
6.14 Netzwerkverkehr bei der Gesichtserkennung 66

Tabellenverzeichnis

6.1 Zuordnungen von Computern und Offloading Systeme 53
6.2 Größen der bei der Gesichtserkennung verwendeten Bilder 60

Verzeichnis der Listings

5.1 Algorithmus zur Bestimmung des besten Caches 42
5.2 Parameter Annotations für Bereichsanfragen 44

1 Einleitung

1.1 Motivation

Der Smartphone Boom der letzten Jahre hat hat dazu geführt, dass bereits Mittelklasse Modelle über eine leistungsfähige Hardwareaustattung verfügen. So ist zum Beispiel das Nexus 5 von Google bereits mit einem Quad-Core-Prozessor vom Typ Snapdragon 800 [1] und 2 GB Arbeitsspeicher ausgestattet. Hinzu kommt eine immer besser Versorgung mit schnellen Mobilfunktechniken wie LTE. Anwendungsentwickler sind somit heute in der Lage, auch anspruchsvolle Anwendung auf mobilen Endgeräten zu realisieren. Doch gerade diese anspruchsvollen, rechenintensiven Anwendungen steht eine limitierte Energieversorgung entgegen. Bereits heute müssen die meisten Smartphones fast täglich geladen werden. Abbildung 1.1 zeigt die Leistungsentwicklung einiger Smartphone Komponenten. Im Gegensatz zu allen anderen Komponenten konnte die Energiedichte der Batterien nur marginal gesteigert werden. Sollte kein gravierender Durchbruch in der Energieversorgung erreicht werden, so wird dieser auch in naher Zukunft der limitierende Faktor bleiben.

Entwickler von neuen, anspruchsvollen Anwendungen und Diensten stehen somit vor einem Dilemma. Einerseits würden sie gerne ihre Anwendung auf einem Smartphone realisieren und die dort vorhandene Rechenleistung nutzen. Andererseits sind die Benutzer kaum dazu bereit, Abstriche an ihrer bereits heute recht kurzen Akkulaufzeit ihrer Smartphones hinzunehmen.

Heutzutage werden unterschiedliche Ansätze verfolgt um Energie zu sparen. So wird versucht mit jeder neuen Prozessorgeneration die Transistoren zu verkleinern. Jedoch führt diese Entwicklung auch zu leistungsfähigeren Prozessoren die die Energieeinsparung wieder zu Nichte machen. Ein weitere Methode in mobilen Geräte Energie zu sparen ist es wenn immer möglich einzelne Komponenten oder das gesamte Gerät in einen Schlafmodus zu versetzen. So kann sich zB. der Anwendungsprozessor in einen

[1] https://www.qualcomm.com/products/snapdragon/processors/800

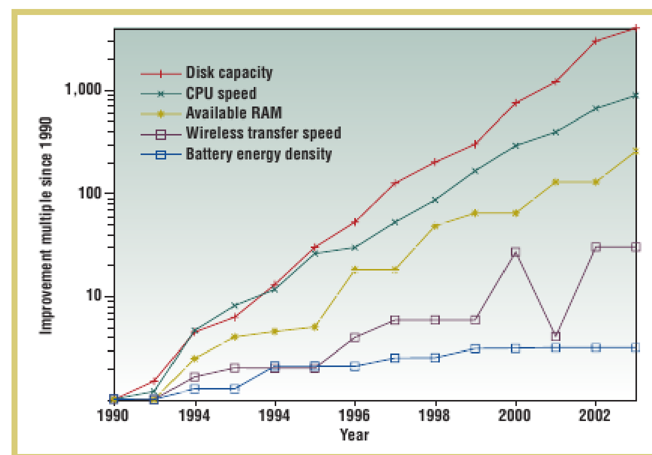

Abbildung 1.1: Leistungsverbesserungen einiger Smartphonekomponenten seit 1990 [PS05]

Schlafzustand versetzt werden während der Baseband Prozessor auf eingehende Mobilfunkkommunikation lauscht. Der Baseband Prozessor weckt dann den Hauptprozessor falls zu verarbeitende Daten eintreffen. Die in dieser Arbeit untersuchte Möglichkeit ist es, bestimmte Berechnungen gar nicht auf dem mobile Endgerät auszuführen.

Beim Code Offloading werden rechenintensive Arbeitsschritte nicht mehr auf dem Smartphone ausgeführt sondern in die Cloud ausgelagert. Das Smartphone spart dabei durch die nicht ausgeführten Arbeitsschritte Energie und kann von einer leistungsfähigen Serverinfrastruktur profitieren welche die Arbeitsschritte schneller ausführen kann. Die höheren Energiekosten für das Senden und Empfangen sowie mögliche Kosten für das Mobilfunknetz und die Serverinfrastruktur müssen dabei in Relation gesetzt werden.

Caching ist in der Informatik eine weit verbreitete Methode zur Effizienzsteigerung. Bereits 1995 postulierte Van Jacobson[Jac95] dass das Web ohne Caching nicht überleben würde. Große Internetunternehmen wie Google oder Amazon betreiben heutzutage eine weltweit verteilte Caching Infrastruktur, um ihren Kunden einen möglichst effizienten Zugriff auf ihr Angebot zu gewährleisten. Dienstleister wie Akamei cachen für ihre Kunden Inhalte in ihrem weltweiten Content Delivery Network und machen damit zB. das Updaten von Millionen von Smartphones zum selben Zeitpunkt erst möglich.

Es ist daher naheliegend, auch Code Offloading mit einem Caching System auszustatten.

1.2 Ziel

In dieser Arbeit soll daher der Frage nachgegangen werden, ob eine Caching Komponente ein Code Offloading System im mobilen Umfeld sinnvoll ergänzen kann.

Dazu soll zunächst eine Caching Komponente, in Form eines über das Netzwerk erreichbaren Servers, konzipiert und entwickelt werden. Zur Anbindung das Code Offloading System wird ein Netzwerkprotokoll entworfen und im Offloading System implementiert.

Zuletzt soll das System evaluiert werden.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen: In diesem Kapitel wird auf die Grundlagen von Code Offloading und Caching eingegangen.

Kapitel 4 – Konzept: In diesem Kapitel wird ein Konzept für die Caching Komponente und für die Anbindung an das Offloading System entworfen.

Kapitel 5 – Implementierung: Dieses Kapitel stellt die Implementierung des Caching Servers vor. Zudem geht es auf die Änderungen am Offloading System ein.

Kapitel 6 – Evaluation: In diesem Kapitel wird der Caching Server anhand unterschiedlicher Szenarien getestet.

Kapitel 7 – Zusammenfassung und Ausblick Dieses Kapitel fasst die Arbeit zusammen und gibt einen Ausblick auf eine mögliche Weiterentwicklung

2 Grundlagen

In diesem Kapitel sollen die Grundlagen des Code Offloading und Caching erläutert werden. Dabei hat dieses Kapitel nicht den Anspruch, allgemein und vollständig zu sein. Es soll jedoch die in dieser Arbeit getroffenen Entscheidungen bezüglich Konzeption und Implementierung untermauern.

2.1 Code Offloading

Mit dem Aufkommen der Smartphones in den letzten Jahren stieg auch auch das Interesse, sich mit Offloading Systemen zu beschäftigen. Gerade die beschränkten Ressourcen dieser mobilen Endgeräte machen sie interessant für die unterschiedlichsten Ansätze. Fasst man Offloading etwas breiter ist es ein schon lange verfolgtes Konzept. So gibt es seit langem sogenannte Thin Clients, bei denen sich das Endgerät nur um die Darstellung der Benutzeroberfläche kümmert. Die eigentliche Hauptarbeit übernehmen Server im Hintergrund. Unter diesem Gesichtspunkt kann auch das World Wide Web gesehen werden, wobei in den letzten Jahren mit Javascript immer mehr Funktionalität in den Client wandert.

Heutige Offloading Ansätze ermöglichen eine feingranularere Aufteilung der Anwendung als nur die Benutzeroberfläche abzukoppeln.

2.1.1 Definition

Unter Code Offloading wird im Folgenden das Auslagern einer Berechnungseinheit auf einen oder mehrere Instanzen verstanden.

Eine Berechnungseinheit kann dabei je nach Offloading System zB. eine Methode, ein Programm oder eine komplette virtuelle Maschine sein.

Eine Instanz ist ein System, das eine Berechnungseinheit verarbeiten kann. Wie ein Instanzen technisch realisiert sind spielt dabei für das grundlegende Konzept keine

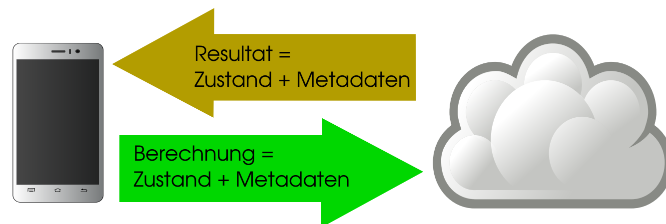

Abbildung 2.1: Kommunikation zwischen Endgerät und Offloading Instanzen

Rolle. Die Instanz muß in der Lage sein, die Berechnungseinheit entgegenzunehmen und auszuführen.

Die beim Code Offloading übermittelten Daten bestehen aus einem 2-Tupel $\alpha(Z, M)$. Z beschreibt den Zustand, in dem sich das auszulagernde System gerade befindet. Dieser Zustand kann dabei zB. ein komplettes Speicherabbild sein oder auch nur ein Delta zwischen jetzigem Ausführungsstand und dem Startzustand.

Die Metadaten M beschreiben, wie das Offloading stattfinden soll. Sie entscheiden wie die Zustandsdaten kodiert sind oder an welchen Endpunkt das Resultat geliefert werden soll. M beinhaltet alle Daten, die benötigt werden, um den übermittelten Zustand in den gewünschten Endzustand zu überführen.

Dieses Tupel α wird von einem Offloading Client an einen Offloading Server übertragen. Der Offloading Client läuft dabei auf dem Endgerät. Dieser kann je nach Offloading System auf unterschiedliche Arten realisiert sein. Er kann zB. bereits bei der Entwicklung als Bibliothek in das Anwendungsprogramm eingebunden werden oder Teil einer Virtuellen Maschine sein, die das eigentliche Anwendungsprogramm ausführt.

Der Offloading Server nimmt das Tupel entgegen und führt dieses aus. Ist dies erledigt wird ein Ergebnistupel an den Client zurückgeliefert. Der Offloading Client setzt nun seine Verarbeitung mit Hilfe des neuen Zustandes fort.

2.1.2 Anwendungsklassen

Unterschiedliche Anwendungen profitieren mehr oder weniger von Code Offloading. In [CM09] werden unterschiedliche Typen von Anwendungen unterschieden, die von einem Code Offloading profitieren könnten. Anwendungen müssen dabei nicht nur exclusive in eine Anwendungsklasse fallen.

Auslagern der Primärfunktionalität

In diese Kategorie fallen rechenintensive Anwendungen. Das bearbeiten von Bilder, Audiodaten oder die Berechnung von Spielzügen im Schach. Die Anwendung kann dabei leicht in zwei Teile getrennt werden. Das Userinterface und der dazu gehörige Programmcode verbleibt auf dem mobilen Endgerät und werden dort ausgeführt. Der rechenintensive Teil wird über das Netzwerk in die Cloud ausgelagert, dort berechnet, und das Ergebnis an die Anwendung zurückgeschickt.

Hintergrundaufgaben

Anwendungen, die zur Bewältigung ihrer Aufgabe über einen längeren Zeitraum keine Benutzereingabe erfordern, sind eine weitere Klasse von Anwendungen. Typische Vertreter sind hierbei zB. Virenscanner oder das durchsuchen von Webseiten. Spezifisch für diese Klasse von Anwendungen ist, dass ihre Hauptfunktionalität nur sehr lose an die eigentliche Anwendung gekoppelt ist. Es ist somit möglich, dass zB. eine Anwendung, die jeden Morgen die neusten Nachrichten von unterschiedlichen Webseiten präsentiert, über die Nacht beendet wird. Der Anwendungsteil zum Durchsuchen der Webseiten wurde bereits zuvor ausgelagert und arbeitet die Nacht über eigenständig. Wird morgens die Anwendung wieder gestartet werden die gesammelten Nachrichten an die Anwendung übergeben.

Hardwareunterstützung

Obwohl bereits heute in den meisten Smartphones leistungsstarke Prozessoren verbaut werden, sind diese immer noch um ein vielfaches langsamer als leistungsstarke Desktop bzw. Server Prozessoren. Zudem können Anwendungen von Spezialprozessoren wie zB. schnellen Grafikkarten oder FPGAs profitieren.

2.1.3 Motivation und Hindernisse

Wie vielleicht bereits durch die Ausführungen ersichtlich wurde kann es unterschiedliche Motivationen geben, Code Offloading zu betreiben. Heutzutage steht vor allem der Leistungszuwachs und die Energieersparnis im Vordergrund. Dazu muß aber genau darauf geachtet werden, wie sich das Verhältnis zwischen Kommunikationsaufwand und Berechnungsaufwand entwickelt. Wie in Abbildung 2.2 zu sehen ist können

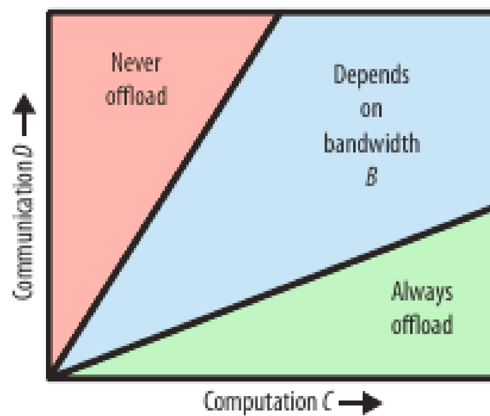

Abbildung 2.2: Offloading im Spannungsfeld zwischen Kommunikationsaufwand D und Berechnungsaufwand C [KL10]

drei Bereiche unterschieden werden. Kann durch das Offloading viel Rechenzeit und dadurch Energie eingespart werden ist auch ein hoher Kommunikationsaufwand vertretbar. Anders sieht es hingegen aus, wenn der Kommunikationsaufwand hoch und die Einsparung im Berechnungsaufwand gering ist. In diesem Fall sollte besser kein Offloading stattfinden. Zwischen diesen beiden Polen befindet sich ein großer Bereich in der die Entscheidung, ob sich Offloading lohnt, von der verfügbaren Bandbreite abhängt.

Ein weiterer Aspekt, Code Offloading durchzuführen, kann auch sein, die Zuverlässigkeit der Anwendung zu erhöht. Auch wenn die großen Cloudanbieter nicht gefeit vor Ausfällen sind [KL10] erreichen sie doch eine um vielfach höhere Verfügbarkeit als das durchschnittliche Smartphones. Dies macht sich besonders bei langlaufenden Prozessen bemerkbar. Ist das Offloading System darauf vorbereitet, kann während der Berechnung auf dem Offloading Server das Smartphone sogar ausgeschaltet werden. Wir es wieder eingeschaltet, meldet sich der Offloading Client beim Offloading Server und das Ergebnis der Berechnung wird an den Client übertragen.

2.1.4 Architektur

Die Architektur des Code Offloading Systems hängt stark von der Anwendung und dessen Entwickler ab. Es gibt Architekturen, bei denen schon bei der Entwicklung der Anwendung das Offloading System eingebunden wird. Auch eignen sich bestimmte

Architekturen besser für bestimmt Anwendungsklassen als andere. Offloading Systeme können durch die folgenden drei Architekturen unterschieden werden.

Feature Offloading

Bei diesen Offloading Systemen werden die Daten von der Anwendung aufbereitet und über eine definierte Schnittstelle an das Offloading System übergeben. Dieses verarbeitet die Daten und liefert ein Ergebnis an die Anwendung zurück. In diese Kategorie fallen die meisten Client-Server Architekturen, die heutzutage im Internet verwendet werden, wie REST[FT02] oder SOAP[W3C].

Bereits bei der Entwicklung der Anwendung muß entschieden werden, ob ein Offloading stattfindet und welche Daten ausgelagert werden sollen. Durch diese direkte Kontrolle lässt sich eine recht effizientes Offloading realisieren. Es erfordert jedoch vom Entwickler gute Kenntnisse in den verwendeten Technologien. Meist ist es auch schwierig bis unmöglich, Anwendungen nachträglich an solch eine Architektur anzupassen. Ein Offloading System das diese Architektur anbietet ist Cuckoo[KPKB12].

Methoden Offloading

In die Kategorie des Methoden Offloading fallen Architekturen, welche die Methode als auszulagernde Berechnungseinheit nutzen. Die Methode ist dabei eine in sich eigenständige Berechnungseinheit und kann auf dem Offloading Server direkt ausgeführt werden. Alle dazu benötigten Daten werden mit der Methode übermittelt. Die Methode muß dabei dem Offloading Server bekannt sein.

Dies kann auf mehreren Wegen geschehen. Die Methode kann dem Offloading Server bereits vor Ausführung bekannt sein, indem zB. das Programm dem Offloading Server bereits vorliegt. Es ist jedoch auch möglich dass die Methode mit der Offloading Anfrage and den Offloading Server übermittelt wird. Nachdem die Methode ausgeführt wurde übermittelt der Offloading Server den Rückgabewert der Methode an den Client. Dieser kann nun die Anwendung so weiter ausführen als habe er die Methode selbst ausgeführt.

Gemein ist allen Architekturen die Methoden Offloading anbieten, dass es eine Möglichkeit geben muß, Methoden für das Offloading zu markieren. Dadurch können die Methoden ausgewählt werden, die vom Offloading profitieren. Diese Markierung kann durch statische oder dynamische Codeanalyse oder durch direkte Annotationen im Quelltext durch den Entwickler erfolgen.

Ein Offloading System das diese Architektur anbietet ist MAUI [CBC+10].

Image Offloading

Image Offloading geht noch einen Schritt weiter als Methoden Offloading. Es wird nicht nur eine Methode auf dem Offloading Server ausgeführt, sondern das gesamte Programm oder es wird sogar ein komplettes Abbild auf dem Offloading Server erstellt. Durch dieses Abbild besteht nun die Möglichkeit, auch innerhalb eines Threads beliebig zwischen lokaler Ausführung und ausgelagerter Ausführung hin und her zu springen. Dazu wird der Thread auf dem Client schlafen gelegt und die Zustandsinformationen dieses Threads an den Server übertragen. Dort wird ein neuer Thread mit den übertragenen Zustandsinformationen gestartet und die Ausführung läuft auf dem Server weiter. Wann ein Offloading stattfindet liegt hier in der Hand des Offloading Systems. Dies muß versuchen, durch statische und dynamische Codeanalyse passende Migrationspunkte zu finden.

Ein Offloading System das diese Architektur anbietet ist Clone Cloud [CIM+11].

2.2 Caching

Ein Cache bezeichnet im Folgenden einen Speicher, der Daten für einen schnelleren Zugriff zwischenspeichert. Dazu befindet sich der Cache zwischen Datenlieferant und Datenkonsument mit dem Ziel, die Daten an den Kosumenten schneller auszuliefern als dies vom Datenlieferanten möglich wäre.

Solche Caches sind heutzutage überall in der Informationsverarbeitung anzutreffen. Moderne Prozessoren besitzen mehrere Caches die zwar um einiges kleiner sind als der Hauptspeicher, dafür können sie aber schneller von der CPU abgefragt werden. Festplattenzugriffe werden von Betriebssystemen im Hauptspeicher zwischengespeichert, um bei einer wiederholten Abfrage der selben Daten nicht auf den langsamen Massenspeicher warten zu müssen.

Im Internet sind Caches integraler Bestandteil der meisten Dienste. So wurde das Domain Name System (DNS) [Moc87] bereits mit integriertem Cache konzipiert. Auch in der Spezifikation des HTTP Protokolls [FGM+99] ist eine Caching Komponente spezifiziert.

Gemein ist diesen Caches, dass sie speziell für das jeweilige Protokoll konzipiert wurden. So ist ein HTTP Cache ungeeignet, eine DNS Eintrag zwischenzuspeichern und via versa.

Im Folgenden soll doch trotzdem auf ein paar Faktoren eingegangen werden, die allen Caching Systemen gemein sind.

2.2.1 Faktoren

Der Nutzen eines Caches hängt von einigen Faktoren ab. Smith [Smi87] hebt dabei die Folgenden hervor.

Lokalität

Ein wichtiges Prinzip für das Funktionieren eines Caches ist das der Lokalität. Es sagt zwei Dinge. Zunächst, dass es wahrscheinlich ist, dass Informationen, die in naher Zukunft benötigt werden, denen ähneln, die gerade benutzt werden. Dies ist die *zeitliche Lokalität*. In informationsverarbeitenden Systemen ist diese zeitliche Lokalität oft anzutreffen. So beinhalten Programme oft Schleifen, die immer wieder auf den selben Daten Berechnungen anstellen.

Zudem sagt das Prinzip, dass es wahrscheinlich ist, dass Informationen, die in naher Zukunft benötigt werden, den gerade benutzten räumlich nahe sind. Dies ist die *räumliche Lokalität*. Auch dies ist in Programmen oft wiederzufinden. Man stelle sich nur eine Schleife vor die über ein Zeichenkette iteriert.

Cachegröße

Ein weiterer wichtiger Faktor ist der der Cachegröße. Je größer der Cache, um so größer die Wahrscheinlichkeit dass sich die gesuchte Information darin befindet.

3 Verwandte Arbeiten

Gerade das Thema Code Offloading ist mit dem aufkommen der Smartphones in den letzten Jahren zu eine beliebten Forschungsgebiet geworden. Dabei entstanden einige Systeme mit zum Teil ganz unterschiedlicher Ausrichtung.

Eines dieser Systeme ist CloneCloud [CIM+11]. Die Grundidee bei CloneCloud ist, einen direkten Klon des Smartphones auf einer leistungsfähigen Maschine zu erstellen. Dieser Klon läuft dabei in einer virtuellen Maschine. Somit kann eine physikalische Maschine durchaus mehrere Klone beheimaten. Dank dieses Klone kann die Ausführung auf dem Smartphone jederzeit unterbrochen und auf dem Klone fortgesetzt werden. CloneCloud operiert dabei auf Thread Ebene. Dazu wird der zur Migration bestimmte Thread angehalten und alle Zustandsinformationen des Threads werden an den Klon übermittelt. Dieser startet einen Thread mit Hilfe der Zustandsinformationen und setzt die Ausführung fort.

Dieses Konzept sieht kein Caching vor. Prinzipiell läßt sich das in dieser Arbeit vorgestellte Caching auch für Clonecloud anpassen. Da das CloneCloud System jedoch dynamisch über die Migrationspunkte zwischen Smartphone und Klon entscheidet, können sehr viele Cacheeinträge entstehen, wodurch die Trefferwahrscheinlichkeit sinkt.

Ein ähnliches Modell wie CloneCloud ist Cloudlet [SBCD09]. Auch hier findet das Offloading mit Hilfe einer virtuellen Maschine statt. Auch dieser Ansatz sieht kein Caching vor und eignet sich auch nicht für den in dieser Arbeit vorgestellten Ansatz.

MAUI [CBC+10] ist ein Offloading System das dem in dieser Arbeit verwendeten ähnelt. Es setzt ebenfalls auf Methoden Offloading und versucht, mit Hilfe eines Profilers Methoden zu Identifizieren, die sich für ein Offloading eignen. Auch MAUI verwendet kein Caching. Durch die Nähe zu dem in dieser Arbeit verwendeten Offloading System sollte MAUI leicht um das hier verwendete Caching erweiterbar sein.

Ein weiteres Offloading System, das ein Framework für Anwendungsentwickler zur Verfügung stellt, ist Cuckoo [KPKB12]. Dieses Framework muß bereits bei der Entwicklung eingebunden werden. Auch Cuckoo sieht sieht kein Caching vor. Da im

Framework aber auch Methoden Offloading stattfindet, sollte es leicht um das hier vorgestellte Caching erweiterbar sein.

Offloading Systeme gehen oft von einer zuverlässigen Verbindung zwischen Offloading Client und Server aus. Dies ist aber gerade bei Mobilfunkverbindungen nur selten der Fall. Methoden wie auch mit unzuverlässigen Verbindungen ein zuverlässiges Offloading stattfinden kann wird bei Berg et al. [BDR14] beschrieben.

In [YCZ04] untersuchten Yuan et al. wie sich Caching und Offloading auf dynamische Webinhalte auswirkt. Untersucht wurde dies Anhand eines typischen Webshops. Dazu wurde der Webshop in Präsentationsschicht, Geschäftslogik und Datenbank unterteilt. Für jede Schicht wurde nun untersucht, wie sich das Offloading und Caching dieser Schicht auswirkt. Dabei konnte gezeigt werden, dass bei typischer Anfragen die Latenz um das 2 bis 3-fache gesenkt werden konnte. Zudem konnten über 70% der Anfragen aus dem Cache beantwortet werden, was zu einer signifikanten Lastreduzierung auf den Servern führte.

Eine weitern Ansatz für das Cachen von dynamischen Webinhalten wird in [CZB99] von Cao et al. vorgestellt. Dabei stehen Cache Applets im Mittelpunkt. Diese sind in einer plattformunabhänigen Sprache implementierte Programme, die über einen oder mehrere Universal Resource Locator (URL) identifiziert werden. Diese Applets können nun in Proxies zwischengespeichert werden. Gehen Benutzeranfragen über einen Proxy mit dem passendem Applet führt dieser, stellvertretend für den Server, das Applet aus. Wurde diese Anfrage für das Applet vorher schon vom Proxy ausgeführt kann der Proxy diese Anfragen aus dem Cache beantworten. Im Gegensatz zu dem in dieser Arbeit vorgestellten Ansatz müssen hier Anwendungen speziell als Cache Applets entwickelt werden.

Reddi et al. zeigen in [RCCS07], dass sich Code Caching von Methode besonders lohnt wenn Methoden anwendungsübergreifend betrachtetet werden, wie es auch in dieser Arbeit der Fall ist. Dazu wurden Standardanwendungen untersucht. Es zeigte sich dass besonders bei den verwendeten Bibliotheken Überschneidungen zwischen den Anwendungen entstehen.

Ein System, das im Gegensatz zu dem hier vorgestellten nicht auf leistungsfähige, stationäre Offloading Systeme setzt, ist Serendipidy [SLAZ12]. Hier besteht das System aus gleichberechtigten, mobilen Knoten.

Breslau et al. kommen in [BCF+99] zu dem Schluss dass Webanfragen der Zipf's Law folgen. Zipf's Law sagt dass die relative Wahrscheinlichkeit für eine Anfrage der $i$ten meistpopulären Seite $1/i$ beträgt. Es gibt zudem Hinweise dass diese Verteilung auch

für andere Netzwerk-Caches gilt. Dies läßt vermuten, dass diese Verteilung auch für das Caching beim Methoden Offloading gilt.

4 Konzept

In diesem Kapitel wird ein Konzept für das Caching in Offloading Systemen entwickelt. Da das Caching System auf einem bereits vorhandenen Offloading System aufbaut wird zunächst auf dieses eingegangen. Im Folgenden wird dieses Offloading System erweitert und eine Caching System konzipiert.

4.1 Ziele und Rahmenbedingungen

Das entwickelte Caching System soll im folgenden besonders zwei Ziele verfolgen. Es soll,wenn möglich, im Vergleich zum reinen Offloading die Ausführungszeit und den Netzwerkverkehr und damit den Stromverbrauch von Anwendungen reduziert werden.

Die folgende Konzeption für ein Caching System baut auf einem vorhandenen Offloading System auf. Dieses Offloading System arbeitet nach dem Prinzip des Methoden Offloading (vgl. Kapitel 2.1.4). Das Caching System muß also in der Lage sein, Methoden, deren Parameter und deren Berechnungsergebnis, also der Rückgabewert der Methode, möglichst eindeutig im Cache zu speichern. Außerdem muß es möglich sein über die Methode und passende Parameterwerte den dazu gespeicherten Rückgabewert abzurufen.

Bei den auszulagernden Anwendungen soll das Hauptaugenmerk auf den Anwendungen liegen, deren Primärfunktionalität (vgl. Kapitel 2.1.2) ausgelagert wird. Dies ist besonders dem schon vorhandenen Offloading System geschuldet, das bereits diese Anwendungsklasse auslagern kann. Anwendungen aus dieser Klasse bieten sich jedoch auch für das Offloading und Caching an. Gerade die rechenintensiven Teile einer Anwendung profitieren am meisten, wenn sie auf einem leistungsfähigeren Computer ausgeführt werden. Wie schon in den Grundlagen ausgeführt lohnt sich Offloading nur, wenn die Vorteile durch die ausgelagerte Berechnung nicht durch den höheren Kommunikationsaufwand zunichte gemacht werden. Diese Gefahr besteht im Besonderen, wenn nicht nur die rechenintensive Primärfunktionalität ausgelagert wird.

Nicht nur im Firmenumfeld ist es von höchster Bedeutung, dass sensitive Daten bestimmte geschützte Bereiche nicht verlassen. Das hier vorgestellte Konzept soll daher eine einfache Abgrenzung dieser Bereiche ermögliche.

Nicht Ziel dieses Konzeptes ist die Entscheidung, ob und welcher Teil des Programms ausgelagert und damit in den Cache gelegt werden soll. Es wird außerdem davon ausgegangen, daß eine Caching Anfrage nur stattfindet, wenn auch eine Entscheidung zum Offloading getroffen wurde.

Bei der Kommunikation wird nur die Verbindung zwischen Smartphone und dem ersten Netzknoten als Kostenfaktor betrachtet. Bei dieser Verbindung handelt es sich fast immer um eine Funkverbindung. Veränderungen bei der Kommunikation über dieses Verbindung hat direkte Auswirkung auf mögliche Zeit- und Energieersparnisse. Die Leitungen hinter dem ersten Netzknoten sind in der Regel kabelgebunden und gut ausgebaut so dass diese zu vernachlässigen sind.

Zudem wird von einer zuverlässige Kommunikation ohne Fehler ausgegangen. Fehler führen zu einer ganzen Klasse von weiteren Problemen, die den Rahmen dieser Arbeit sprengen würden. Genauere Ausführungen zu einem robusteren Offloading lassen sich bei Berg et al.[BDR14] nachlesen.

Das vorgestellte Caching soll, wenn möglich, ohne Anpassungen an der Anwendung funktionieren. Dies ist vor allem für schon vorhandene Anwendungen wichtig, da diese sich oft nur schwer nachträglich anpassen lassen. Anwendungsspezifische Optimierungen sind daher optional.

4.2 Offloading System

Das im folgenden vorgestellte Konzept baut auf einem bereits vorhanden Offloading System auf. Dabei handelt es sich um eine modifizierte Jikes Research Virtual Machine (jRVM) [The]. Dies ist eine in Java geschriebene Java Virutal Maschine. Die jRVM ist somit in der Lage, Java Anwendungen auszuführen. Diese wurde um Methoden Offloading (vgl. Kapitel 2.1.4) erweitert. Durch Parameter beim Start läßt sich diese modifizierte jRVM sowohl als Offloading Client als auch als Offloading Server betreiben. Dabei muß sich das Programm sowohl auf dem Server als auch auf dem Client befinden. Die auszulagernde Methode wird im Voraus bestimmt und wird ebenfalls als Parameter beim Start übergeben.

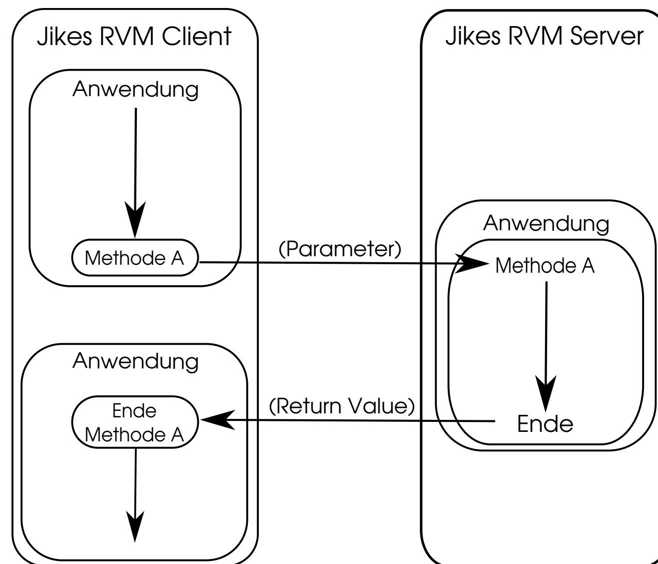

Abbildung 4.1: Methoden Offloading in der Jikes RVM

Bei der ausgewählten Methode sind einige Einschränkungen zu beachten. Es ist nicht möglich eine Methode auszulagern die auf clientspezifische Hardware zurück greift. So ist es zB. nicht möglich eine Methode auszulagern, die auf das GPS des Clients zugreift, denn man mußt davon ausgehen, dass diese Hardware auf dem Server nicht zur Verfügung steht. Selbst wenn der Server über ein GPS verfügen würde, wäre eine Ausführung für den Client nicht mehr transparent, da das GPS des Servers möglicherweise ganz andere Daten liefert als das GPS des Clients. Das selbe gilt natürlich auch für Daten die auf die jeweiligen Datenträger geschrieben werden.

Bei dem verwendeten Offloading Systems werden alle Eingabevariablen als Parameter an die Methode übergeben. Dies liegt daran, dass nur die Methodenparameter vom Client an den Server übermittelt werden. Alle anderen Variablen wie zB. Instanzvariablen oder Klassenvariablen können zwischen Client und Server divergieren und müssen daher auch als Parameter übergeben werden.

Wie bereits erwähnt wird davon ausgegangen, dass eine Caching Anfrage nur erfolgt, wenn auch ein Offloading stattfand. Somit werden nur solche Methoden in den Cache gelegt, die sich vorher als lohnend für das Offloading herausgestellt haben.

4.2.1 Offloading Server

Nach dem Start des Servers wartet dieser auf einem konfigurierbaren TCP/IP Port auf eingehende Anfragen. Sobald sich ein Client mit dem Server verbindet übermittelt dieser den Programmname, den Methodennamen und den Namen der Klasse in dem sich diese Methode befindet. Damit kann der Server eindeutig die Methode bestimmen die der Client auslagern möchte. Erreicht der Client die zum Offloading bestimmte Methode übermittelt er die Methodenparameter wie in Abbildung 4.1 zu sehen. Der Server springt nun zu der vorher übermittelten Methode, passt den Heap so an als wäre die Methode mit den übermittelten Parametern aufgerufen worden, und führt die Methode aus. Anschließend wird der Rückgabewert an den Client zurückgegeben.

4.2.2 Offloading Client

Der Offloading Client übermittelt nach den Start den Programmname, den Methodenname und den Namen der Klasse an den Offloading Server. Dann führt er das definierte Programm aus. Erreicht er dabei die auszulagernde Methode werden die Parameterwerte an den Server übermittelt. Dieser liefert den Rückgabewert der Methode zurück. Dieser wird geparst, an die richtige Stelle auf dem Heap geschrieben und das Programm weiter ausgeführt.

4.3 Architektur

Um den unterschiedlichen Ressourcen der involvierten Systeme Rechnung zu tragen wird eine mehrstufige Caching Hierarchie vorgeschlagen. Eine solche Architektur ist im Internet weit verbreitet und wurde bereits von Chankhunthod et al. [CDN+95] 1995 propagiert. Bei dieser Hierarchie wird davon ausgegangen, dass je weiter entfernt sich der Cache von eigentlich Smartphone befindet, desto mehr Ressourcen in Form von Speicher und Rechenleistung stehen zur Verfügung. Die Caching Komponente wird wird daher als eigenständiger Server konzipiert.

Tier 1: Lokaler Cache Der lokale Cache läuft als eigenständiges Programm auf dem Smartphone. Es wird davon ausgegangen, dass auf dem Smartphone nur eine Instanz dieses Caches läuft und sich die lokalen Offloading Systeme mit diesem Cache über eine Netzwerkverbindung verbinden. Dieser Cache steht somit für alles Offloading

System auf diesem Gerät zur Verfügung. Da sowohl der Arbeitsspeicher als auch der Massenspeicher eines Smartphones sehr limitiert ist kann der Speicher des Caches nicht allzu groß werden. Auch die Lebenszeit des Caching Prozesses ist begrenzt. Das Smartphones kann ausgeschaltet werden oder das Betriebssystem kann den Caching Prozess beenden wenn zB. der Arbeitsspeicher knapp wird.

Tier 2: LAN Cache Der LAN Cache befindet sich im selben lokalen Netzwerk wie der anfragende Offloading Client. Er läuft als eigenständiger Prozess auf einem oder mehreren Computern. Durch die nähe zum Offloading Client stellt dieser der erste Cache mit ausreichend Ressourcen dar.

Tier 3: Cloud Cache Dieser Cache ist nicht genauer spezifiziert außer dass er über das unten beschriebene Protokoll ansprechbar ist. Hier können zB. Dienstleister Offloading Caches anbieten.

Von Tier 2 und Tier 3 ist der Offloading Server über eine Kabelverbindung zu erreichen.

4.3.1 Sicherheit

Ein solch gestaffeltes Caching ermöglicht neben dem Ausnutzen vorhandener Ressourcen auch die Durchsetzung von Richtlinien. Für Unternehmen ist es essentiell, dass bestimmte unternehmenskritische Daten nicht auf fremden Rechnern gespeichert werden. Die Schnittstellen zwischen den unterschiedlichen Hierarchieebenen bieten dabei die Möglichkeit, die ausgetauschten Daten zu kontrollieren. Besonders interessant ist dabei der Übergang von Tier 2 zu Tier 3. Tier 2 ist in der Regel Teil des Unternehmensnetzwerkes und somit unter eigener Kontrolle wohingegen Tier 3 nur sehr schwer zu kontrollieren ist.

4.3.2 Inter Cache Kommunikation

Jedes Tier kann mit beliebig vielen Caches ausgestattet sein. Ein neuer Eintrag in den Cache erfolgt immer über eine INSERT Nachricht (siehe 4.6.2) an einen Knoten im darüberliegenden Tier. Würden INSERT Nachrichten auch nach unten propagiert werden würden diese bis zum Smartphone gelangen und damit hohe Kosten für Kommunikation und Speicherung erzeugen.

Auf Tier 1 laufende lokale Caches propagieren keine INSERT Nachrichten an darüber liegende Schichten. Da die Kommunikation zwischen Tier 1 und Tier 2 über eine teure Funkstrecke erfolgt wird Tier 2 direkt vom Offloading System kontaktiert.

Innerhalb eines Tiers werden neue Einträge an alle Knoten weitergegeben. Optimalerweise erfolgt dies über eine gemeinsame Multicast Gruppe.

Suchanfrage werden ebenfalls von den Caches weitergeleitet. Da alle Caches des selben Tier die gleichen Cacheeinträge besitzen werden Suchanfragen immer nur an das darüberliegende Tier weitergeleitet.

4.3.3 Skalierbarkeit

Das hier vorgestellte Konzept bietet mehrere Ansätze, um auch eine große Anzahl an Cachinganfragen zu verarbeiten.

Zunächst ist durch die Inter Cache Kommunikation ein möglichst hoher Replizierungsgrad sichergestellt. Es muß somit nur sichergestellt werden, dass die Anfragen an das Caching System möglichst gleich verteilt über die Caching Server stattfinden. Dies kann einfach über Round Robin DNS [Bri95] erreicht werden. Dabei sind alles IP Adressen der Caching Server unter einem Domain Namen erreichbar. Der DNS Server liefert dann bei einer Namensanfrage alle Einträge in jeweils zufälliger Reihenfolge zurück. Der Resolver des Betriebssystems liefert in der Regel den ersten Eintrag an Anwendungen zurück wodurch eine zufällige Namensauflösung erreicht wird.

Eine Partitionierung des Caching Systems kann durch die Spezialisierung von Caching Servern erreicht werden. Durch diese Spezialisierung ist es möglich, ausgewählte Systeme bestimmten Anwendungsklassen zuzuordnen. Damit können zB. wichtige Anwendungen auf besonders leistungsfähigen Systemen gespeichert werden.

4.4 Anwendungsspezifische Optimierungen

Um die Trefferwahrscheinlichkeit zu erhöhen werden im Folgenden zwei anwendungsspezifische Optimierungen vorgestellt.

4.4.1 Anwendungsklassen

Für jeden Caching Server kann eine Anwendungsklasse definiert werden. Bei der Anwendungsklasse handelt es sich um eine frei wählbare Zeichenkette. Diese Zeichenkette kann von Offloading Clients mit Hilfe der INFO Nachricht (vgl. Kapitel 4.6.2) abgefragt werden. Es ist somit möglich, dass sich Offloading Clients die Caching Server aussuchen, die für ihre Anwendungsklasse spezialisiert sind. So könnte zB. ein Caching Server auf Schachprogramme spezialisiert sein indem dessen Cache mit einer großen Schachdatenbank gefüllt wurde. Ein Offloading Client kann nun diesen Cache ermitteln und so seine Trefferwahrscheinlichkeit erhöhen. Es ist dabei Sache von Offloading Client und Caching Server sich auf eine gemeinsame Taxonomie zu einigen.

4.4.2 Wertbereiche für Parameter

Über die Nachrichtenoptionen `parameter_min` bzw. `parameter_max` des Nachrichtentyp SEARCH (vgl. 4.6.2) können Rückgabewerte für Methoden gesucht werden deren Parameter größer bzw. kleiner eines definierten Wertes sind.

Es gibt eine Reihe von Algorithmen die, je mehr Zeit ihnen zur Verfügung steht, desto bessere Ergebnisse liefern. Klassische Beispiele sind hier Strategiespiele wie Dame, Mühle oder Schach. Je mehr Spielzüge der Spielealgorithmus evaluieren kann um so besser ist auch der nächste Zug. Durch die Bestimmung von Minimal- bzw. Maximalwerten für bestimmte Methodenparameter ist es möglich, Zwischenergebnisse aus dem Cache zu beziehen.

Angenommen ein Schachalgorithmus wird für eine bestimmte Stellung so aufgerufen dass er bis zur Tiefe 6 den Entescheidungsbaum durchsuchen soll. Bevor der Algorithmus seine Suche aufnimmt kann nun der Cache nach Zwischenergebnissen befragt werden indem nach Methodenaufrufen mit dieser Ausgangstellung und einer Maximalsuchtiefe von 6 gesucht wird. Befindet sich im Cache zB. ein Ergebnis für die Suchtiefe 4, so muß der Algorithmus nun nur noch die Schritte 4 bis 6 berechnen.

Um die beschriebene Optimierung zu nutzen ist es nötig, die modifizierten Methodenaufrufe zu annotieren. Dies geschieht am besten direkt durch den Anwendungsentwickler. Eine Möglichkeit wie dies unter Java geschehen kann ist in Abschnitt 5.1.4 auf Seite 44 beschrieben.

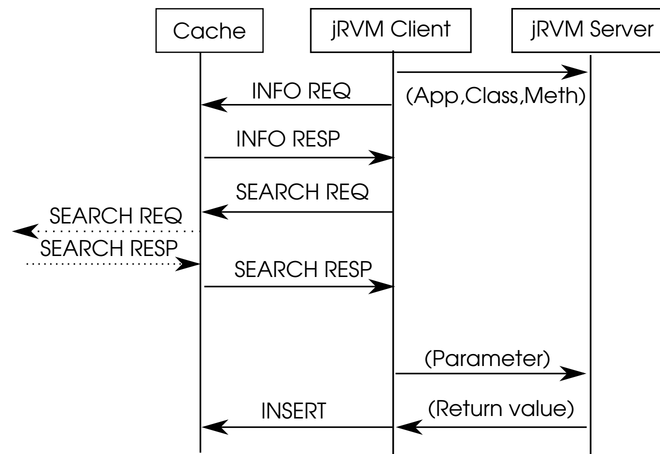

Abbildung 4.2: Netzwerkkommunikation zwischen Cache und Offloading Client und Server

4.5 Kodierung

Für einen Austausch zwischen Offloading System und Cache ist es wichtig dass Methoden, Parameter und der Rückgabewert eindeutig kodiert sind. Hinzu kommt dass besonders für die Übertragung über das Mobilfunknetz die Kodierung möglichst kompakt sein sollte.

4.5.1 Methoden

Methoden werden durch einen 256-bit Wert identifiziert. Dieser wird als Zeichenfolge in Hexadezimaldarstellung übertragen und ist somit 64 Zeichen lang. Dies bedeutet, das egal wie groß die zu identifizierende Methode ist, über das Netzwerk immer nur 64 Zeichen übertragen werden müssen. Somit ist der Netzwerkverkehr begrenzt. Wie die Abbildung der Methode auf einen 256-bit Wert stattfindet bleibt dabei der Implementierung überlassen. Die Implementierung sollte jedoch sicherstellen dass die Abbildung möglichst eindeutig ist. Zwei unterschiedliche Methoden mit dem selben Bezeichner führt zu Fehleinträgen im Cache und somit zu fehlerhaften Programmen.

Eine Möglichkeit diesen 256-bit Wert für eine Methode unter Java zu generieren ist in Abschnitt 5.1.3 auf Seite 42 beschrieben. Dort wird der Bytecode der Java Methode verwendet um daraus mithilfe von SHA-256 ein 256 bit Wert zu generiert.

4.5.2 Parameter- und Rückgabewerte

Methodenparameter können auf zwei unterschiedliche Arten kodiert werden wohingegen für den Rückgabewert nur die Standardkodierung zulässig ist. Die Standardkodierung enthält sowohl den Datentyp als auch den Wert des Parameters. Diese Kodierung muß auch verwendet werden wenn der Parameter für Bereichsanfragen verwendet werden soll. Die alternative Kodierung besteht aus einem Hashwert.

Standardkodierung

Parameter- und Rückgabewerte werden werden als 4-Tupel `(intern,tStr,rStr,vStr)` übertragen. Dabei wird das selbe Format verwendet das auch vom Offloading System verwendet wird.

- *`intern`*
  Dieser Parameter wird intern vom Offloading System verwendet und vom Caching System nicht weiter beachtet
- *`tStr`*
  Ein Zeichen das den Typ von *`vStr`* angibt
- *`rStr`*
  Ein Zeichen das die Transportkodierung den Typ *`vStr`* angibt
- *`vStr`*
  Der eigentliche Wert in der definierten Kodierung

Die Zeichen zur Datentypbestimmung werden dabei folgendermaßen definiert:

I → Integer	S → Short
F → Float	L → Klasse
J → Long	V → Void
Z → Boolean	D → Double
C → Char	B → Byte
[→ Array	

Mehrere 4-Tupel können mit # konkateniert werden.

Hash-Kodierung

Bei der Hash-Kodierung handelt es sich um die Standardkodierung, die mit Hilfe von SHA-256 [ErH06] zu einem 256-bit Wert umgewandelt wird. Dieser wird als Zeichenkette in Hexadezimaldarstellung dargestellt.

Diese Darstellung hat sowohl Vor- als auch Nachteile. Wird eine Parameter auf diese Weise kodiert und an den Caching Server gesendet ist es nicht mehr möglich damit Bereichsanfragen durchzuführen. Durch das Hashen verliert der Parameter seine lexikalische Ordnung, was ein Vergleich mit gespeicherten Werten unmöglich macht.

Diese Kodierung hat aber auch Vorteile. So ist die Länge des Hashwertes auf 64 Zeichen begrenzt. Damit können selbst große Werte eindeutig identifiziert und verglichen werden.

Wenn ein Parameter in Standardkodierung länger als 64 Zeichen ist muß dieser in Hash-Kodierung an das Caching System übermittelt werden. Damit wird die übertragene Datenmenge begrenzt und Bereichsanfragen für alle kürzeren Parameter bleibt trotzdem möglich.

4.6 Kommunikationsprotokoll

Die Kommunikation zwischen Offloading System und Cache erfolgt über eine an RFC 2616 [FGM+99] angelehntes Protokoll. Die einzelnen Kommunikationsschritte sind in Abbildung 4.2 skizziert.

Bevor das Offloading System Kontakt mit einem Cache aufnehmen kann muß ein oder mehrere Caches dem Offloading System bekannt sein. Wie dieses sogenannte Bootstrapping vonstatten gehen soll ist der Implementierung überlassen da die Möglichkeiten stark von der vorhandenen Infrastruktur beeinflußt werden. Eine Möglichkeit wäre zB. mDNS [CK13], auch bekannt als Bonjour.

Wenn Offloading Client und der Caching Prozess zum ersten mal Kontakt aufnehmen, hat der Offloading Client die Möglichkeit, mit einer INFO Nachricht Rahmenbedingungen über den Cache abzufragen. Damit hat er die Möglichkeit, mehrere Caches zu Kontaktieren und den passendsten auszuwählen. Dazu übermittelt der Cache seinen Füllstand und die Anwendungsklasse auf die dieser Cache spezialisiert ist. Mißt der

Offloading Client noch die Reaktionszeit so stehen drei Parameter zur Verfügung. Welche Priorität dabei den einzelnen Parametern zugeordnet werden ist eine Entscheidung der Implementierung.

Trifft der Offloading System auf eine auszulagernde Methode wird vor dem Offloading eine Anfrage an den gewählten Cache geschickt. Die Anfrage erfolgt Mithilfe einer SEARCH Nachricht. Diese Nachricht übermittelt einen Methodenbezeichner und die gewünschten Paramterwerte in der oben beschriebenen Kodierung. Kann der Cache selbst kein passenden Eintrag finden, leitet er die Anfrage wie in Abschnitt 4.3.2 auf Seite 29 beschrieben weiter. Das Ergebnis der Suche wird an das Offloading System weitergeleitet. Wurde im Cache ein Eintrag gefunden kann der Offloading Client diesen Rückgabewert an das Programm übergeben und die Methode muß nicht ausgeführt bzw. ausgelagert werden. Wurde kein Eintrag gefunden findet das oben Offloading an den Offloading Server statt. Der Offloading Server überträgt sein Ergebnis an den Offloading Client und an einen Caching Server in Tier 2 oder Tier 3.

4.6.1 Nachrichtenformat

Eine Anfrage ist folgendermaßen aufgebaut:

```
Nachricht = Nachrichtentyp
            *(Nachrichtenoption CRLF)
          CRLF}
```

Je nach *Nachrichtentyp* sind unterschiedliche *Nachrichtenoptionen* zulässig. Die Antwort ist abhängig von der jeweiligen Anfrage.

4.6.2 Nachrichtentypen

INFO

Über eine INFO Nachricht kann der Client Informationen über den Caching Server anfragen. Dazu sind folgende Nachrichtenoptionen zulässig.

- `get-cache-type`
  Über die `get-cache-type` Nachrichtenoption kann ein Client erfahren, ob dieser Cache auf Einträge einer bestimmten Anwendungsklasse spezialisiert ist. Der Server antwortet auf diese Option mit `cache-type = type`. *`type`* ist eine Zeichenkette welche die Anwendungsklasse beschreibt. Es ist dabei Sache der teilnehmenden Systeme sich auf eine gemeinsame Taxonomie zu einigen.

- `get-cache-size`
  Der Füllstand des Caches in Form von Cacheeinträgen läßt sich über `get-cache-size` erfahren.

SEARCH

Mithilfe einer SEARCH Nachricht kann ein Cache nach einem (Methode,Parameter) Paar durchsucht werden. Bei einem Treffer liefert der Cache den passenden Rückgabewert zurück. Die `parameter_*` Optionen sind in Reihenfolge der Parameter von links nach rechts einzufügen.

- `method_hash =` *`Method`*
  `Method` ist ein 64 Zeichen lange Zeichenkette die die bezeichnete Methode identifiziert.

- `parameter_value =` *`parameter`*
  `parameter_value` sucht eine Methode mit genau diesem Parameterwert.

- `parameter_min =` *`parameter`*
  Ermöglicht der Datentyp eine Ordnungsrelation kann mit `parameter_min` eine Methode mit einem minimalen Parameterwert gesucht werden.

- `parameter_max =` *`parameters`*
  Ermöglicht der Datentyp eine Ordnungsrelation kann mit `parameter_max` eine Methode mit einem maximalen Parameterwert gesucht werden.

Bei einem Treffer wird der *`return_value`* in der Form `(`*`parameter`*`)` zurückgegeben. Sollten durch eine Bereichsanfrage mehrere Einträge auf die Anfrage passen wird vom Caching Server eine beliebige passende Antwort zurückgeliefert. Konnte keine passende Methode gefunden werden wird `CC_NOTFOUND` zurückgegeben.

INSERT

Eine INSERT Nachrichten fügt einen neuen Eintrag zum Cache hinzu.

- `method_hash = ` *`Method`*
  `Method` ist ein 64 Zeichen lange Zeichenkette die die bezeichnete Methode identifiziert.
- `method_type = ` *`type`*
  `type` ist eine Zeichenkette welche die Anwendungsklasse beschreibt. Es ist dabei Sache der teilnehmenden Systeme sich auf eine gemeinsame Taxonomie zu einigen.
- `parameter_value = ` *`parameter`*
  Wert eines Parameters.
- `return_value = ` *`parameter`*
  Rückgabewert der Methode

5 Implementierung

Im folgenden Kapitel wird die Implementierung des Caching Systems beschrieben. Dazu wurde ein vorhandenes Offloading System so modifiziert dass es mit Hilfe des in Kapitel 4.6 beschriebene Protokolls Kontakt zu einem Caching Server aufnimmt. Die Implementierung dieses Caching Servers wird im Folgenden ebenfalls erläutert.

5.1 Offloading System

Das in dieser Arbeit benutzte Offloading System ist eine modifizierte Jikes RVM (Research Virtual Machine) [The]. Bei der Jikes RVM (jRVM) handelt es sich um eine in Java geschriebene virtuelle Maschine, die in der Lage ist, selbst Java Programme auszuführen. In der Standardkonfiguration benutzt die jRVM GNU Classpath [Fre] in der Version 0.97.2p13 und unterstützt somit das Java SDK bis Version 5. Die jRVM wurde bereits vor dieser Arbeit so erweitert dass sie sowohl als Offloading Client als auch als Offloading Server dienen kann.

5.1.1 Erweiterungen

Um die jRVM mit dem Caching Programm zu verbinden war es nötig, die jRVM an mehreren Stellen zu erweitern. Dazu wurden die folgenden Klassen hinzugefügt.

org.jikes.offloading.OffloadingCache

In dieser Klasse wird mit Hilfe der connect() Methode TCP/IP Verbindungen zu den Caching Servern aufgebaut. Außerdem stellt sie folgende Methoden für die unterschiedlichen Nachrichtentypen aus Kapitel 4.6.2 bereit:

- `void info()`
  Diese Methode startet pro verbundenem Caching Server einen Thread und ruft dann `org.jikes.offloading.OffloadingCacheInfoThread()` auf.
- `add(String` *`methHash`*`, String` *`para`*`, String` *`retValue`*`)`
  Diese Methode fügt neue Einträge zu den verbunden Caching Servern hinzu. Dazu startet diese Methode eine Thread pro verbundenem Caching Server und ruft dann die Methode `org.jikes.offloading.OffloadingCacheAddThread()` auf die sich um die Kommunikation kümmert. Der String *`methHash`* besteht aus dem Rückgabewert von `hashMethod()` in Hexadezimaldarstellung. `para` und `retValue` sind kodiert wie in Kapitel 4.5 beschrieben. `para` enthält die Parameter und `retValue` den Rückgabewert dieser Methode mit diesen Parametern.
- `String find(String methHash, String para)`
  Sucht in den verbundenen Caches nach einem Rückgabewert zu der Methode `methHash` und den Parametern `para`. Kodiert sind diese wie in Kapitel 4.5 beschrieben. Zur Kommunikation mit den Caches wird pro Cache ein Thread gestartet und die Methode `OffloadingCacheFindThread()` aus der Klasse `org.-jikes.offloading.OffloadingFindThread` aufgerufen.

org.jikes.offloading.OffloadingCacheAddThread

Da diese Klasse von der Klasse `Thread` erbt, wird deren `run()` Methode als eigenständiger Thread aufgerufen. Dieser Thread übernimmt die Kommunikation mit einem Cache und übergibt ihm einen neuen Eintrag mit Hilfe einer `INSERT` Nachricht wie in Kapitel 4.6.2 beschrieben.

org.jikes.offloading.OffloadingCacheFindThread

Diese Klasse erbt ebenfalls von `Thread` und `run()` wird als eigenständiger Thread aufgerufen. Dieser Thread sucht in einem Cache mit Hilfe der SEARCH Nachricht aus Kapitel 4.6.2 nach Methoden mit passenden Parametern. Wird eine solche gefunden wird der Rückgabewert zurückgeliefert.

org.jikes.offloading.OffloadingCacheInfoThread

Diese Klasse implementiert den Nachrichtentyp INFO aus Kapitel 4.6.2. Sie erbt ebenfalls von `Thread` und kann somit in einem eigenen Thread ausgeführt werden. Sie

liefert, wenn vorhanden, die Nachrichtenoptionen `cache-size` und `cache-type` zurück.

5.1.2 Ablauf

Nach dem Starten der jRVM führt diese mehrere Initialisierungsschritte durch bevor letztendlich das eigentliche Programm gestartet wird. Der letzte dieser Initialisierungsschritte erfolgt in `org.jikes.vm.finishBooting()`. Hier erfolgt neben dem Initialisieren des Offloading auch das Initialisieren des Caching.

Dazu werden mit der oben beschriebenen `connect()` Methode aus `org.jikes.offloading.OffloadingCache` TCP/IP Verbindungen zu den per Parameter übergebenen Caches aufgebaut. Die einzelnen Caches werden daraufhin parallel, mit Hilfe der `info()` Methode aus `org.jikes.offloading.OffloadingCache`, über ihre Eigenschaften befragt. Der Algorithmus zur Bestimmung des besten Caches ist im Listing 5.1 zu finden.

Der beste Cache wird dabei durch drei Kriterien bestimmt. Zuallererst der `cache-type`. Caches mit dem selben `cache-type` werden vor allen anderen bevorzugt da dort die Wahrscheinlichkeit auf einen Treffer im Cache am höchsten ist. Caches mit dem selben `cache-type` werden nach Cachegröße prioritisiert. Sollten hier zwei Caches mit der selben Cachegröße existieren wird derjenige mit der niedrigeren Round Trip Time (RTT) ausgewählt, also der Cache, der schneller geantwortet hat.

Die eigentlichen Anfragen an den Cache erfolgen vor dem Offloading in der Klasse `org.jikesrvm.offloading.CCBaselineOffloader`. Dort wird für jede zum Offloading markierte Anwendungsmethode die Methode `startOffloading()` aufgerufen. Zunächst wird überprüft ob für diese Methode Annotations vorhanden sind. Wenn ja werden diese ausgewertet. Wie dies geschieht wird weiter unten in Abschnitt 5.1.4 auf Seite 44 beschrieben. Diese Annotations dienen dazu Bereichsparameter, wie im Nachrichtentyp SEARCH (siehe Abschnitt 4.6.2 auf Seite 36) definiert, zu implementieren.

Als nächstes wird der Methode ein möglichst eindeutiger 32 Byte Wert zugeordnet. Zusammen mit den Parametern werden diese Werte mit der Methode `find()` aus der Klasse `org.jikes.offloading.OffloadingCache` an den oben ermittelten Cache gesendet. Wie die Werte kodiert sind wird in Abschnitt 5.1.3 auf der nächsten Seite genauer erläutert. Sollte ein Rückgabewert für dieses Tupel aus Methode und Parametern im Cache gespeichert sein wird dieser von `find()` zurückgegeben. Ansonsten besteht der Rückgabewert aus der Zeichenkette `CC_NOTFOUND`. In diesem Fall findet das Offloading statt. Der im Offloading ermittelte Rückgabewert wird dann über

Listing 5.1 Algorithmus zur Bestimmung des besten Caches

```
Cache bestCache(Cache[] caches, String cache-type)
{
 Cache bestCache = null;
 for (Cache c : caches) {
  if (c.cache-type.equals(cache-type)) {
   if (!bestCache || c.cache-size > bestCache.cache-size) {
    bestCache = c;
    continue;
   }
   else if (c.cache-size == bestCache.cache-size) {
    if (c.rtt < bestCache.rtt) {
     bestCache = c;
     continue;
    }
   }
  }
 }
}
```

die `add()` Methode aus der Klasse `org.jikes.offloading.OffloadingCache` zum Cache hinzugefügt.

5.1.3 Methodenkodierung

Wie schon in Abschnitt 4.5 auf Seite 32 erläutert ist es wichtig, Methoden möglichst eindeutig zu identifizieren. Unter Java bieten sich dafür unterschiedliche Möglichkeiten.

Namensraum und Klassenhierarchie

In Java werden Klassen üblicherweise in Namensräumen organisiert. Dies und der Umstand, dass es sich bei Java um eine objektorientierte Programmiersprache handelt, ermöglicht eine hierarchische Zuordnung von Methoden. Eine Methode ist Teil einer Klasse und diese wiederum ist Teil eines Namensraums. Diese Zuordnung wird von dem hier verwendeten Offloading System verwendet. Für diesen Fall ist eine solche Zuordnung auch eindeutig da sowohl dem Offloading Client als auch dem Offloading Server das auszuführende Programm bekannt ist. Die Methode muß also nur in dem einen auszuführenden Programm eindeutig herleitbar sein. Da sich im Cache jedoch

üblicherweise Einträge von ganz unterschiedlichen Programmen befinden können bietet diese Klassifizierung keine hinreichende Eindeutigkeit. Denn es ist nicht ausgeschlossen dass zwei Programme unter dem selben Namensraum und mit der selben Klassenbezeichnung jeweils unterschiedliche Methoden definieren.

Quelltext

Wenn Bezeichner sich als zu ungenau für die möglichst eindeutige Identifizierung einer Methode herausgestellt haben so bleibt nur noch die Methode selbst. Diese kann in Java in mehreren Formen vorliegen. Zunächst als Quelltext. Dieser muß jedoch nicht vorhanden sein und auch sonst eignet sich dieser nur bedingt. Die selbe Funktionalität kann im Quelltext durch unterschiedliche Semantik erreicht werden. Eine eindeutige Abbildung von Funktionen gestaltet sich dadurch recht aufwändig.

Maschinencode

Java Programme werde in zwei Schritte compiliert. Zunächst wird aus dem Quelltext ein Bytecode generiert. Dieser ist maschinenunabhängig und wird zur Laufzeit in Maschinencode umgewandelt. Moderne Just-In-Time (JIT) Compiler sind in der Lage, zur Laufzeit Optimierungen am Maschinencode vorzunehmen. Der Maschinencode ist somit nicht nur abhängig vom verwendeten Compiler sonder auch von Rahmenbedingungen während der Laufzeit. Damit eignet sich der Maschinencode nur sehr eingeschränkt als eindeutiger Bezeichner.

Bytecode

Bleibt zuletzt noch der Bytecode. Dieser wird von der Java Virtual Maschine gelesen, interpretiert und in Maschinencode umgesetzt. Der Bytecode selbst wird von einem Java Compiler aus dem Quelltext generiert. Somit ist der Bytecode für eine Methode abhängig vom verwendeten Compiler bzw. von dessen Version. Der Markt der Java Compiler wird dominiert vom Hauptenwickler Oracle. Es ist somit davon auszugehen dass eine Überdeckung von Methoden mit selber Funktionalität und Methoden mit dem selben Bytecode recht hoch ist.

Nach Abwägung der vorgestellten Möglichkeiten wurde in dieser Implementation der Bytecode als Methodenidentifikation gewählt. Dieser kann jedoch theoretisch beliebig lange sein. Um den Bytecode auf die in der Konzeption definierten 32 Byte

Listing 5.2 Parameter Annotations für Bereichsanfragen

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)

@Target({ElementType.PARAMETER})
public @interface Min { }
public @interface Max { }
```

abzubilden wurde in der Implementierung aus dem Bytecode ein SHA-256 [ErH06] Wert generiert.

5.1.4 Parameter Annotations

Java bietet die Möglichkeit, den Quelltext zu annotieren, also Zusatzinformationen einzufügen, die nicht zum eigentlichen Programmcode gehören. Diese Annotationen können zur Übersetzungszeit oder zur Laufzeit ausgewertet werden. Im Folgenden werden Annotationen an Methodenparameter benutzt um Bereichsparameter wie in Abschnitt 4.4.2 auf Seite 31 beschrieben umzusetzen.

Dazu müssen die in Listing 5.2 aufgeführten Markierungsannotationen im Quelltext definiert werden. Der Anwendungsentwickler kann nun mit den `@Min` bzw. `@Max` Annotationen Methodenparameter annotieren. Als Beispiel soll folgende Methode dienen:

```
int t(int p1)
```

Möchte nun der Anwendungsentwickler, dass diese Methode auch für x mit $x <= p1$ zurückkehrt, dann kann er sie folgendermaßen annotieren:

```
int t(@Min int p1)
```

Analoges gilt für die `@Max` Annotation.

5.2 Caching Server

Die Implementierung des Caching Servers besteht aus drei Komponenten. Zum ersten aus einer Netzwerkkomponente die sich um die Annahme und den Aufbau von

Verbindungen kümmert. Zum zweiten eine Komponente die sich um das Parsen der Nachrichtentypen aus Kapitel 4.6.2 kümmert und zuletzt die Datenstrukturen, die die gespeicherten Daten verwalten.

5.2.1 Netzwerk

Die Kommunikation mit dem Offloading System und anderen Caching Systemen erfolgt über TCP/IP. Damit der Caching Server in der Lage ist, mehrere Verbindungen parallel zu bedienen, wurde ein Thread Pool gewählt. Dazu werden zu Beginn eine feste Anzahl an Threads gestartet. Diese lauschen auf dem selben TCP/IP Port und nehmen dort Verbindungen entgegen. Wird ein Thread beendet, wird für diesen wieder ein neuer Thread gestartet, so dass sich immer gleich viele Threads im Pool befinden.

Dieses Konzept hat mehrere Vorteile. Eine Thread zu starten kostet Rechenzeit. Durch das Vorhalten von schon gestarteten Threads fällt diese Arbeit nur beim Programmstart an. Programme die im Gegensatz für jede Verbindung einen neuen Thread erstellen können bei vielen Verbindungsanfragen sehr viele Threads starten und somit viele Ressourcen binden. Dies kann bei einem Thread Pool nicht geschehen, da die Anzahl der Threads begrenzt ist.

5.2.2 Parser

Das Parsen der unterschiedlichen Nachrichtentypen aus Kapitel 4.6.2 erfolgt in der Klasse `OffloadingCache`. Dort kümmert sich zunächst die Methode `parseMsg()` um das parsen der Nachricht. Je nach Nachrichtentyp wird die passende Methode zur Weiterverarbeitung aufgerufen. `add()` fügt einen neuen Eintrag in die weiter unten beschriebene Datenstruktur ein. `getRetValue()` durchsucht diese Datenstruktur nach übereinstimmenden Einträgen und liefert, falls ein Eintrag gefunden wurde, den Rückgabewert zurück.

5.2.3 Datenstruktur

Bei der in Abb. 5.1 auf Seite 47 vorgestellten Datenstruktur handelt es sich um eine reine In-Memory Datenstruktur. Dies ermöglicht schnelle Lese- und Schreibvorgänge und hält die Referenzimplementierung übersichtlich. Einer Persistenzschicht läßt sich, wenn gewünscht, leicht einfügen.

Aufbau

Die Methodenbezeichner in Form von 32 Byte Werten befinden sich in einer Hashmap, die diese auf einen `OffloadingCacheEntry` abbildet. Wie viele Sprachen ermöglicht auch Java, Methoden zu überladen. Es muß daher nicht nur möglich sein, unterschiedliche Parameterwerte zu speichern, sondern auch unterschiedliche Parameter. `OffloadingCacheEntry` verwaltet daher eine Liste mit `ParaList` Einträgen. Jede dieser `ParaList` Einträge repräsentiert eine Kombination von bekannten Parameterwerten und den dazugehörigen Rückgabewerte.

Eintrag hinzufügen

Neue Einträge bestehen aus einem Methodenbezeichner in Form eines 32 byte Wertes, die Parameterwerte dieser Methode und deren Rückgabewert. Die Methode `add()` in der Klasse `OffloadingCache` prüft zunächst, ob bereits eine Methode mit diesem Bezeichner, also ein `OffloadingCacheEntry`, vorhanden ist. Ist dies der Fall so kann entweder

- Ein Eintrag mit dieser Methode, genau diesen Parametern und Parameterwerten ist bereits vorhanden.
- Ein Eintrag mit dieser Methode aber mit unterschiedlichen Parametern oder Parameterwerten ist bereits vorhanden.

In beiden Fällen muß die Datenstruktur durchsucht werden, da im zweiten Fall ein neuer Eintrag hinzugefügt werden muß. Das Durchsuchen erfolgt wie unten beschrieben. Wird kein Eintrag gefunden wird eine neue `ParaList` mit den passenden `ParaEntry` Werten zum `OffloadingCacheEntry` hinzugefügt.

Eintrag suchen

Die Suche nach einem Eintrag geschieht in der Methode `getRetValue()`. Zunächst wird in der Hashmap nachgesehen ob bereits ein Eintrag für diese Methode, in Form eines 32 Byte Wertes, vorhanden ist. Ist dies nicht der Fall kann bereits abgebrochen werden. Ansonsten werden in dem zu dieser Methode gehörenden `OffloadingCacheEntry` die einzelnen `ParaList` Einträge nach einer Übereinstimmung durchsucht. Dazu werden die `ParaEntry` Einträge in den `ParaList` Listen Paarweise verglichen. Wird ein Eintrag gefunden wird der dazugehörige Rückgabewert zurückgeliefert.

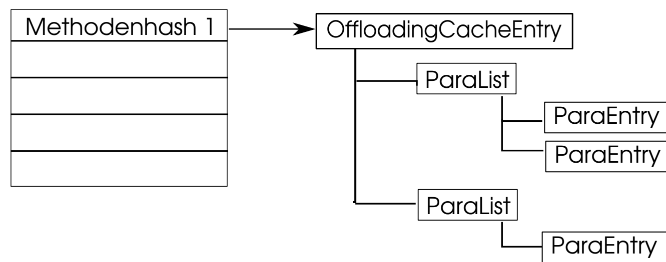

Abbildung 5.1: Datenstruktur zum Speichern von Methoden, deren Parameter und Rückgabewert

6 Evaluation

Im folgenden Kapitel wurde der in dieser Arbeit entwickelte Caching Server anhand zweier typischen Beispielanwendungen getestet.

6.1 Testumgebung

Die folgende Evaluation erfolgte mit der in dieser Arbeit erweiterten Jikes RVM als Offloading Client und Server sowie mit dem entwickelten Cache Server. Die Jikes RVM war zum Testzeitpunkt nicht auf ARM Architekturen läuffähig. Alles Tests wurden daher auf x86 Architekturen durchgeführt.

Für die folgenden Testdurchläufe standen drei unterschiedlich leistungsfähige Computer zur Verfügung.

- Ein Dell Inspiron Mini v10 (1011) **Netbook** mit einem Intel Atom N270 CPU bei einer Taktrate von 1.60GHz
- Lenovo ThinkPad T61 **Laptop** mit einem Intel Core 2 Duo T7300 CPU bei einer Taktrate von 2.00GHz
- Ein **Cloudserver** mit Intel Xeon E312xx CPU

Das Netbook ist dabei leistungsmäßig vergleichbar mit heutigen Smartphones. Es dient daher in den folgenden Betrachtungen als Offloading Client. Laptop und Cloudserver dienen jeweils als Offloading Server bzw. Caching Server. Offloading Server und Caching Server wurden jeweils auf dem selben Computer ausgeführt. Diese drei Rechner waren über ein kabelgebundenes Ethernet Netzwerk miteinander verbunden.

Alles Testläufe wurden unter 32bit GNU/Linux Debian Wheezy durchgeführt.

6.2 Getestete Anwendungen

Die folgende Evaluation wurde mit zwei unterschiedlichen Anwendungsprogrammen durchgeführt.

6.2.1 Schach

Schachprogramme fallen in eine Kategorie von Anwendungen, die sehr von Offloading profitieren. Dies hat mehrere Gründe. Die Primärfunktionalität, also das Suchen nach dem nächsten Zug, läßt sich klar abgrenzen. Zudem bindet sie den Großteil der vom Schachprogramm benötigten Ressourcen. Gerade die Suche nach den nächsten Zug benötigt oft mehrere Sekunden. Wenn hier eine Halbierung der Ausführungszeit durch Offloading erreicht werden kann, fällt selbst eine Kommunikationslatenz von mehreren 100 Millisekunden kaum ins Gewicht.

Hinzu kommt, dass die Suche nach dem nächsten Zug nur sehr wenige Eingangsparameter benötigt. Je nach Schachprogramm reichen als Eingangsparameter das Schachbrett und die maximale Suchtiefe. Als Rückgabewert reicht der gefundene Zug. Somit ist auch die Menge an Daten, die zwischen Offloading Client und Server ausgetauscht werden müssen nur minimal.

Diese Klasse von Anwendungen wurde ausgewählt um zu evaluieren, wie sich ein zusätzliches Caching auf Anwendungen, die bereits stark vom Offloading profitieren, auswirkt.

6.2.2 Gesichtserkennung

Die Gesichtserkennung fällt, wie das Schachprogramm, in die Kategorie von Anwendungen mit einer rechenintensiven Primärfunktionalität. In der hier getesteten Variante wird der Erkennungsroutine jeweils ein Bild übergeben. Diese Routine liefert dann ein Bild der selben Dimension, in dem die Flächen, die als ein Gesicht erkannt wurden, weiß eingefärbt wurden. Der Rest des Bildes ist schwarz. Ein Beispiel ist in Abbildung 6.1 zu sehen.

Was die Gesichtserkennung vom Schachprogramm unterscheidet ist, dass hier die eigentliche Hauptroutine, also das Erkennen von Gesichtern, deutlich mehr Eingabedaten benötigt. So muß dieser das gesamte Bild übergeben werden was schnell mehrere hundert Kilobyte oder Megabyte bedeuten kann. Das selbe gilt auch für den

Rückgabewert, der ebenfalls ein Bild enthält. Diese Daten müssen beim Offloading bzw. Caching gegebenenfalls über das Netzwerk übertragen werden.

Die Gesichtserkennung steht also für Anwendungen mit einer rechenintensiven Primärfunktionalität und einem hohen Kommunikationsvolumen. Auch hier wurde untersucht, wie sich ein zusätzliches Caching auswirkt.

(a) Ausgangsbild[1]

(b) Erkannte Gesichter

Abbildung 6.1: Beispiel Gesichtserkennung

6.3 Szenarien

Für die Evaluation wurden vier Szenarien entworfen. Im Vordergrund stand dabei die Untersuchung, ob und welche Verbesserungen ein Caching im Gegensatz zu einem reinen Offloading bringt. Eine Verbesserung kann in zwei Aspekten erfolgen.

Zum einen in einer schnelleren Ausführung. Dies hat für den Energieverbrauch des Offloading Clients und damit des Smartphones nur eine geringe Bedeutung. Durch eine schneller Ausführung wird die Zeit zwischen der Offloading Anfrage und der Antwort des Offloading Servers verkürzt. In dieser Zeit befindet sich der Offloading Client und damit das Smartphone im Leerlauf. Moderne Smartphonebetriebsysteme erkennen einen solchen Leerlauf und versetzten das Smartphone in einen Schlafmodus. Neben diesen Aspekten ist eine schnellere Ausführung immer zu begrüßen und kann eine Klasse von Anwendungen auf Smartphones ermöglichen, die vorher nicht realisierbar waren.

[1] https://flic.kr/p/7R58b8

Eine weitere Verbesserung kann sich in einem reduzierten Kommunikationsaufwand niederschlagen. Besonders die Kommunikation des Smartphones über eine Mobilfunkverbindungen hat Einfluß auf den Energieverbrauch, da die rechenintensiven Teile der Anwendung ausgelagert werden. Eine genaue Berechnung der für die Kommunikation aufgebrachten Energie gestaltet sich jedoch recht schwierig. Wie Saarinen et al. in [SSX+12] ausführen hängt der Energieverbrauch von mehreren, sich dynamisch ändernden Faktoren ab.

Eine Verbindung über Wlan hat ein ganz anderen Energieverbrauch als eine Verbindung über Edge im GSM Netz. Selbst die Netzwerkverkehr kurz vor und nach dem Offloading hat Auswirkung auf den Energieverbrauch des Offloading. Ebenso wie Prozessoren kennen auch Mobilfunknetze diverse Energiesparmodi, die je nach Netzwerkzustand wechseln. Es ist somit äußerst schwierig, statische Aussagen über den Energieverbrauch zu treffen. Offloading Systeme wie CloneCloud [CIM+11] versuchen daher, dynamisch während der Laufzeit ihr Verhalten zu optimieren, indem wechselnde Rahmenbedingungen gemessen und ausgewertet werden.

In der folgenden Evaluation wurde daher von einer einfacheren Prämisse ausgegangen. Wenn sich der Kommunikationsaufwand, im Gegensatz zum reinem Offloading, durch zusätzliches Caching reduzieren lassen kann, so ist es auch wahrscheinlich, dass der Energieverbrauch insgesamt geringer ausfällt.

Als Testarchitektur wurde ein Aufbau wie in Abbildung 6.2 gewählt. Durch die Verbindung aller Komponenten über Ethernet können Latenzen in der Kommunikation vernachlässigt werden. Es ist daher für Messungen der Ausführungszeit unerheblich, auf welcher der oben genannten Systemen der Offloading Cache ausgeführt wird. Wo Offloading Client bzw. Server ausgeführt werden wird in den jeweiligen Szenarien besprochen.

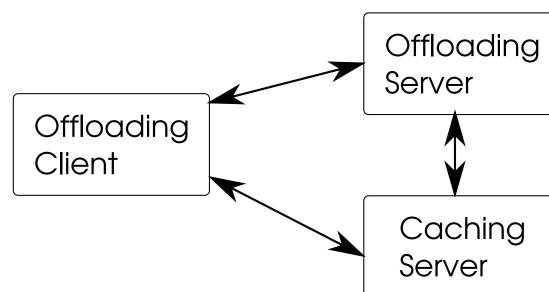

Abbildung 6.2: Evaluationsaufbau

Folgende Szenarien wurden in dieser Evaluation genauer betrachtet.

Offloading Client	Offloading Server
Netbook	Laptop
Netbook	Cloudserver
Laptop	Cloudserver

Tabelle 6.1: Zuordnungen von Computern und Offloading Systeme

Lokale Ausführung

Bei der lokale Ausführung werden die Programme mit dem jeweils gleichen Eingangsparametern auf den drei unterschiedlichen Rechnern ausgeführt. Dieses Szenarium soll zeigen, wie sehr die Testprogramme von der Rechenleistung der jeweiligen Computer profitieren und wie die Computer in ihrer Rechenleistung untereinander zu verorten sind.

Ausführung mit Offloading

In diesem Szenarium sollen die Ausführungszeiten beim klassischen Offloading gemessen werden. Die Testprogramme werden dazu mit der bereits in Abschnitt 5.1 auf Seite 39 besprochenen jRVM ausgeführt. Dazu sind sie sowohl auf dem Offloading Client als auch auf dem Offloading Server vorhanden. Die auszulagernden Methoden werden vor der Ausführung bestimmt. Die Zuordnung von Offloading System und Computern erfolgt wie in Tabelle 6.1 aufgeführt. Es wird immer von einem schwächeren auf eine leistungsfähigeres System ausgelagert.

Ausführung mit Offloading und Caching

Zusätzlich zu dem vorherigen Szenario wird in diesem, neben dem Offloading, auch der in Abschnitt 5.2 auf Seite 44 beschriebene Caching Server aktiviert. Offloading Client und Server werden wie in Tabelle 6.1 auf die unterschiedlichen Rechner verteilt. Wie bereits erwähnt macht die Positionierung des Caching Server keinen Unterschied. Er wird daher immer auf dem selben Rechner wie der Offloading Server gestartet.

Im ersten Lauf wird die Zeitmessung mit leerem Cache durchgeführt. Dies bedeutet das jede Anfrage, die der Offloading Client an den Cache stellt, negativ beantwortet wird. Darauf hin findet das Offloading an den Offloading Server statt. Dieser liefert sein

Rückgabewert an den Offloading Client und an den Caching Server. Dieser speichert diesen Rückgabewert zusammen mit der Methode und deren Parameter.

Im zweiten Durchlauf wird die Anwendung im Offloading Client mit den selben Eingabedaten noch einmal gestartet. Dieses Mal befinden sich im Cache Rückgabewerte für die angefragten Methoden und deren Parameter. Es können somit alle Anfragen des Offloading Clients aus dem Cache beantwortet werden. Der Offloading Cache ist in diesem Durchlauf nicht involviert.

Netzwerkverkehr

Im letzten Szenario wird der Netzwerkverkehr zwischen den beteiligten Systemen gemessen. Das Hauptaugenmerk wird dabei auf Verkehr zu und vom Offloading Client gelegt. Der Offloading Client läuft typischerweise direkt auf dem Smartphone. Daher ist Kommunikation mit dem Offloading Client Netzwerkverkehr über ein Funkstrecke. Im Mobilfunknetz ist die Bandbreite begrenzt, die Latenzen können hoch sein und es entstehen auch monetär höhere Kosten als bei kabelgebundenen Verbindungen.

Es ist daher interessant wie sich der Netzwerkverkehr durch ein zusätzliches Caching im Gegensatz zum reinen Offloading verändert. Dazu werden die Testdurchläufe des vorherigen Szenarios hier wiederholt. Nur dieses mal werden nicht die Ausführungszeiten gemessen sondern der Netzwerkverkehr des Offloading Clients. Dazu wird der Verkehr mit Hilfe von `tcpdump` auf Betriebsystemebene mitgeschnitten. Damit ist sichergestellt dass der gesamte Protokollstack mitgezählt wird.

In diesem Szenario ist von Bedeutung, wo sich der Cache in der in Kapitel 4.3 vorgestellten Architektur befindet. Zudem müssen INSERT und SEARCH Nachrichten unterschieden werden.

INSERT Nachrichten werden, wie bereits in Kapitel 4.3.2 besprochen, nur an Caches in darüberliegenden Tier weitergeleitet. Zudem propagieren Caches auf Tier 1 keine INSERT Nachichten an den darüberliegenden Tiers. Durch diese beiden Regeln erzeugen INSERT Nachrichten keinen Netzwerkverkehr auf der Funkstrecke.

SEARCH Nachrichten dagegen können an darüberliegende Tiers weitergeleitet werden. Ob eine SEARCH Nachricht und deren Antwort über die Funkstrecke zwischen Tier 1 und Tier 2 gehen hängt davon ab, ob der Cache auf Tier 1 bereits einen Eintrag für diese Anfrage enthält. In diesem Szenario wird von dem schlechten Fall ausgegangen dass dies nicht der Fall ist. Die SEARCH Nachicht und deren Antwort müssen über die Funkstrecke übertragen werden.

6.4 Testergebnisse

6.4.1 Schach

Um die folgenden Testergebnisse zu ermitteln wurde das Schachprogramm so eingestellt dass es die ersten 5 Züge gegen sich selbst spielt. Die Methode

```
public int[] search(int[][] board, final int steps, final int maxsteps, int alpha,
int beta)
```

wurde für das Offloading markiert. `board` beinhaltet das zweidimensionale Schachbrett. Die restlichen Parameter bestimmen die Spielstärke. Diese Parameter waren für alle Aufrufe fest voreingestellt. Als Rückgabewert liefert die Methode den gefundenen Zug in Form eines eindimensionalen Arrays.

Ausführungszeit Lokale

Für die lokale Ausführung wurde die Ausführungszeit der ersten 5 Züge des Schachprogramms ermittelt. Wie in Abbildung 6.3 zu sehen ist zeigt sich dabei gut das Leistungsgefälle der unterschiedlichen Computer. So ist der Laptop gut doppelt so schnell wie das Netbook und der Cloudserver ein drittel schneller als der Laptop. Zudem sieht man an den Ausführungszeiten sehr schön dass Schach ein sehr rechenintensive Klasse von Programmen darstellt. So dauert es bereits 6 Sekunden den 4. Zug auf dem Netbook zu ermitteln. Auch ist zu sehen dass im 5. Zug der Algorithmus schneller als im 4. Zug zu einer Entscheidung über den nächsten Zug gekommen ist.

Ausführungszeit mit Offloading

Abbildung 6.4 zeigt die Ausführungszeiten auf dem Netbook mit und ohne Offloading. Man sieht dass ein Offloading immer schneller ist als eine lokale Ausführung. Dies gilt sogar beim ersten Zug, wenn auch nur marginal mit 2858 Millisekunden auf dem Netbook lokal zu 2535 Millisekunden beim Offloading auf den Laptop. Das selbe gilt auch für den Laptop in Abbildung 6.5. Auch hier lohnt sich ein Offloading.

Vergleicht man die Ausführungszeiten zwischen Netbook und Laptop beim Offloading in die Cloud, so Zeit sich dass die Leistungsfähigkeit des Offloading Client keinen Einfluss auf die Ausführungszeit hat. Dies zeigt dass der rechenintensive Teil des

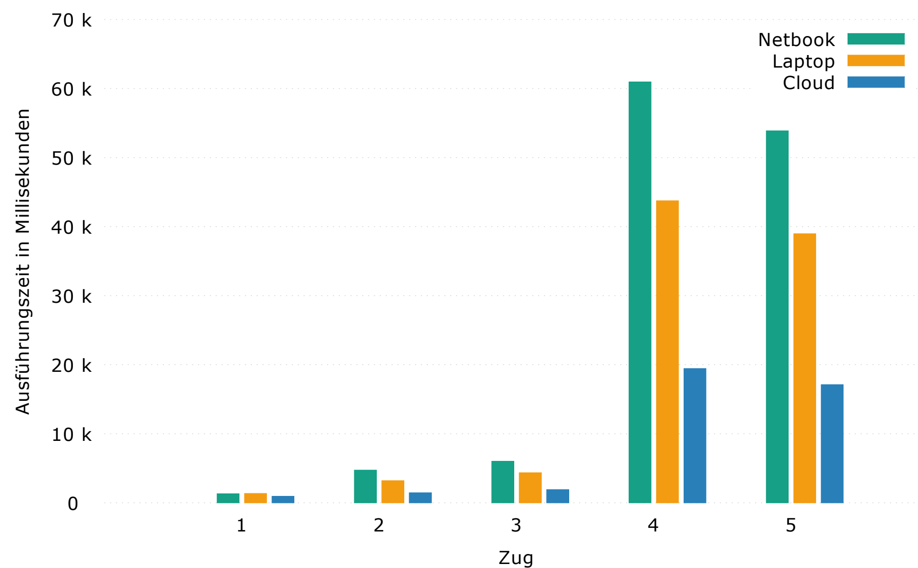

Abbildung 6.3: Ausführungszeiten für 5 lokale Schachzüge

Schachprogrammes vom Offloading Server berechnet wird. Der verbleibende ist marginal und macht messtechnisch zwischen Netbook und Laptop keinen Unterschied.

Überträgt man diese Ergebnisse auf ein schlechtes mobiles Funknetz mit Latenzen zwischen 100 und 200 Millisekunden lohnt sich das Offloading für den ersten Zug nicht mehr. Doch bereits der 2. Zug benötigt auf dem Laptop 2806 Millisekunden und auf dem Netbook bereits 4739 Millisekunden. Selbst bei einer sehr schlechten Latenz von 200 Millisekunden ist das Offloading auf dem Laptop schneller. Auch im Falle des Laptops ist die Zeitersparnis so groß dass sich selbst bei einer schlechten Leitung ein Offloading lohnt.

Ausführungszeit mit Offloading und Caching

Abbildung 6.6 zeigt die Ausführungszeiten für 5 Schachzüge jeweils mit und ohne Caching. Das Schachprogramm läuft dabei auf dem Netbook. Cache und Offloading System laufen jeweils auf dem Laptop oder Cloudserver.

Ist der Cache noch nicht gefüllt liefert dieser eine negative Antwort und es muß noch zusätzlich das Offloading ausgeführt werden. Hier zeigt sich, dass die zusätzliche Caching Anfrage durchaus Zeit benötigt. So schwanken der zusätzliche Zeitbedarf

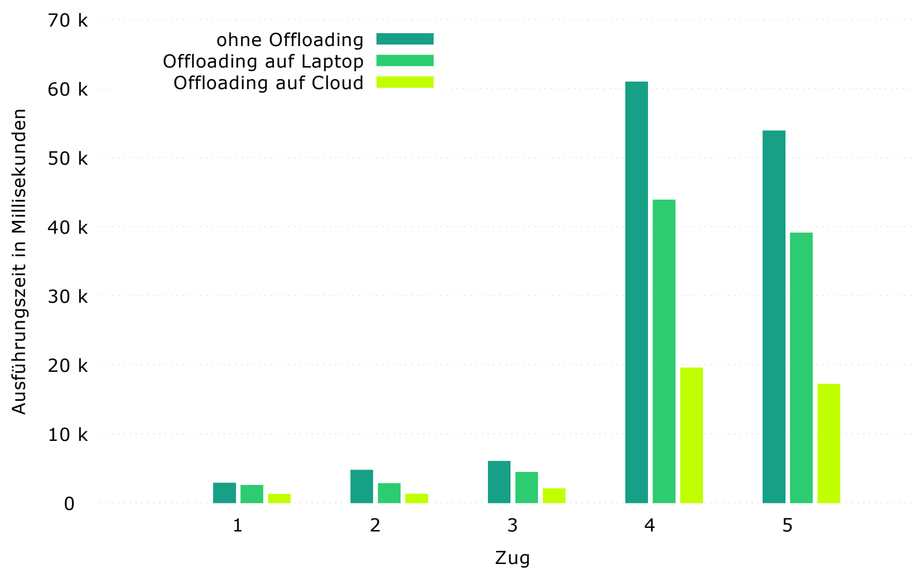

Abbildung 6.4: Ausführungszeiten für 5 Schachzüge auf dem Netbook als Offloading Client

zwischen 50 und 200ms. Ist der Cache jedoch mit der richtige Antwort gefüllt liefert dieser innerhalb von 50ms ein Ergebnis.

Das selbe Bild zeigt sich in Abbildung 6.7. Hier wird das Schachprogramm auf dem Laptop ausgeführt und der Cache bzw. das Offloading findet auf dem Cloudserver statt. Wird ein Eintrag im Cache gefunden erhält das Schachprogramm innerhalb von 17ms eine Antwort. Ist jedoch kein Eintrag vorhanden bedeutet der Cache einen zusätzlichen Zeitbedarf von ebenfalls 50 bis 200ms.

Werden Einträge im Cache gefunden, ist die Zeitersparnis sehr groß. So reduziert sich die Ausführungszeiten von bis zu 2 Sekunden auf durchgehend zweistellige Millisekunden. Geht man von einer durchschnittlichen Ausführungszeit von 10 Sekunden pro Zug bei einem durchschnittlichen Schachspiel mit einem Schachprogramm aus, so wird durch Caching bereits eine Zeitersparnis erreicht wenn nur $1/1000$ Züge im Cache gefunden werden kann.

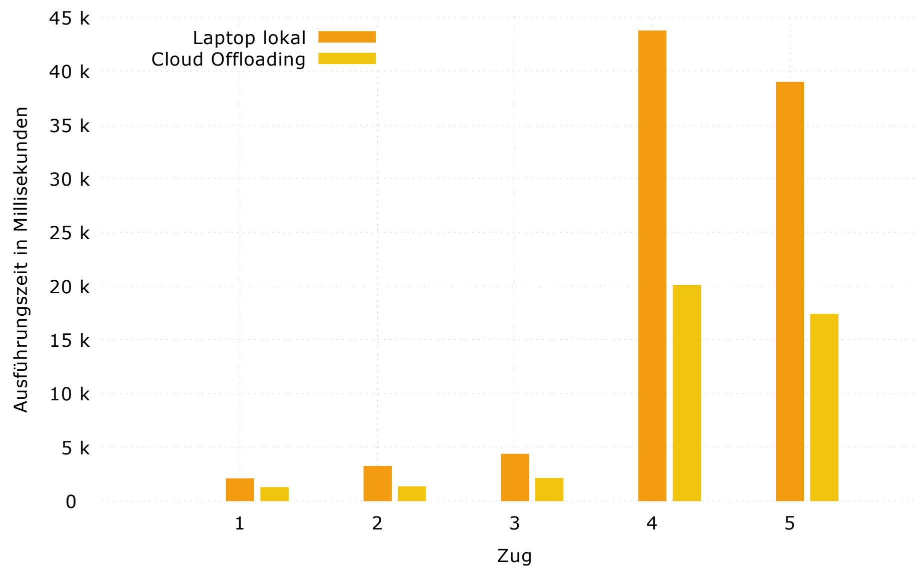

Abbildung 6.5: Ausführungszeiten für 5 Schachzüge auf dem Laptop als Offloading Client

Netzwerkverkehr

Der Netzwerkverkehr des Offloading Clients für 5 Schachzüge ist in Abbildung 6.8 zu sehen. Dabei wurde der Netzwerkverkehr für alle 5 Züge addiert. Der Netzwerkverkehr beinhaltet dabei sowohl die Kommunikation mit dem Offloading Server als auch mit dem Caching Server.

Es ist zu sehen dass für die fünf Züge nur ein sehr geringes Datenvolumen übertragen werden muß. Am meisten Daten müssen für das Offloading mit leerem Cache übertragen werden. Im Gegensatz zum reinen Offloading kommen hier noch die Kosten für die Kommunikation mit dem Cache hinzu.

Würden nicht der Caching Server auf Tier 2 die Anfragen beantworten, wie für dieses Szenario vorgesehen, sondern der Cache auf Tier 1, würde der Verkehr im Fall des vollen Caches auf 0 fallen. Alle Anfragen könnten lokal beantwortet werden. Für den Fall des leeren Caches wäre der Verkehr auf Niveau des reinen Offloading.

Dieses Szenario zeigt, dass bei Kommunikationsvolumen im einstelligen Kilobyte-Bereich ein zusätzliches Caching das Datenvolumen bei leerem Cache um $1/3$ steigert. Im umgekehrten Fall eines vollen Caches reduziert sich das Datenvolumen um $1/3$. In

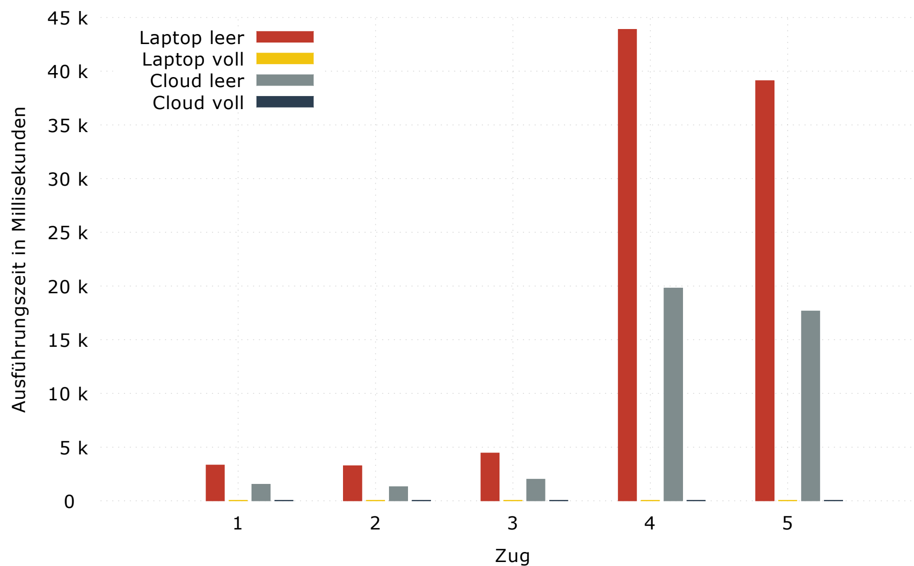

Abbildung 6.6: Ausführungszeiten für 5 Schachzüge auf dem Netbook mit Offloading und Cache

einem solchen Szenario hängt das Einsparpotential sehr stark von der Trefferwahrscheinlichkeit im Cache ab.

6.4.2 Gesichtserkennung

Die Messungen zur Gesichtserkennung wurden Anhand 5 unterschiedlicher Bilder durchgeführt. Die Bilder waren von unterschiedlicher Auflösung und in Folge dessen auch von unterschiedlicher Größe. In Tabelle 6.2 auf der nächsten Seite sind die unterschiedlichen Bildgrößen aufgeführt.

Das Offloading erfolgte, wie schon bei der Schachanwendung, mit Hilfe der jRVM. Diese Bilderkennung war sowohl auf dem Offloading Server als auch auf dem Offloading Client vorhanden. Zum Offloading wurde folgende Methode markiert.

```
public static String findFaces(String imageString, int minScale, int maxScale)
```

Im Parameter `imageString` wird ein Bild als Base64 [Jos06] kodierte Zeichenkette übergeben. `minScale` und `maxScale` setzten interne Erkennungsparameter und wur-

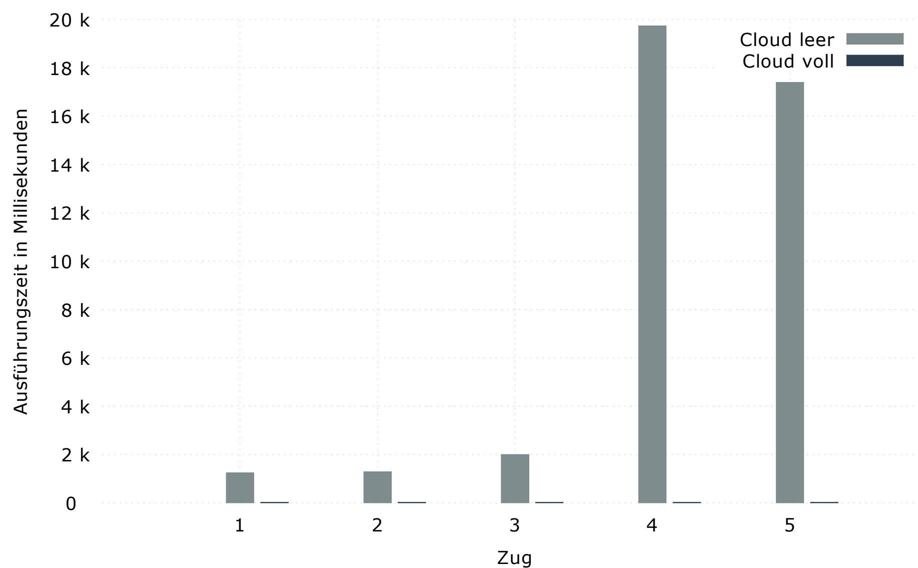

Abbildung 6.7: Ausführungszeiten für 5 Schachzüge auf dem Laptop mit Offloading Cache

Bildername	Bildgröße
Gruppe	380 Kilobyte
Menschen	157 Kilobyte
Titel	85 Kilobyte
Team	63 Kilobyte
Test	587 Kilobyte

Tabelle 6.2: Größen der bei der Gesichtserkennung verwendeten Bilder

den auf 1 und 40 gesetzt. Die Methode liefert als Rückgabewert ein als Base64 kodiertes Bild zurück.

Lokale Ausführung

Abbildung 6.9 zeigt die Ausführungszeiten auf Netbook, Laptop und dem Cloudserver. Hier zeigt sich, wie schon bei der Schachanwendung, dass es sich bei der Gesichtserkennung ebenfalls um eine rechenintensive Anwendung handelt. Auch zeigen sich

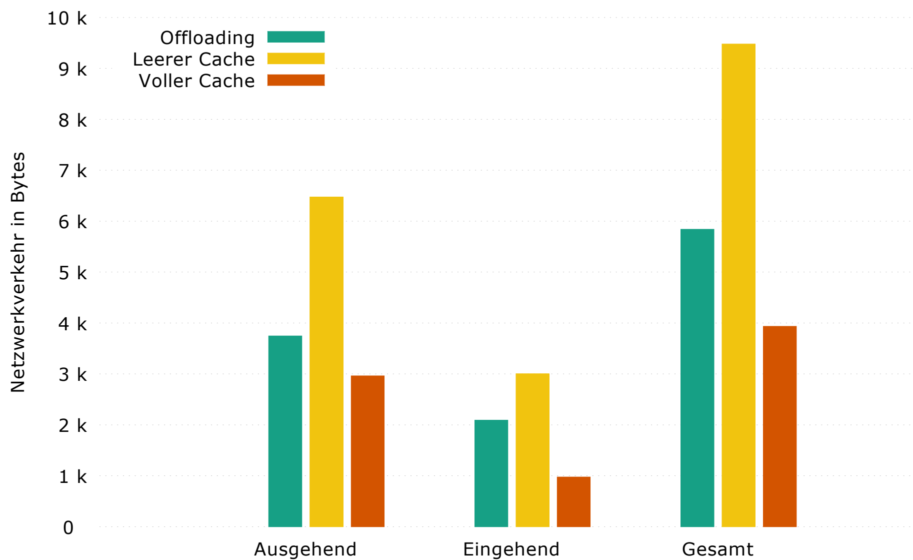

Abbildung 6.8: Netzwerkverkehr des Offloading Clients für 5 Schachzüge mit Offloading und Caching

wieder die gleichen Leistungsverteilungen zwischen den Computern. Das Netbook ist ca. halb so schnell wie der Laptop und dieser wiederum ein Drittel langsamer als der Cloudserver.

Zudem ist zu sehen, dass die Laufzeit des Gesichtserkennungsalgorithmus von der Größe des Bildes abhängt. Das Bild mit dem Namen *Test* ist mit 587 Kilobyte das größte Bild und benötigt auch am meisten Zeit.

Ausführung mit Offloading

Die Ausführungszeiten für die Gesichtserkennung mit dem Netbook sind in Abbildung 6.10 zu sehen. Man sieht, dass auch die Gesichtserkennung stark vom Offloading profitiert. Dies ist auch nicht weiter verwunderlich. Ebenso wie die Schachroutine ist auch der Gesichtserkennungsalgorithmus sehr berechnungsintensiv und daher prädestiniert für das Offloading.

Vergleicht man die Ausführungszeiten des Netbooks mit denen des Laptops in Abbildung 6.11 beim Offloading in die Cloud, so zeigt sich, dass die Leistungsfähigkeit

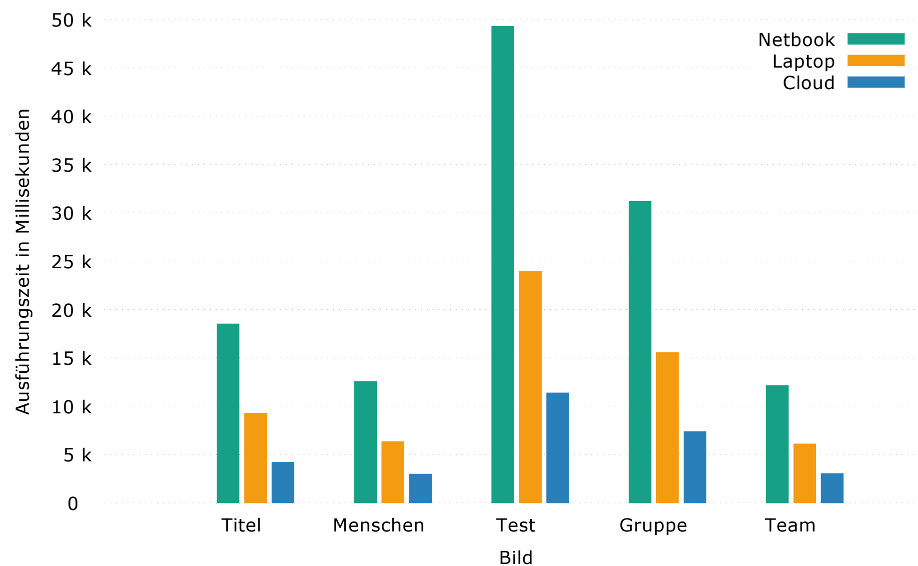

Abbildung 6.9: Ausführungszeiten für die Gesichtserkennung lokal

des Offloading Clients keinen Unterschied macht. Sowohl Netbook als auch Laptop erzielen die selben Ausführungszeiten.

Bei allen Bildern ist durch das Offloading eine Zeitersparnis von mehreren Sekunden zu verzeichnen. Geht man von einer schlechten Mobilfunkverbindung mit 25 Kilobyte/s aus, dann benötigt das größte Bild ca. 24 Sekunden für die Übertragung. Diesen 24 Sekunden stehen beim Netbook eine Zeitersparnis von 75 Sekunden und beim Laptop von 13 Sekunden entgegen.

Diese Beispiel zeigt dass, je größer die zu übertragenen Daten beim Offloading, um so mehr fällt die Datenverbindung ins Gewicht.

Ausführungszeit mit Offloading und Caching

Abbildung 6.12 zeigt die Ausführungszeiten für die Gesichtserkennung mit dem Netbook als Offloading Client. Als Offloading Server dienen jeweils der Laptop und der Cloudserver.

Am meisten Zeit benötigt die Gesichtserkennung, wenn kein passender Eintrag im Cache gefunden gefunden werden konnte. Diese Ausführungszeiten sind auch langsamer

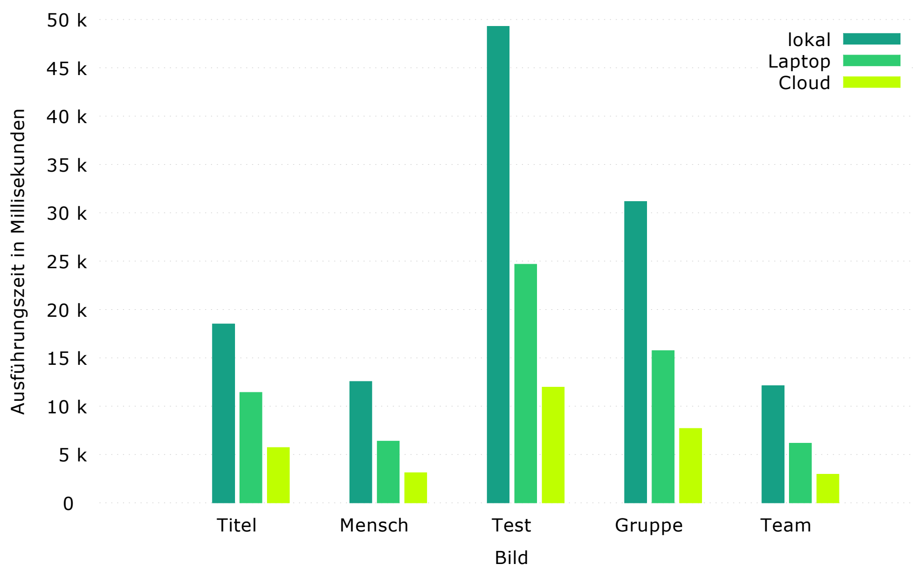

Abbildung 6.10: Ausführungszeiten für die Gesichtserkennung von fünf Bildern mit dem Netbook als Offloading Client

als das reine Offloading. So benötigt das reine Offloading ca. 24 Sekunden für das Bild *Test*. Bei vorherigem negativen Cachezugriff sind es 28 Sekunden. Diese große Zeitdiskrepanz ist vor allem der jRVM geschuldet. Wie bereits ausgeführt werden die Bilder als Base64 kodierte Zeichenketten an die Gesichtserkennungsmethode übergeben. Ebenfalls als Base64 kodierte Zeichenkette gibt die Methode das Bild mit den erkannten Gesichtern zurück. Für diese Umwandlungen benötigt die jRVM recht viel Zeit. Hinzu kommt, dass diese Zeichenketten für den Transport noch einmal konvertiert werden müssen. All dies zusammen addiert sich auf die beobachtete Verzögerung.

Bei gefülltem Cache ergeben sich enorme Zeitersparnisse. Hier fällt nur noch die eben erwähnte Konvertierung für die Anfrage beim Caching Server ins Gewicht. Im Schnitt wird nur $1/15$ der Ausführungszeit des reinem Offloading benötigt.

Das selbe Bild zeigt sich auch in Abbildung 6.13. Dort ist der Laptop Offloading Client und das Offloading findet auf dem Cloudserver statt. Im Gegensatz zum Schachprogramm zeigt sich aber hier der Leistungszuwachs gegenüber dem Laptop. Benötigt zB. das Netbook 934 Millisekunden für das Offloading von Bild *Titel* auf den Cloudserver so benötigt der Laptop hierfür nur 596 Millisekunden. Der auf dem Offloading Client verbliebene Rechenaufwand macht sich im Falle der Gesichtserkennung also durchaus bemerkbar.

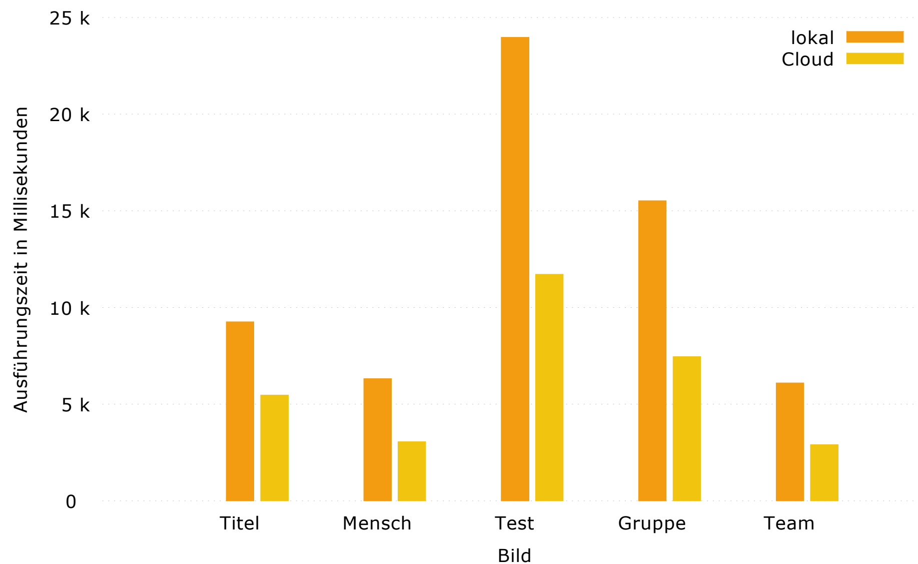

Abbildung 6.11: Ausführungszeiten für die Gesichtserkennung von fünf Bildern mit dem Laptop als Offloading Client

Netzwerkverkehr

Der durch das Offloading und Caching entstehende Netzwerkverkehr bei der Gesichtserkennung ist in Abbildung 6.14 abzulesen. Dabei wurde der Verkehr für alle fünf Bilder addiert. Der Netzwerkverkehr beinhaltet dabei die Kommunikation zwischen Offloading Client und Offloading Server sowie die Kommunikation zwischen Offloading Client und Caching Server.

Alleine schon am reinen Offloading läßt sich erkennen, dass sehr viel Daten vom Offloading Client gesendet werden. Dies liegt an den Bildern die zur Analyse an den Offloading Server übertragen werden müssen. Die Bilddateien alleine ergeben zusammen 1272 Kilobyte. Die über 5 Megabyte aus der Abbildung entstehen zum einen durch die Base64 Kodierung der Bilder und zum Anderen durch die Transportkonvertierung. Die Bilder mit den erkannten Gesichtern machen im Gegensatz zu den Originalbildern nur einen kleinen Bruchteil des Datenverkehres aus.

Ein leerer Cache ändert wenig an den übertragenen Daten. Durch die zusätzliche Kommunikation mit dem Cache steigt das Datenvolumen etwas. Können hingegen die Gesichtserkennungsanfragen aus dem Cache beantwortet werden, sinkt Datenvolumen massiv. Dies ist der in Kapitel 4.5.2 eingeführten Hash-Kodierung zu verdanken.

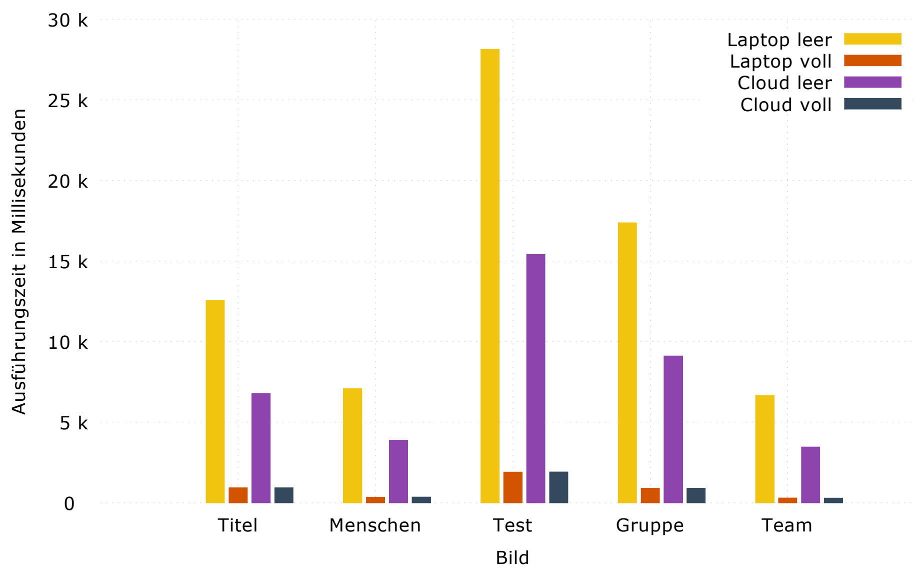

Abbildung 6.12: Ausführungszeiten für die Gesichtserkennung auf dem Netbook mit Offloading und Caching

Dadurch übermittelt der Offloading Client nicht das gesamte Bild an den Cache sonder nur ein Hash des Bildes. Dieser wird im Caching Server verglichen und dieser liefert letztendlich das passende Bild.

Dank dieser Maßnahme kann der ausgehende Netzwerkverkehr des Offloading Clients in diesem Beispiel von über 5 Megabyte auf 56 Kilobyte reduziert werden. Dies entspricht $1/90$ des Verkehrs der beim reinen Offloading entsteht.

In diesem Szenario zeigt sich, dass das in dieser Arbeit vorgeschlagene Caching gerade bei großen Datenmengen seine stärke ausspielen kann. Der zusätzliche Datenverkehr für Anfragen, die sich nicht im Cache befinden, spielt im Vergleich zum eigentlichen Offloading Datenverkehr kaum eine Rolle. Kann eine Anfrage jedoch aus dem Cache beantwortet werden ist die Reduzierung im Datenverkehr enorm.

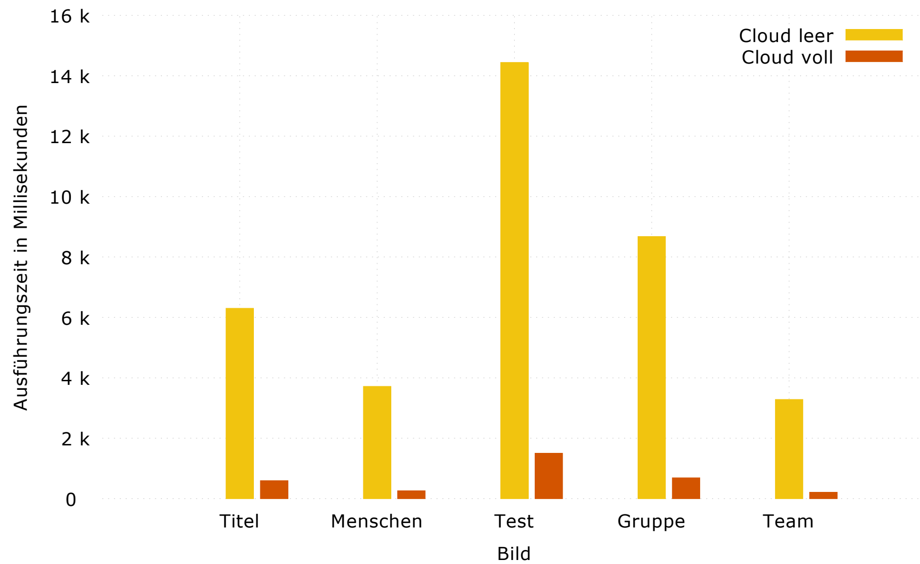

Abbildung 6.13: Ausführungszeiten für die Gesichtserkennung auf dem Laptop mit Offloading und Caching

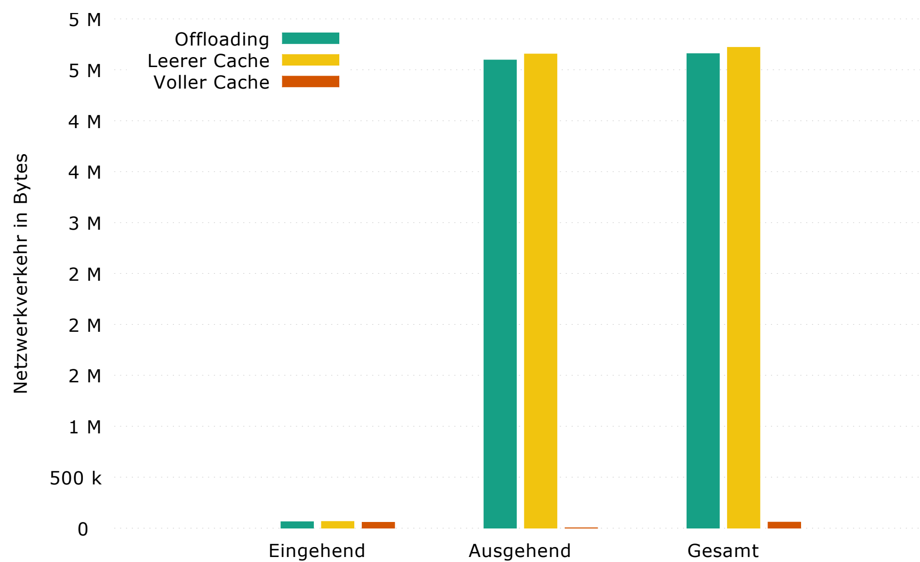

Abbildung 6.14: Netzwerkverkehr bei der Gesichtserkennung

7 Zusammenfassung und Ausblick

In dieser Arbeit wurde der Frage nachgegangen, ob eine Caching Komponente ein Code Offloading System im mobilen Umfeld sinnvoll ergänzen kann.

Dazu wurde zunächst die Caching Komponente in Form eines über das Netzwerk erreichbaren Server konzipiert und implementiert. Zur Anbindung an das Code Offloading System wurde ein Netzwerkprotokoll entworfen und in einem zuvor schon vorhandenen Offloading System implementiert.

Zuletzt wurden die Komponenten mit Hilfe eines Schach- und eines Gesichtserkennungsprogrammes in unterschiedlichen Szenarien evaluiert.

Dabei konnte gezeigt werden, dass das in dieser Arbeit entwickelte Caching System sowohl die Ausführungszeit beim Offloading, als auch den dabei entstehende Datenverkehr reduzieren kann. Dabei konnte gerade der Datenverkehr auf der Mobilfunkstrecke zwischen Smartphone und den dahinterliegenden Netz um bis zu 90% reduziert werden. Gerade diese Kommunikation stellt den Hauptenergieverbraucher beim Offloading auf Smartphones dar. Das hier vorgestellte Caching spart also auch Energie im Vergleich zum reinen Offloading.

Zudem wurde gezeigt, dass das Caching sich besonders für rechenintensive Aufgaben mit großen Eingangsdaten eignet. Bei Anwendungen diesen Typs konnten die größten Einsparungen festgestellt werden.

Je größer und je besser gefüllt ein Caching System ist, desto mehr Anfragen wird es positiv beantworten können. Um dies sicherzustellen wurden in dieser Arbeit mehrere Konzepte erarbeitet. So wurde ein mehrstufiges Caching Netzwerk zur Vergrößerung des Caches beschrieben. Außerdem wurde das Konzept der spezialisierten Caches eingeführt. Damit lassen sich Caches auf bestimmte Anwendungsklassen spezialisieren. Mit den ebenfalls vorgestellten Bereichsparameter lassen sich Einträge im Cache als Zwischenergebnis für weitere Berechnungen verwenden.

Ein nächster Schritt wäre nun, das vorgestellte System unter realen Bedingungen zu testen und zu sehen, ob die gemachten Annahmen ihre Gültigkeit behalten. Dazu

müßte die Implementierung auf eine Smartphone VM, wie zB. die Android Dalvik, portiert werden.

Zudem könnte der Caching Server um eine Persistenzschicht erweitert werden um die Einträge im Cache länger zu speichern. Hier wäre eine Anbindung an eine Datenbank denkbar.

Literaturverzeichnis

[BCF+99] L. Breslau, P. Cao, L. Fan, G. Phillips, S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, Band 1, S. 126–134. IEEE, 1999. (Zitiert auf Seite 22)

[BDR14] F. Berg, F. Dürr, K. Rothermel. Increasing the Efficiency and Responsiveness of Mobile Applications with Preemptable Code Offloading. In *Proceedings of the 3rd IEEE International Conference on Mobile Services: MS'14; Anchorage, Alaska, USA, June 27 - July 2, 2014*. IEEE Computer Society, 2014. (Zitiert auf den Seiten 22 und 26)

[Bri95] T. Brisco. DNS Support for Load Balancing. RFC 1794, Internet Engineering Task Force, 1995. URL `http://www.rfc-editor.org/rfc/rfc1794.txt`. (Zitiert auf Seite 30)

[CBC+10] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, P. Bahl. MAUI: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, S. 49–62. ACM, 2010. (Zitiert auf den Seiten 18 und 21)

[CDN+95] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, K. J. Worrell. A hierarchical Internet object cache. Technischer Bericht, DTIC Document, 1995. (Zitiert auf Seite 28)

[CIM+11] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, S. 301–314. ACM, 2011. (Zitiert auf den Seiten 18, 21 und 52)

[CK13] S. Cheshire, M. Krochmal. Multicast DNS. RFC 6762 (Proposed Standard), 2013. URL `http://www.ietf.org/rfc/rfc6762.txt`. (Zitiert auf Seite 34)

[CM09] B.-G. Chun, P. Maniatis. Augmented Smartphone Applications Through Clone Cloud Execution. In *HotOS*, Band 9, S. 8–11. 2009. (Zitiert auf Seite 14)

[CZB99] P. Cao, J. Zhang, K. Beach. Active cache: Caching dynamic contents on the web. *Distributed Systems Engineering*, 6(1):43, 1999. (Zitiert auf Seite 22)

[ErH06] D. Eastlake 3rd, T. Hansen. US Secure Hash Algorithms (SHA and HMAC-SHA). RFC 4634 (Informational), 2006. URL http://www.ietf.org/rfc/rfc4634.txt. (Zitiert auf den Seiten 34 und 44)

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, Internet Engineering Task Force, 1999. URL http://www.rfc-editor.org/rfc/rfc2616.txt. (Zitiert auf den Seiten 18 und 34)

[Fre] Free Software Foundation. GNU Classpath. URL http://www.gnu.org/software/classpath/. (Zitiert auf Seite 39)

[FT02] R. T. Fielding, R. N. Taylor. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002. (Zitiert auf Seite 17)

[Jac95] V. Jacobson. How to kill the Internet. *Talk at the SIGCOMM*, 95, 1995. (Zitiert auf Seite 10)

[Jos06] S. Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648 (Proposed Standard), 2006. URL http://www.ietf.org/rfc/rfc4648.txt. (Zitiert auf Seite 59)

[KL10] K. Kumar, Y.-H. Lu. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, 43(4):51–56, 2010. (Zitiert auf den Seiten 6 und 16)

[KPKB12] R. Kemp, N. Palmer, T. Kielmann, H. Bal. Cuckoo: a computation offloading framework for smartphones. In *Mobile Computing, Applications, and Services*, S. 59–79. Springer, 2012. (Zitiert auf den Seiten 17 und 21)

[Moc87] P. Mockapetris. Domain names - concepts and facilities. RFC 1034 (INTERNET STANDARD), 1987. URL http://www.ietf.org/rfc/rfc1034.txt. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936. (Zitiert auf Seite 18)

[PS05] J. A. Paradiso, T. Starner. Energy Scavenging for Mobile and Wireless Electronics. *IEEE Pervasive Computing*, 4(1):18–27, 2005. (Zitiert auf den Seiten 6 und 10)

[RCCS07] V. J. Reddi, D. Connors, R. Cohn, M. D. Smith. Persistent code caching: Exploiting code reuse across executions and applications. In *Proceedings of the international symposium on Code Generation and Optimization*, S. 74–88. IEEE Computer Society, 2007. (Zitiert auf Seite 22)

[SBCD09] M. Satyanarayanan, P. Bahl, R. Caceres, N. Davies. The case for vm-based cloudlets in mobile computing. *Pervasive Computing, IEEE*, 8(4):14–23, 2009. (Zitiert auf Seite 21)

[SLAZ12] C. Shi, V. Lakafosis, M. H. Ammar, E. W. Zegura. Serendipity: enabling remote computing among intermittently connected mobile devices. In *Proceedings of the thirteenth ACM international symposium on Mobile Ad Hoc Networking and Computing*, S. 145–154. ACM, 2012. (Zitiert auf Seite 22)

[Smi87] A. J. Smith. *Design of CPU cache memories*. Computer Science Division, University of California, 1987. (Zitiert auf Seite 19)

[SSX+12] A. Saarinen, M. Siekkinen, Y. Xiao, J. K. Nurminen, M. Kemppainen, P. Hui. Can offloading save energy for popular apps? In *Proceedings of the seventh ACM international workshop on Mobility in the evolving internet architecture*, S. 3–10. ACM, 2012. (Zitiert auf Seite 52)

[The] The Jikes RVM Project. Jikes RVM. URL `http://jikesrvm.org/`. (Zitiert auf den Seiten 26 und 39)

[W3C] W3C. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). URL `http://www.w3.org/TR/soap12-part1/`. (Zitiert auf Seite 17)

[YCZ04] C. Yuan, Y. Chen, Z. Zhang. Evaluation of edge caching/off loading for dynamic content delivery. *Knowledge and Data Engineering, IEEE Transactions on*, 16(11):1411–1423, 2004. (Zitiert auf Seite 22)

Alle URLs wurden zuletzt am 2. 11. 2014 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift