Efficient Code Offloading Techniques
for Mobile Applications

Von der Fakultät Informatik, Elektrotechnik und
Informationstechnik der Universität Stuttgart
zur Erlangung der Würde eines Doktors der Naturwissenschaften
(Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Florian Andreas Berg
aus Ludwigsburg

Hauptberichter: Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Mitberichter: Prof. Dr. phil. nat. Christian Becker

Tag der mündlichen Prüfung: 20.12.2017

Institut für Parallele und Verteilte Systeme (IPVS)
der Universität Stuttgart
2017
Acknowledgements

At this point, I take the opportunity to thank all the people that continuously supported and encouraged me during my research making this dissertation possible.

Above all, I express my special thanks and appreciation to my doctoral supervisor, Prof. Dr. Kurt Rothermel, who give me the opportunity to conduct my research in his group. His appropriate guidance, excellent support, and prolific discussions during my whole time at the University of Stuttgart were invaluable contributions for the success of my research presented in this thesis. He always gave helpful feedback and advice, asking the right questions whenever necessary. The great conditions provided in his group for learning and enhancing fundamental skills let me grow as a research scientist.

Furthermore, my thanks also go to the co-advisor of my thesis, Prof. Dr. Christian Becker, for his time and effort to review my dissertation giving me appropriate feedback.

Special thanks also go to my great project supervisor, Dr. Frank Dürr, for his encouragement, support, and (technical as well as personal) help during my research with many inspiring discussions and constructive feedback, sometimes to the detriment of his full schedule. His love and dedication to research like pursuing new ideas inspired not only me but also the approaches presented in this dissertation significantly.

During my research in the Distributed Systems group at the University of Stuttgart, I met exceptional colleagues, creating an amazing atmosphere within the institute. The productive and friendly working environment was both inspiring and fun, why I had a great time as a doctoral researcher – not only due to the countless matches at the soccer table. The brilliant, enthusiastic colleagues inspired me with new ideas, provided to me insightful feedback, and improved my research work a lot. In particular, I want to mention (in alphabetical order) Thomas Bach, Dr. Patrick Baier, Dr. Andreas Benzing, Ben Carabelli, Christoph Dibak, Dr. Stefan Föll, Hannes Hannak, Thomas Kohler, Dr. Boris Koldehofe, Christian Mayer, Ruben Mayer, Naresh Nayak, Dr. Damian Philipp, Stephan Schnitzer, David Schäfer, and of course all the other members of the research group. Having worked in the ARAMiS project – Automotive, Railway and Avionics Multicore Systems – over several years, I enjoyed very much the collaboration with researchers from other universities and employees from companies
Acknowledgements

all across Germany, elaborating on interesting research problems. In this context, I also express my gratitude towards the “Bundesministerium für Bildung und Forschung” for partially funding my research through the ARAMiS project. This funding enabled my research in the first place and allowed me to present my results to the research community on various international conferences. Also, I am grateful to Annemarie Rösler, Eva Strähle, and Martin Brodbeck for supporting me in administrative matters and thus, allowing me to focus on my research.

Finally, I thank my family and friends for their strong support, encouragement, and patience over all these years, sharing with me the joy of good times and supporting me through hard times. Special thanks go to my parents Günther and Regina, my brother Markus, and my sisters Franziska and Julia for their care and love. I am very happy to know that I can count on them not only during this important stage of my life but also in the future whatever happens next.

The last “thank you” goes out to my wife Corinna and my son Jonas, who were and are always there for me, why I am deeply grateful for having you both in my life.

Thank you all very much!

Florian Andreas Berg, December 20, 2017
<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
</table>

Abstract 21

Deutsche Zusammenfassung 25

1. Introduction 31
   1.1. Motivation 32
   1.2. Research Focus 34
   1.3. Contributions 36
   1.4. Structure of the Thesis 39

2. Background 43
   2.1. Mobile Computing 43
      2.1.1. Environment 44
      2.1.2. Architecture 44
      2.1.3. Limitations 45
   2.2. Wireless Network 46
      2.2.1. Wireless Fidelity Network 46
      2.2.2. Cellular Network 47
      2.2.3. Bluetooth Network 49
   2.3. Cloud Computing 50
      2.3.1. Essential Characteristics 50
      2.3.2. Service Models 51
      2.3.3. Deployment Models 51
   2.4. Technological Trends 52
   2.5. Mobile Cloud Computing 53
      2.5.1. Overview 53
      2.5.2. Architectures 54
      2.5.3. Framework 56
      2.5.4. Challenges 57
   2.6. Summary 58
### Contents

3. System Overview 61
   3.1. System Model 61
   3.2. Problem Statement 62
   3.3. System Components 65
      3.3.1. Offloading Client 66
      3.3.2. Offloading Service 66
      3.3.3. Communication Network 67
   3.4. Failure Model 67
   3.5. System Requirements 68
   3.6. Summary 69

4. Efficient Code Offloading with Annotations 71
   4.1. Basic Distribution 71
   4.2. System Overview 73
   4.3. Runtime-layer Offloading 75
      4.3.1. Overview 75
      4.3.2. Offloading Client 76
      4.3.3. Offloading Service 77
      4.3.4. Java Platform 78
   4.4. Offloading Timeline 82
   4.5. Offloading Framework 87
      4.5.1. Extended Java Compiler 87
      4.5.2. Offloading Client 87
      4.5.3. Offloading Service 94
   4.6. Implementation 96
      4.6.1. Jikes Research Virtual Machine 96
      4.6.2. Open Java Development Kit 97
      4.6.3. Android Open-Source Project 98
      4.6.4. Modifications 99
      4.6.5. Measurement Boards 100
   4.7. Evaluation 102
      4.7.1. Setup 102
      4.7.2. Results 104
   4.8. Summary 111

5. Robust Code Offloading through Safe-point’ing 113
   5.1. Preemptable Distribution 113
## Contents

5.2. System Overview ........................................... 115
5.3. Offloading Timeline ....................................... 116
5.4. Offloading Framework ..................................... 118
  5.4.1. Offloading Client ..................................... 118
  5.4.2. Offloading Service ................................... 120
5.5. Evaluation .................................................. 125
  5.5.1. Setup .................................................. 125
  5.5.2. Results ............................................... 126
5.6. Summary ................................................... 134

6. Deadline-aware Code Offloading with Predictive Safe-point’ing 137
  6.1. Predictive Distribution ................................. 137
  6.2. System Overview ....................................... 139
  6.3. Optimal Schedule for Safe-point’ing .................... 141
    6.3.1. Prediction of Link Connectivity .................... 145
    6.3.2. Prediction of Remaining Runtime .................... 147
  6.4. Evaluation .............................................. 148
    6.4.1. Setup .............................................. 148
    6.4.2. Results ............................................ 153
  6.5. Summary ............................................... 165

7. Optimized Code Offloading through Cooperative Caching 167
  7.1. Caching-aware Distribution ............................. 167
  7.2. System Overview ....................................... 169
    7.2.1. System Model ....................................... 170
    7.2.2. Problem Statement ................................ 170
    7.2.3. System Components ................................ 173
  7.3. Offloading Timeline .................................... 174
  7.4. Offloading Framework .................................. 176
    7.4.1. Offloading Client .................................. 176
    7.4.2. Offloading Service ................................ 178
    7.4.3. Caching Service .................................... 178
  7.5. Evaluation .............................................. 179
    7.5.1. Setup .............................................. 180
    7.5.2. Results ............................................ 182
  7.6. Summary ............................................... 190
## Contents

8. **Code Offloading in Environments with Multiple Tiers** 193
   8.1. Bubbling Distribution ............................................. 194
   8.2. System Overview .................................................. 196
   8.3. Offloading Timeline .............................................. 198
   8.4. Offloading Framework ............................................ 200
       8.4.1. Application Programming Interface ....................... 200
       8.4.2. Offloading Client ........................................... 202
       8.4.3. Offloading Service ......................................... 204
       8.4.4. Tier Service ................................................ 205
   8.5. Evaluation ........................................................ 206
       8.5.1. Setup ......................................................... 207
       8.5.2. Results ....................................................... 209
       8.5.3. Securing Overhead .......................................... 219
   8.6. Summary .......................................................... 220

9. **Related Work** ...................................................... 223
   9.1. Efficient Code Offloading with Annotations ................... 224
   9.2. Robust Code Offloading through Safe-point’ing ............... 226
   9.3. Deadline-aware Code Offloading with Predictive Safe-point’ing ... 228
   9.4. Optimized Code Offloading through Cooperative Caching ........ 228
   9.5. Code Offloading in Environments with Multiple Tiers ........... 230
   9.6. Summary .......................................................... 231

10. **Conclusion** ......................................................... 233
    10.1. Summary .......................................................... 233
    10.2. Outlook .......................................................... 237

**Appendix** .............................................................. 239

A. **Java Bytecode Instructions** ...................................... 241

B. **Mobile Applications** .............................................. 245
   B.1. “Hello, World!” Application .................................... 245
   B.2. Chesspresso Application ......................................... 245
   B.3. Chess Game ........................................................ 249
   B.4. Face Recognition Application ................................... 253
   B.5. Text-to-Voice Application ....................................... 255
List of Figures

2.1. A Typical Two-tier Architecture from Mobile Cloud Computing . . . . . 54
2.2. A Multi-tier Architecture from Mobile Cloud Computing . . . . . . . . 55
3.1. An Exemplary Call Graph $G(V, E)$ for an Application $A$ . . . . . . . . 63
3.2. The System Components for Code Offloading . . . . . . . . . . . . . . 65
4.1. The System Components for the Basic Distribution . . . . . . . . . . . . 74
4.2. Overview of (a) the Software Stack and (b) the Run-time Environment
ton a Resource to Execute Portable Code from an Application . . . . . . . . 75
4.3. The Extended Runtime Environment for the Offloading Client Enabling
   Code Offloading . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
4.4. The Extended Runtime Environment for the Offloading Service Enabling
   Code Offloading . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
4.5. Overview of the Internal Architecture for a Java Virtual Machine [Venry] . 80
4.6. Offloading Timeline of the Basic Distribution for an Application Part . . 83
4.7. Overview of (a) the Timeline and (b) the Decision Making for Code
   Offloading . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
4.8. The Runtime Environment on the Offloading Client for the Basic Dis-
   tribution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
4.9. The Runtime Environment on the Offloading Service for the Basic Dis-
   tribution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
4.10. The Software Stack from the Android OS [And16a, And16b] . . . . . . . . 97
4.11. The Measurement Board for the Samsung Galaxy Nexus . . . . . . . . . 100
4.12. The Measurement Board for the Dell Inspiron Mini 10v and the Lenovo
   ThinkPad T61 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101
4.13. Power Consumption on the Netbook for a Local Execution and the Basic
   Distribution of the Chesspresso Application . . . . . . . . . . . . . . . . 105
4.14. Power Consumption on the Laptop for a Local Execution and the Basic
   Distribution of the Chesspresso Application . . . . . . . . . . . . . . . . 105
List of Figures

4.15. Power Consumption on the Smart Phone for a Local Execution and the Basic Distribution of the Mobile Application ........................................ 110

5.1. The Preemptable Distribution of an Application Part (a) without and (b) with Failures between an Offloading Client and an Offloading Service via a Communication Network ........................................ 117

5.2. Overview of the Runtime Environment on the Offloading Client for the Preemptable Distribution ......................................................... 119

5.3. Overview of the Runtime Environment on the Offloading Service for the Preemptable Distribution ......................................................... 120

5.4. Power Consumption on the Netbook for the Preemptable Distribution of the Chesspresso Application ...................................................... 127

5.5. Power Consumption on the Laptop for the Preemptable Distribution of the Chesspresso Application ...................................................... 128

5.6. Power Consumption on the Netbook for a Remote Execution of the Chesspresso Application based on the Preemptable Distribution with Failures at \( t_e = 16.14 \text{s}, 16.31 \text{s}, \) and \( 17.31 \text{s} \) .......................................................... 129

5.7. Power Consumption on the Netbook for a Remote Execution of the Chesspresso Application based on the Preemptable Distribution with failures at \( t_e = 18.54 \text{s}, 17.97 \text{s}, \) and \( 19.02 \text{s} \) .......................................................... 130

5.8. Power Consumption on the Laptop for a Remote Execution of the Chesspresso Application based on the Preemptable Distribution with a failure at \( t_e = 5.424 \text{s} \) .............................................................. 132

5.9. Power Consumption on the Laptop for a Remote Execution of the Chesspresso Application based on the Preemptable Distribution with a failure at \( t_e = 6.146 \text{s} \) .............................................................. 133

5.10. Power Consumption on the Laptop for a Remote Execution of the Chesspresso Application based on the Preemptable Distribution with a failure at \( t_e = 6.749 \text{s} \) .............................................................. 133

6.1. An Exemplary Execution Graph for a Java Method at Compile-time. ... 141

6.2. An Exemplary Modification of Variables for Each Execution Block ... 143

6.3. An Exemplary Execution Graph for a Java Method at Run-time ..... 147

6.4. Power Consumption on the Laptop for (a) a Local Execution and (b) a Remote Execution of the Chesspresso Application .......................... 149

6.5. A Section of the Topographic Map from the Public Transport of Stuttgart .......................... 150

6.6. Frequency of Execution Times for the Chesspresso Application on the Netbook based on the Connectivity Traces for T-Mobile .......................... 153
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.7</td>
<td>Frequency of Energy Consumption for the Chesspresso Application on the Netbook based on the Connectivity Traces for T-Mobile</td>
<td>154</td>
</tr>
<tr>
<td>6.8</td>
<td>Frequency of Execution Times for the Chesspresso Application on the Netbook based on the Connectivity Traces for O₂</td>
<td>155</td>
</tr>
<tr>
<td>6.9</td>
<td>Frequency of Energy Consumption for the Chesspresso Application on the Netbook based on the Connectivity Traces for O₂</td>
<td>156</td>
</tr>
<tr>
<td>6.10</td>
<td>Frequency of Execution Times for the Chesspresso Application on the Laptop based on the Connectivity Traces for T-Mobile</td>
<td>157</td>
</tr>
<tr>
<td>6.11</td>
<td>Frequency of Energy Consumption for the Chesspresso Application on the Laptop based on the Connectivity Traces for T-Mobile</td>
<td>158</td>
</tr>
<tr>
<td>6.12</td>
<td>Frequency of Execution Times for the Chesspresso Application on the Laptop based on the Connectivity Traces for O₂</td>
<td>159</td>
</tr>
<tr>
<td>6.13</td>
<td>Frequency of Energy Consumption for the Chesspresso Application on the Laptop based on the Connectivity Traces for O₂</td>
<td>159</td>
</tr>
<tr>
<td>6.14</td>
<td>Frequency of Execution Time for the Face Recognition Application on the Netbook based on the Connectivity Traces for T-Mobile</td>
<td>160</td>
</tr>
<tr>
<td>6.15</td>
<td>Frequency of Energy Consumption for the Face Recognition Application on the Netbook based on the Connectivity Traces for T-Mobile</td>
<td>161</td>
</tr>
<tr>
<td>6.16</td>
<td>Frequency of Execution Time for the Face Recognition Application on the Netbook based on the Connectivity Traces for O₂</td>
<td>162</td>
</tr>
<tr>
<td>6.17</td>
<td>Frequency of Energy Consumption for the Face Recognition Application on the Netbook based on the Connectivity Traces for O₂</td>
<td>163</td>
</tr>
<tr>
<td>6.18</td>
<td>Frequency of Execution Time for the Face Recognition Application on the Laptop based on the Connectivity Traces for T-Mobile</td>
<td>163</td>
</tr>
<tr>
<td>6.19</td>
<td>Frequency of Energy Consumption for the Face Recognition Application on the Laptop based on the Connectivity Traces for T-Mobile</td>
<td>164</td>
</tr>
<tr>
<td>6.20</td>
<td>Frequency of Execution Time for the Face Recognition Application on the Laptop based on the Connectivity Traces for O₂</td>
<td>165</td>
</tr>
<tr>
<td>6.21</td>
<td>Frequency of Energy Consumption for the Face Recognition Application on the Laptop based on the Connectivity Traces for O₂</td>
<td>165</td>
</tr>
<tr>
<td>7.1</td>
<td>The System Components for the Caching-aware Distribution</td>
<td>173</td>
</tr>
<tr>
<td>7.2</td>
<td>Offloading Timeline of the Caching-aware Distribution for an Application Part</td>
<td>174</td>
</tr>
<tr>
<td>7.3</td>
<td>The Runtime Environment on the Caching Service for the Caching-aware Distribution</td>
<td>178</td>
</tr>
</tbody>
</table>
List of Figures

7.4. Execution Time, Energy Consumption, and Monetary Cost on the Netbook for the Evaluation of the Chess Game with the Different Opening Moves .......................................................... 183
7.5. Execution Time, Energy Consumption, and Monetary Cost on the Laptop for the Evaluation of the Chess Game with the Different Opening Moves ...................................................... 185
7.6. Execution Time, Energy Consumption, and Monetary Cost on the Netbook for the Evaluation of the Text-to-Voice Application ................................................................. 186
7.7. Execution Time, Energy Consumption, and Monetary Cost on the Laptop for the Evaluation of the Text-to-Voice Application ................................................................. 188
7.8. Execution Time, Energy Consumption, and Monetary Cost (a) on the Netbook and (b) on the Laptop for the Evaluation of the Mobile Scenario189

8.1. An Exemplary Environment of Multiple Tiers .................................................. 195
8.2. The System Components for the Bubbling Distribution ..................................... 197
8.3. The Runtime Environment on the Tier Service for the Bubbling Distribution .......... 205
8.4. Overview of the Different Environments Evaluated for the Bubbling Distribution ................................................................. 207
8.5. Execution Time of a Local Execution, the Two-tier Distribution, and the Bubbling Distribution Evaluated in the Environments with Two Tiers210
8.6. Energy Consumption of a Local Execution, the Two-tier Distribution, and the Bubbling Distribution Evaluated in the Environments with Two Tiers ................................................................. 211
8.7. Monetary Cost of a Local Execution, the Two-tier Distribution, the Multi-tier Distribution, and the Bubbling Distribution Evaluated in the Environments with Two Tiers and the Environment with Multiple Tiers 212
8.8. Power Consumption of the Bubbling Distribution Evaluated in the Environment $\text{ScC}$ ................................................................. 213
8.9. Execution Time of a Local Execution, the Multi-tier Distribution, and the Bubbling Distribution Evaluated in the Environment with Multiple Tiers ................................................................. 215
8.10. Energy Consumption of a Local Execution, the Multi-tier Distribution, and the Bubbling Distribution Evaluated in the Environment with Multiple Tiers ................................................................. 216

B.1. The Opening Move for the Chesspresso Application ........................................ 246
List of Figures

B.2. The Different Configurations of the Chess Board Evaluated for a Chess Game .................................................. 250
# List of Tables

4.1. Overview of Execution Time, Energy Consumption, and Monetary Cost on the Netbook and on the Laptop for the Evaluation of the Chess Game with Different Opening Moves ............................................. 107
4.2. Overview of Execution Time, Energy Consumption, and Monetary Cost on the Netbook and on the Laptop for the Evaluation of the Text-to-Voice Application ......................................................... 109

6.1. Timing Parameters Calibrated on Real-world Measurements for the Evaluation of the Predictive Distribution with Safe-points ............................................. 151

8.1. Overview of the Execution Time, the Energy Consumption, and the Monetary Cost for a Local Execution, the Two-tier Distribution, and the Bubbling Distribution Evaluated in the Environments with Two Tiers with and without Beneficial Resources ........................................................................ 218
8.2. Overview of the Execution Time, the Energy Consumption, and the Monetary Cost for a Local Execution, the Multi-tier Distribution, and the Bubbling Distribution Evaluated in the Environment with Multiple Tiers with and without Beneficial Resources ........................................................................ 218
8.3. Overview of Execution Time on an Offloading Service for the Basic Distribution and the Bubbling Distribution for Each Resource-intensive Application Part. ........................................................................ 219

A.1. Part I of the Java Bytecode Instructions from 00 to 86 [LYBB15] .... 241
A.2. Part II of the Java Bytecode Instructions from 87 to 147 [LYBB15] .... 242

B.1. Comparison of the Java Applications Evaluated for Code Offloading .... 246

C.1. Overview of the Capabilities from the System Devices Utilized for Performance Measurements of Code Offloading ............................................. 258
List of Algorithms

5.1. Algorithm of the Decision Making for Safe-point’ing from the Offload Controller on the Offloading Service .......................... 124
7.1. Algorithm of the Decision Making for the Caching-aware Distribution on the Offload Controller of the Offloading Client ............... 177
Abstract

Since the release of the first smart phone from Apple in the year 2007, smart phones in general experience a fast growth of rising popularity. A smart phone typically possesses among others a touchscreen display as user interface, a mobile communication for accessing the Internet, and a System-on-a-Chip as an integrated circuit of required components like a central processing unit. This pervasive computing platform derives its required power from a battery, where an end user runs upon it different kinds of applications like a calendar application or a high-end mobile game. Differing in the usage of the local resources from a battery-operated smart phone, a heavy utilization of local resources like playing a resource-demanding application drains the limited resource of energy in few hours. Despite the constant increase of memory, communication, or processing capabilities of a smart phone since the release in 2007, applications are also getting more and more sophisticated and demanding. As a result, the energy consumed on a smart phone was, still is, and will be its main limiting factor.

To prevent the limited resource of energy from a quick exhaustion, researchers propose code offloading for (resource-constrained) mobile devices like smart phones. Code offloading strives for increasing the energy efficiency and execution speed of applications by utilizing a server instance in the infrastructure. To this end, a code offloading approach executes dynamically resource-intensive parts from an application on powerful remote servers in the infrastructure on behalf of a (resource-constrained) mobile device. During the remote execution of a resource-intensive application part on a remote server, a mobile device only waits in idle mode until it receives the result of the application part executed remotely. Instead of executing an application part on its local resources, a (resource-constrained) mobile device benefits from the more powerful resources of a remote server by sending the information required for a remote execution, waiting in idle mode, and receiving the result of the remote execution.

The process of offloading code from a (resource-constrained) mobile device to a powerful remote server in the infrastructure, however, faces different problems. For instance, code offloading introduces some overhead for additional computation and communication on a mobile device. Moreover, spontaneous disconnections during a remote execution can cause a higher energy consumption and execution time than a
Abstract

local execution on a mobile device without code offloading. To this end, this dissertation addresses the whole process of offloading code from a mobile device not only to one but also to multiple remote resources, comprising the following steps:

1) First, code offloading has to identify feasible parts from an application for a remote execution, where the distributed execution of the identified application part is more beneficial than its local execution. A feasible part for a remote execution typically has the following properties: A low size of information required for transmission before a remote execution, a resource-intensive computation not accessing local sensors, and a low size of information required for transmission after a remote execution. In the area of identification of application parts for a remote execution, this dissertation presents an approach based on code annotations from application developers that automatically transforms a monolithic execution on a mobile device to a distributed execution on multiple heterogeneous resources. In contrast to related approaches in the literature, the annotation-based approach requires least interventions from application developers and end users, keeping the overhead introduced on a mobile device low.

2) For an application part identified for a remote execution, code offloading has to determine its execution side, executing the application part either on the local resources of a mobile device or on the remote resource at the infrastructure. In the area of determining the execution side for an application part, this dissertation presents the offloading problem, where a mobile device decides whether to execute an application part locally or remotely. Furthermore, this dissertation also presents an approach called “code bubbling” that shifts the decision making into the infrastructure. In contrast to related approaches in the literature, the decision-based approach on a mobile device and the bubbling-based approach minimize the execution time, energy consumption, and monetary cost for an application.

3) To determine the execution side for an application part identified for a remote execution, code offloading has to obtain different parameters from the application, participating resources, and utilized links. In the area of obtaining the information required from an application, this dissertation presents a bit-flipping approach that dynamically flips a bit at the modification of application-related information. Furthermore, this dissertation also presents an offload-aware Application Programming Interface (API) that encapsulates the application-related information required for code offloading. In contrast to related approaches in the literature, the bit-flipping approach and the offload-aware API provide an efficient gathering of information at run-time, keeping the overhead introduced on a mobile device low.

4) Beside the information from an application, code offloading has to obtain further information from participating resources and utilized links. In the area of obtaining the
information required from participating resources and utilized links, this dissertation presents the approach of code bubbling, already mentioned above. In contrast to related approaches in the literature, the bubbling-based approach makes the offload decision at the place where the related information occurs, keeping the overhead introduced on a mobile device, participating resources, and utilized links low.

5) In case of a remote execution of an application part, code offloading has to send the information required for a remote execution to the remote resource that subsequently executes the application part on behalf of the mobile device. In the area of sending the required information and executing an application part remotely, this dissertation presents code offloading with a cache on the remote side. The cache on the remote side serves as a collective storage of results for already executed application parts, avoiding a repeated execution of previously run application parts. In contrast to related approaches in the literature, the caching-aware approach increases the efficiency of code offloading, keeping the energy consumption, execution time, and monetary cost low.

6) While a remote resource executes an application part, code offloading has to handle the occurrence of failures like a failure of the remote resource or a disconnection. In the area of handling the occurrence of failures, this dissertation presents a preemptable offloading of code with safe-points. The preemptable offloading of code with safe-points enables an interruption of an offloading process and a corresponding continuation of a remote execution on a mobile device, without abandoning the complete result calculated remotely so far. Based on a preemptable offloading of code with safe-points, this dissertation further presents a predictive offloading of code with safe-points that minimizes the overhead introduced by safe-point’ing and maximizes the efficiency of a deadline-aware offloading. In contrast to related approaches in the literature, the preemptable approach with safe-point’ing increases the robustness of code offloading in case of failures. Furthermore, the predictive approach for safe-point’ing ensures a minimal responsiveness and a maximal efficiency of applications despite failures.

7) At the end of a remote execution of an application part, code offloading has to gather on the remote resource the required information after the execution and send this information to the mobile device. In the area of gathering the required information, a remote resource utilizes the same approaches as a mobile device, already mentioned above (cf. the bit-flipping approach and the offload-aware API).

8) Last, code offloading has to receive on the mobile device the information from a remote resource, install the information on the mobile device, and continue the execution of the application on the mobile device. In the area of installing the information and continuing the execution locally, a mobile device utilizes the approaches already mentioned above (cf. the bit-flipping approach and the offload-aware API).
Deutsche Zusammenfassung


Um einer allzu schnellen Entladung der begrenzten Energiekapazität eines batteriebetriebenen mobilen Endgerätes, beispielsweise eines Smartphones, entgegen zu wirken, schlagen Forscher in der Literatur den Ansatz des Code Offloadings vor. Der Ansatz des Code Offloadings strebt nach einer verbesserten Energieeffizienz und einer schnelleren...
Deutsche Zusammenfassung


1) Zuallererst muss das Code Offloading geeignete Anwendungsteile identifizieren, bei denen eine verteilte Ausführung gegenüber einer lokalen Ausführung des Anwendungsteils vorzuziehen ist. Ein geeigneter Anwendungsteil besitzt hierbei eine geringe Menge an benötigten Informationen, die für eine entfernte Ausführung übertragen werden müssen, eine ressourcenintensive Berechnung, die keine lokalen Ressourcen wie zum Beispiel einen Sensor benötigt, und eine geringe Menge an benötigten Informationen, die nach einer entfernten Ausführung übertragen werden müssen. Auf dem Gebiet der Identifikation von geeigneten Anwendungsteilen präsentiert die vorliegende Dissertation einen Ansatz, der auf Annotationen des Quellcodes vom Anwendungsentwickler basiert. Dieser Ansatz mit Hilfe von Annotationen ermöglicht es, automatisch eine monolithische Ausführung auf einem mobilen Endgerät in eine verteilte Ausführung auf mehreren verschiedenartigen Ressourcen zu transformieren. Im Gegensatz zu ver-
Deutsche Zusammenfassung

wandten Arbeiten in der Literatur benötigt der präsentierte Ansatz basierend auf Codeannotationen nur sehr geringe Hilfe vom Anwendungsentwickler und Benutzer des mobilen Endgerätes, wobei der Ansatz nur einen sehr geringen Mehraufwand auf dem mobilen Endgerät verursacht.


5) Im Falle einer entfernten Ausführung eines Anwendungsteils muss das Code Of-


8) Zuletzt muss das Code Offloading die benötigten Informationen von einer entfernten Ressource nach der Ausführung eines Anwendungsteils auf dem mobilen Endgerät
empfangen, die entsprechenden Informationen auf dem mobilen Endgerät installieren und darauf basierend die Anwendungsaufrührung auf dem mobilen Endgerät fortsetzen. Auf dem Gebiet der Installation von benötigten Informationen sowie der lokalen Fortsetzung einer Anwendungsaufrührung benutzt ein mobiles Endgerät die gleichen Ansätze des Bitflippings und der erweiterten Programmiertinstallstelle für das Code Offloading, die bereits weiter oben beschrieben wurden.
In the year 2007, Apple released the first model of its iPhone series, “going to reinvent the phone” [Inf07a]. Compared to popular phones at this time like the mobile phone Q from Motorola or the E62 from Nokia, the first iPhone combines a revolutionary user interface with exceeding hardware components like a wide multi-touch touchscreen, plenty of sensors, and a powerful processing unit. Furthermore, the iPhone (1st generation) acquires, disseminates, and shares information with services in the Internet based on multiple connectivity for data transfer like a Wireless Fidelity (Wi-Fi) or cellular connection. This new interplay of the revolutionary software together with the exceeding hardware from the first iPhone extended the typical functionality of a “traditional” mobile phone at this time, changing significantly the everyday use of mobile phones. Due to the “smarter” functionality compared to traditional mobile phones, the community calls such mobile phones as smart phones.

A smartphone is a ubiquitous computing platform with Internet access, powered by a battery. An end user executes on the hardware different applications like a web browser or a mobile game. Apart from a couple of applications pre-installed by default, a mobile operating system like iOS on the first iPhone enables an end user to individualize the functionality of the smart phone to its need. An end user can install and execute further applications from an application store like the Apple App Store, where the execution of (resource-friendly and resource-intensive) applications on a smart phone utilizes local resources differently. A heavy utilization of the constrained resources from a battery-operated smart phone drains its limited energy in few hours. For instance, Apple announced 5 hours of talk time, Internet use, or video playback or 16 hours of audio playback for the preliminary version of the first iPhone [Inf07b]. For the final version, Apple further increased the battery life of the first iPhone, shipping it with 8 hours of talk time, 6 hours of Internet use, 7 hours of video playback, or 24 hours of audio playback. This fact clearly shows that already at the time of delivery the main limiting factor of the first smart phone from Apple was its constrained battery life.

Ten years later, the hardware and software of smart phones improved significantly. Regarding the current flagship smart phone from Samsung presented in March 2017,
1. Introduction

the Samsung Galaxy S8 [Sam17] possesses, for instance, a 3D multi-touch touchscreen with a higher resolution, additional sensors with higher accuracy, and a more powerful processing unit compared to the iPhone (1st generation). Furthermore, the capabilities of the connectivity for data transfer also improved significantly, where an end user now accesses services in the Internet via a high-bandwidth and low-latency connection – e.g., up to 1 Gbit/s for the down link, up to 500 Mbit/s for the up link, and a latency less than 10 ms based on LTE-Advanced\(^1\) [NK16]. Associated with the improvements to the hardware of smart phones, the software steadily improved as well, enabling an execution of more sophisticated and more demanding applications. An execution of more sophisticated software on more sophisticated hardware overall increases the energy demands on a battery-operated smart phone. Thus, an execution of a resource-intensive application like a high-end video game still drains the battery of today’s smart phones in few hours (cf. [CH10]). Like for the first iPhone released in 2007, the main limiting factor of a today’s smart phone is still its constrained battery life. An end user typically has to charge its smart phone every day. According to Sekar [Sek13], the power consumption of mobile devices “remains and will remain a first-class design constraint”. Consequently, a battery-operated mobile device like a smart phone requires techniques to minimize its energy consumption without sacrificing the computational power available for an execution of an application.

1.1. Motivation

A corresponding technique for a smart phone arose with the rise of cloud computing that dramatically changed the landscape of Information Technology (IT) over the last years [AFG+10]. Cloud providers like Amazon, Google, or Microsoft offer services like computing infrastructure, platforms, or software to customers in a pay-as-you-go manner (cf. Amazon Web Services, Google Cloud Platform, or Microsoft Azure). Empowered by this new computing paradigm of cloud computing, end users now lease on-demand reliable, elastic, and cost-effective resources from commercial data centers at the cloud. As a result, the technique of Code Offloading emerged both from academic researchers and industrial pioneers, striving to increase the energy efficiency of a (resource-constrained) mobile device and the execution speed of a (resource-intensive) application. Code offloading bridges the gap between Mobile Computing and Cloud Computing with computation offloading via a Wireless Network, called Mobile Cloud Computing (MCC) [SGKB13, uRKOMK14]. In detail, mobile computing comprises

\(^1\)Long-Term-Evolution-Advanced (LTE-Advanced)
(battery-operated) mobile devices like a smart phone, providing a ubiquitous platform with sensing, communicating, and computing capabilities. The rise of cloud computing in the last years brought powerful resources from the cloud closer to mobile computing, typically paid in a pay-as-you-go manner. Last, a mobile device communicates with resources at the cloud via a Wi-Fi or cellular connection (wireless network), providing a high-bandwidth and low-latency link to remote resources.

Due to the combination of mobile computing with cloud computing via a wireless network, code offloading overcomes the energy limitation of a (resource-constrained) mobile device by executing resource-intensive parts from applications on powerful server instances at the cloud. To this end, code offloading regards the cost for an execution of an application part on local resources of a mobile device and on remote resources at the cloud. The local cost includes the execution time and energy consumption for a local execution of an application part on a mobile device. The remote cost includes the execution time, energy consumption, and monetary cost for a remote execution of an application part on a server instance at the cloud. Compared to a local execution, a remote execution takes time and consumes energy on a mobile device for sending information to a remote resource, waiting in idle mode during the execution on a remote resource, and receiving the execution state of a remote resource after the execution. It causes monetary cost for sending bytes to a remote resource, utilizing a remote resource for an execution, and receiving bytes from a remote resource. Regarding the trade-off between the local cost and the remote cost, the challenge is now to find an optimal distribution of application parts between the local resources on a mobile device and the remote resources at the cloud, resulting in a minimum execution time, energy consumption, and monetary cost.

Beside the remote resources at the cloud for code offloading, technological trends like Big Data [JBD+14, Cen12], Cyber-Physical Systems [RLSS10, KM15], and Internet of Things [BLMN13, AFGM+15] made multiple classes of highly distributed heterogeneous resources available for code offloading. As a result, multiple resources augment an execution of an application on a (resource-constrained) mobile device, where a mobile device has to distribute optimally an execution of application parts between multiple resources. For instance, Fog Computing brings the cloud closer to end users, providing powerful, elastic resources to mobile devices at the edge of the network [BMZA12, VRM14]. An exemplary environment for code offloading with multiple resources consists of a smart watch that a Bluetooth link connects to a smart phone. Further, a Wi-Fi link connects the smart phone to a smart car that again an LTE-Advanced link connects to a server instance at the edge. Finally, a fixed link connects the server instance at the edge to a server instance in a data center at the cloud. These
heterogeneous devices in such an environment differ significantly in terms of resources for energy, computation, and communication, increasing typically from very limited (e.g., a smart watch) to virtually unlimited (e.g., the cloud). Code offloading in such environments with multiple resources will become relevant in future, where a plenty of highly distributed heterogeneous resources with different performance characteristics and cost implications surround a (resource-constrained) mobile device.

Summarizing, code offloading can increase the energy efficiency of a mobile device and the execution speed of an application by distributing an execution of application parts from a mobile device to remote resources in the infrastructure, causing monetary cost for a utilization of remote resources. Regarding the whole process of offloading code from a mobile device to a remote resource, an optimal distribution of application parts from a (resource-constrained) mobile device to a or multiple remote resources arises challenging research questions like feasibility, robustness, or adaptability.

1.2. Research Focus

To optimally distribute parts from an application, a realization of an efficient system for code offloading has to address several challenging research questions. The key challenges for a realization of an efficient system directly result from the distributed execution of an application to multiple resources. Accordingly, the whole process of code offloading from the identification of application parts – that can be distributed – to the receive of the result from a remote execution classifies the scope of this dissertation, where prior research does not sufficiently address the key challenges. The following paragraphs give a more detailed view on the research focus of this dissertation that encompasses the whole process for a distribution of an application execution between a mobile device and remote resources.

First of all, code offloading has to identify parts from an application for a remote execution. It divides an application into application parts that a mobile device has to execute locally and into application parts that can be distributed to a remote resource (offloadable). An application part that is offloadable only utilizes the processing capabilities on a mobile device, without accessing further local capabilities like taking a photo via a camera. A good candidate for a remote execution of an application part that is offloadable has a long running computation, utilizing the computing capabilities on the remote resource heavily. Moreover, it has a small size of information required for a remote execution and a small size of information received on a mobile device after a remote execution, utilizing the uplink and downlink from a communication network to a remote resource sparingly. Thus, the arising question is how to (efficiently) identify
application parts that are offloadable, keeping the burden on application developers together with the overhead introduced to a mobile device low.

After the identification of an application part that is offloadable, code offloading has to determine the execution side for this application part. To this end, it regards for the application part the trade-off between the cost for a local execution on a mobile device and the cost for a remote execution on a remote resource, typically formulated as an optimization problem like an Integer Linear Program. The solution of the optimization problem determines whether a mobile device executes the application part that is offloadable locally on its resources or distributes it to a remote resource for a remote execution. Thus, the arising question is how to (efficiently) determine the execution side for an application part identified as offloadable, keeping the time taken for solving the optimization problem minimal.

To solve the optimization problem related to code offloading for an application part that is offloadable, code offloading has to obtain a set of parameters required for the solution of the optimization problem. The required parameters result from the cost – the time taken and energy consumed – for a local execution and – the time taken, energy consumed, and monetary cost caused – for a remote execution. In detail, the parameters comprise information about the application like its execution state, local and remote resources like the performance, and network links like the bandwidth. Thus, the arising question is how to (efficiently) obtain parameters required to solve the optimization problem for code offloading, keeping the overhead introduced to a mobile device, participating resources, and network links low.

After the determination of the execution side for an application part identified as offloadable, code offloading has to execute an application part remotely if a remote execution is more beneficial compared to a local execution. To this end, it sends the execution state gathered locally from a mobile device to a remote resource and waits subsequently for the end of a remote execution on a remote resource, to finally receive the execution state on the remote resource after the remote execution. Due to the utilization of the uplink, the remote resource, and the downlink in case of a remote execution, an end user has to pay a corresponding fee for the utilization charged by (link and resource) providers. Thus, the arising question is how to (efficiently) optimize the process of sending, waiting, and receiving for a remote execution, keeping the introduced overhead together with the charged monetary cost low.

While a remote resource executes an application part on behalf of a mobile device, code offloading has to handle an occurrence of failures on participating nodes or connecting links. For instance, in case of a failure of a remote resource near the end of a remote execution, the occurrence of the failure delays – e.g., due to a node recovery
1. Introduction

and a subsequent re-execution – the receive on a mobile device of the execution state after the remote execution. It gets even worse in case of a long-lasting disconnection between a mobile device and a remote resource near the end of a remote execution, where a local execution of the application part would be retrospectively more beneficial. Thus, the arising question is how to (efficiently) handle an occurrence of failures during a remote execution, keeping the efficiency of code offloading high.

Summarizing, this dissertation tackles a couple of challenging research questions that prior research does not sufficiently address. It regards the whole process of a distribution from an efficient identification to failure handling to a receive of a remote result, proposing different concepts, algorithms, and approaches. In detail, this dissertation realizes an efficient, scalable, and applicable system for code offloading that distributes application parts between a mobile device and multiple remote resources.

1.3. Contributions

The dissertation investigates the challenging research questions described above, contributing concepts, algorithms, and approaches for the investigated questions. We published the contributions of this dissertation at international conferences [BDR14a, BDR14b, BDR15, BDR16] and in an international journal [BDR18], where all publications were peer-reviewed. Concepts, algorithms, and implementations of the approaches presented in [BDR14a, BDR14b, BDR15, BDR16, BDR18] are original work of the author, where a thesis of a student supervised by the author provided preliminary understanding of the work proposed in [BDR15]. At all times, Kurt Rothermel and Frank Dürr helped to improve the presentation of concepts, algorithms, and results, where the author of this thesis contributed about 75 %, 80 %, 85 %, 90 %, and 95 % of the scientific content for the papers [BDR14a], [BDR14b], [BDR15], [BDR16], and [BDR18]. In particular, this dissertation has the following individual contributions:

1. To identify at run-time application parts that are offloadable, this dissertation proposes an annotation-based distribution with least interventions from an application developer as well as end user. An application developer only annotates at development-time application parts that are offloadable, where an offload-aware compiler inserts at compile-time instructions for an application part annotated as offloadable. Based on the offload-specific instructions inserted at compile-time, the approach automatically partitions at run-time an application to parts that have to be executed locally and to parts that are offloadable. Thus, it does not require any interventions from an application developer or end user. In contrast
1.3. Contributions to related approaches in the literature, the annotation-based distribution with offload-specific instructions keeps the burden on application developers and end users low. Moreover, it also keeps the overhead introduced to the run-time system on a (resource-constrained) mobile device low.

2. To determine the execution side for an application part that is offloadable, this dissertation proposes a distribution with the concept of “code bubbling”. Code bubbling shifts the offload decision – determining whether to execute an application part locally or remotely – from a (resource-constrained) mobile device into the infrastructure. Based on the concept of code bubbling, the approach dynamically determines the best available resource for an execution of an application part at run-time, especially in case of multiple remote resources participating in code offloading. In contrast to related approaches in the literature, code bubbling does not require a global view onto all resources of an environment, keeping the overhead introduced to a run-time system on a mobile device low.

3. To obtain from an application the parameters required for the offload decision, this dissertation proposes a bit-flipping approach. The bit-flipping approach marks information from an execution state modified during run-time by flipping a corresponding bit in the information header. Based on the dynamic flipping of a bit at run-time, the bit-flipping approach minimizes the execution state synchronized between a mobile device and a remote resource. It only includes information from an execution state that is modified during run-time. In contrast to related approaches in the literature, the dynamic flipping of a bit does not require a resource-intensive monitoring of information for an execution state in parallel of an execution. As a result, the dynamic flipping of a bit keeps the overhead introduced to a run-time system on a mobile device low.

4. To obtain from an application part the parameters required for the offload decision, this dissertation also proposes an offload-aware Application Programming Interface (API) for the Java programming language beside the dynamic flipping of a bit. The offload-aware Java API extends the Java core class libraries with additional classes for code offloading, making the Java programming language directly aware for offloading code. Based on the additional Java core classes, an application developer only extends a corresponding Java core class, enabling code offloading for its Java application. In contrast to related approaches in the literature, the offload-aware Java API provides an ease of use and an efficient processing of code offloading that keeps the overhead introduced to a run-time
1. **Introduction**

system on a mobile device and on a remote resource low.

5. To *obtain* from participating resources and network links the parameters required for the offload decision, this dissertation proposes the distribution with code bubbling. The distribution with code bubbling makes a decision to offload an application part at the point where the information required for an offload decision occurs, namely at the infrastructure. Due to an offload decision on remote resources at the infrastructure, a (resource-constrained) mobile device only submits an offload request into the infrastructure. Afterwards, it automatically receives offers from participating remote resources, where a remote execution is more beneficial compared to a local execution. In contrast to related approaches in the literature, the distribution with code bubbling does not obtain all parameters required for the offloading decision at a (resource-constrained) mobile device, keeping the overhead introduced to a mobile device, participating remote resource(s), and related network link(s) low.

6. To *optimize* the offloading process of sending, waiting, and receiving in case of a remote execution, this dissertation proposes a caching-aware distribution. It extends a distribution of application parts with a cache on the remote side. The cache on the remote side stores collectively execution states after a local execution or a remote execution of application parts. A mobile device queries the cache for a corresponding entry just before offloading an application part to a remote resource. Based on the cache on the remote side, a mobile device immediately receives a queried execution state in case of a cache hit, whereas in case of a cache miss it starts a distribution of the application part to a remote resource. In contrast to related approaches in the literature, the caching-aware distribution avoids a repeated execution of previously run application parts, keeping the time taken, energy consumed, and monetary cost charged for a remote execution low.

7. To *handle* the occurrence of failures (e.g., node or link failures) during a remote execution, this dissertation proposes a preemptable distribution with safe-point’ing and a predictive distribution with safe-point’ing. The preemptable distribution stores at different points in time the execution state on a remote resource during a remote execution called safe-points. It also transmits these safe-points during a remote execution to a mobile device, enabling an interruption of an offloading process and a continuation of a remote execution on a mobile device in case of a failure. Thus, safe-point’ing does not abandon the intermediate result calculated remotely so far. The predictive distribution extends the preemptable
distribution with an adaptive scheduler that predicts the optimal times for a creation and transmission of a safe-point during a remote execution. For instance, in the optimal case, it only transmits one safe-point just before a disconnection occurs. In contrast to related approaches in the literature, the preemptable distribution based on safe-point'ing increases the robustness of code offloading in case of a failure, keeping the efficiency of code offloading high. Additionally, the predictive distribution based on safe-point’ing ensures a minimal responsiveness and a maximal efficiency of distributed applications despite link failures.

1.4. Structure of the Thesis

Now, this section describes the structure of the dissertation which is as follows:

The current chapter, Chapter 1, shows the ropes and starts with an introduction about resource limitation of smart phones in the past and present time. Afterwards, Section 1.1 gives a motivation for the technique of code offloading that augments a (resource-constrained) mobile device with powerful remote resources. Section 1.2 details about the research focus within code offloading, where Section 1.3 highlights the research contributions of this dissertation, both comprising the whole process of distribution for code offloading. Finally, Section 1.4 outlines the structure of the thesis.

The next chapter, Chapter 2, describes in detail the background information required for this dissertation. It starts with a description of mobile computing in Section 2.1 that communicates with other devices and the Internet via different wireless networks described in Section 2.2. A mobile computing environment accesses via a wireless network computing resources from cloud computing, described in Section 2.3. It also accesses nearby computing resources at the edge of a network, favored by different technological trends described in Section 2.4. The combination of mobile computing with cloud computing enables mobile cloud computing described in Section 2.5, providing services like computation offloading. Last, Section 2.6 presents a summary.

The next chapter, Chapter 3, gives a system overview for code offloading in the landscape of Mobile Cloud Computing (MCC). First, Section 3.1 outlines a related system model before Section 3.2 formulates the problem statement. Related to the system model and problem statement, Section 3.3 describes the system components involved in code offloading. As each component of the system might fail, Section 3.4 highlights the underlying failure model for this dissertation, including crash failures with eventually recovery of nodes and links. Then, Section 3.5 presents the system requirements of a system for code offloading. Last, Section 3.6 presents a summary.

The next chapter, Chapter 4, presents an efficient code offloading with annotations,
1. Introduction

where a (resource-poor) mobile device offloads computation to a (powerful) remote resource. Section 4.1 introduces this basic distribution and Section 4.2 gives its related system overview. Utilizing a runtime environment for code offloading, Section 4.3 describes the functionality of a runtime-layer offloading. Section 4.4 outlines the offloading timeline for a basic distribution and Section 4.5 the functionality provided by an offloading framework. Afterwards, Section 4.6 highlights the implementation of the offloading framework in different prototypes, before Section 4.7 presents the evaluation. Last, Section 4.8 presents a summary.

The next chapter, Chapter 5, presents a robust code offloading through safe-pointing. The preemptable distribution described in Section 5.1 reuses, in case of a failure, intermediate states of a remote execution, without abandoning the remote result achieved so far. To this end, Section 5.2 gives a related system overview, Section 5.3 outlines the offloading timeline for the preemptable distribution, and Section 5.4 describes the functionality provided by an offloading framework. Afterwards, Section 5.5 presents the evaluation of the preemptable distribution in multiple scenarios, where failures occur at different point in times. Last, Section 5.6 presents a summary.

The next chapter, Chapter 6, presents a deadline-aware code offloading with predictive safe-pointing. The predictive distribution described in Section 6.1 utilizes an adaptive algorithm based on prediction models to dynamically adapt the point in times for a creation and transmission of safe-points. To this end, Section 6.2 gives a related system overview before Section 6.3 describes the optimal schedule for safe-pointing that minimizes the number of safe-points received on a mobile device. Afterwards, Section 6.4 presents the evaluation of a MATLAB simulation in different scenarios. Last, Section 6.5 presents a summary.

The next chapter, Chapter 7, presents an optimized code offloading through cooperative caching. The caching-aware distribution described in Section 7.1 optimizes a basic distribution with a cache on the remote side. The cache stores execution states from application parts executed previously. To this end, Section 7.2 gives a related system overview, Section 7.3 outlines the offloading timeline for a caching-aware distribution, and Section 7.4 describes the functionality provided by an offloading framework. To evaluate the overhead and benefits for a caching-aware distribution, Section 7.5 presents the evaluation of an OpenJDK prototype on different mobile devices running different mobile applications. Last, Section 7.6 presents a summary.

The next chapter, Chapter 8, presents a code offloading in environments with multiple tiers. The distribution with code bubbling described in Section 8.1 enables a basic distribution to efficiently offload computation in environments with highly distributed heterogeneous resources. To this end, Section 8.2 gives a related system overview, Sec-
1.4. Structure of the Thesis

Section 8.3 outlines the offloading timeline for a distribution, and Section 8.4 describes the functionality provided by an offloading framework. To evaluate the overhead and benefits for a distribution with code bubbling, Section 8.5 describes the evaluation of a prototype based on Android and OpenJDK. Last, Section 8.6 presents a summary.

The next chapter, Chapter 9, presents the work related to this dissertation. The discussion includes the work related to an efficient code offloading with annotations described in Section 9.1 (cf. the basic distribution in Chapter 4), to a robust code offloading through safe-point’ing described in Section 9.2 (cf. the preemptable distribution in Chapter 5), to a deadline-aware code offloading with predictive safe-point’ing described in Section 9.3 (cf. the predictive distribution in Chapter 6), to an optimized code offloading through cooperative caching described in Section 9.4 (cf. the caching-aware distribution in Chapter 7), and to a code offloading in environments with multiple tiers described in Section 9.5 (cf. the distribution with code bubbling in Chapter 8). Last, Section 9.6 presents a summary.

The last chapter, Chapter 10, presents the conclusion of this dissertation by giving a summary described in Section 10.1 on the topics covered in this dissertation and an outlook described in Section 10.2 on remaining open research questions.
Chapter 2

Background

This chapter presents the background information for code offloading in the scope of Mobile Cloud Computing (MCC). As the name *Mobile Cloud Computing* suggests, MCC combines mobile computing with cloud computing via a wireless network. To this end, Section 2.1 gives a brief overview of the typical environment, classic architecture, and inherent limitations related to mobile computing. Due to the portability of a device from mobile computing, a wireless network described in Section 2.2 connects the mobile computing environment with resources in an intranet or the Internet. Typically connected via a Wi-Fi, cellular, or Bluetooth link, a device from mobile computing can access powerful resources from cloud computing described in Section 2.3 with its essential characteristics, service models, and deployment models. Different technological trends described in Section 2.4 like Big Data, Internet of Things, and Cyber-Physical System bring the resources from a distant cloud closer to a device from mobile computing, providing nearby resources at the edge of a network. The augmentation of mobile computing with nearby and distant resources via a wireless network enables code offloading in the field of Mobile Cloud Computing. To this end, Section 2.5 first gives an overview of MCC, before it describes the related architectures, requirements for a framework, and arising challenges. Finally, Section 2.6 summarizes the main facts from the background information presented in this chapter.

2.1. Mobile Computing

This section gives a brief overview of mobile computing, where devices from mobile computing transmit data via a wireless link according to Koudoumas and Iqbal [KI96]. Improvements in energy-efficient but powerful hardware, wireless telecommunication, and adaptive software enabled this portable computing [Sat10]. To this end, Subsection 2.1.1 characterizes the typical environment of mobile computing, Subsection 2.1.2 the classic architecture of client-server and an extended architecture of client-interlayers-server, and Subsection 2.1.3 the inherent limitations from mobility.
2. Background

2.1. Environment

Mobile computing provides an environment to an end user for sensing, communicating, and processing, where the end user is physically on the move and not stationary in front of a static computer at a single location. The mobile computing environment deploys a portable device with sensing and processing resources that connects via a wireless network to other portable and/or static devices, moving along with a mobile end user. Due to the portability of a computing environment, inherent issues from mobility – e.g., the size, weight, or battery life of the environment – characterize the hardware for a mobile computing environment. For instance, the small size of a mobile computing environment requires an alternate interface for user Input/Output (user IO) like speech recognition compared to classic user IO of static computers. [Sat10, TAY10]

A mobile end user of a mobile computing environment collects, processes, stores, accesses, and spreads information anywhere and anytime via a communication network, avoiding the spatial and temporal constraint from static computers like a desktop computer. This (spatial and temporal) flexibility enables the vision of “information at your fingertips anywhere, anytime” [Sat10], driving academic researchers and industrial pioneers in the past, present, and future. To perform anywhere at anytime a task like spreading information via a mobile computing environment, a corresponding device is a condensed modification of multipurpose computers, where a battery powers the downgraded hardware of a mobile device. Owing to the mobility, a mobile device communicates via a wireless communication link at a low bandwidth and high latency compared to a wired communication link of a static computer. [Sat10, TAY10]

The availability of computation and connectivity irrespective of time and place quickly raised the popularity and importance of devices from mobile computing in today’s mobile systems of networked computing. Furthermore, sensing, communicating, and computing capabilities of devices from mobile computing like wireless sensors, Personal Digital Assistants (PDAs), cellular phones, laptops, and many others are nowadays seamlessly integrated into the world of everyday life. Being not distinguishable from physical objects in the world, people accomplish nowadays ordinary tasks by unconsciously using ubiquitous computers – e.g., compare Weiser’s seminal paper [Wei99] on the vision of ubiquitous computing. [Sat10, TAY10]

2.1.2. Architecture

The typical architecture in mobile computing is a hierarchy with two tiers, where a transient infrastructure consists of a “client” (first tier) and a “server” (second tier) – the classic model of client-server. To indicate the presence of resources from multiple
servers, nowadays “cloud” is also a name for the second tier (cf. Section 2.3). The first tier, a client, of the two-tier hierarchy corresponds to a mobile computing environment described above, limited by the mobility concerns like downgraded hardware or limited battery life. The second tier, a server or the cloud, of the two-tier hierarchy corresponds to a well-managed, stationary environment, being free of mobility concerns. This classic model of client-server or client-cloud exists from the very beginning of mobile computing. Nowadays, future architectures extend this two-tier hierarchy for mobile computing with one or more interlayers between a client and a server (cf. Section 2.4). For instance, Satyanarayanan et al. [SBCD09] propose an intermediate layer called “cloudlet” as a remote resource at, for instance, a Wi-Fi access point. Compared to resources in a distant cloud, it provides a lower latency and higher bandwidth of a network link, offering its resources for processing and storage to nearby devices. [Sat10]

2.1.3. Limitations

The mobility of mobile computing – not being spatial and temporal constraint – is also responsible for its intrinsic limitations, where technological issues are not bearing the blame. Due to the mobility of a device from mobile computing, it has poor, vulnerable, and variable resources compared to static resources from multipurpose computers or the cloud. The device’s mobility defines its acceptable weight and compact size, also affecting the performance from sensing, communicating, and computing resources like processor speed or memory size. Additionally, end users always want to have a smaller, lighter, faster, longer-running mobile device. Despite past or future progress and improvement to mobile devices in absolute ability\(^1\), performance on mobile devices were, are, and will be always a compromise, making a mobile device resource-poor in relation to static resources. Especially as a limited source of energy primarily powers the hardware and software of a mobile device\(^2\), where the battery technology evolves and improves at a slow pace. Thus, energy concerns accompany the flexibility of mobile computing, limiting its usage to the charge cycles of the limited source of energy. Associated with the mobility of a device, the mobile device itself and the communication from or to a mobile device via a wireless link are exposed to a higher danger for loss, damage, or attack in relation to static resources, for instance, locked in the office and communicating via a wired link. The quality of a wireless link – especially while

\(^1\)For instance, the iPhone (1\textsuperscript{st} generation) executes applications on a Samsung 32-bit RISC ARM processor running at 620 MHz with a 128 MB memory. The Samsung Galaxy S7 executes applications on the Samsung Exynos 8 Octa 8890 System-on-a-Chip (SoC) with a big.LITTLE 64-bit architecture of eight cores (maximum frequency: 2.6 GHz) and 4 GB memory. [Sam16]

\(^2\)The iPhone (1\textsuperscript{st} generation) has a battery capacity of 1400 mAh, whereas the Samsung Galaxy S7 has a battery capacity of 3000 mAh.
2. Background

on the move – is also highly variable in relation to static resources. For instance, a Wi-Fi network within a building provides a reliable, high-bandwidth and low-latency connection, whereas a cellular network provides an unreliable, low-bandwidth and high-latency connection, suffering under transient outages. [Sat96]

2.2. Wireless Network

To provide information anywhere at anytime for ubiquitous computing, communicating via a wireless link with other mobile or static devices in an intranet or the Internet is a key characteristic of mobile computing. As a result, properties like coverage, data rates, or latency from a wireless link of a communication network impact significantly the performance of mobile computing and thus, its acceptance. Due to the rapid improvements and vast number of different technologies available in mobile computing for a wireless communication network, this section gives a brief overview on the network technologies utilized in this dissertation. In detail, this dissertation utilizes Wireless Fidelity networks described in Subsection 2.2.1, cellular networks described in Subsection 2.2.2, and Bluetooth networks described in Subsection 2.2.3.

2.2.1. Wireless Fidelity Network

Wireless Fidelity (Wi-Fi) is a brand name for a family of standards that implements a Wireless Local Area Network (WLAN) communication. It specifies techniques for Media Access Control (MAC) and physical layer, administrated by the Institute of Electrical and Electronics Engineers (IEEE) LAN/MAN Standards Committee (IEEE 802) in IEEE 802.11. IEEE 802.11 is a mobile communication network via a wireless link. It connects two or more mobile devices – also with the wider Internet – in a limited area like a building, where an end user can walk around and stay connected as long as being in the local coverage of a Wi-Fi Access Point (AP). Typically, the coverage of a Wi-Fi AP depends on different properties like radio frequency, output power, or receiver sensitivity. IEEE released the first version of IEEE 802.11 in the year 1997, improved and extended by multiple amendments over the years (cf. [Soc12]). The following paragraphs describe the amendments utilized in this dissertation.

802.11-1997: In the year 1997, IEEE released the standard 802.11-1997 that provides a raw data rate of 1 Mbit/s or 2 Mbit/s with a local coverage of approximately

3For a cellular link, the iPhone (1st generation) offers a peak data rate of 220 kbit/s [Inf07c], whereas the Samsung Galaxy S7 offers a peak data rate of 150 Mbit/s (uplink) and of 600 Mbit/s (downlink) with a latency less than 10 ms owing to LTE-Advanced [NK16, Sam16].
2.2. Wireless Network

20 m indoor and 100 m outdoor. It transmits signals either via infrared or in the ISM\textsuperscript{4} frequency band of 2.4 GHz – like other network technologies as Bluetooth (cf. Subsection 2.2.3) causing interference – with a frequency bandwidth of 22 MHz. [Soc12]

802.11b: In the year 1999, IEEE released the standard 802.11b that extends the modulation method of 802.11-1997. It increases the raw data rate to up to 11 Mbit/s with a local coverage of approximately 35 m indoor and 140 m outdoor. [Soc12]

802.11a: In the year 1999, IEEE also released the standard 802.11a that utilizes the same protocols on the data link layer (Layer 2\textsuperscript{5}) and frame format as 802.11-1997, but changes the modulation technique on the physical layer (Layer 1\textsuperscript{5}). It operates in the relatively unused frequency band of 5.8 GHz with a bandwidth of 20 MHz, enabling raw data rates of up to 54 Mbit/s. Due to the higher carrier frequency, a solid object like a wall absorbs the signal strongly. As a result, it lowers the data rates and local coverage, being approximately 35 m indoor and 120 m outdoor. [Soc12]

802.11g: In the year 2003, IEEE released the standard 802.11g that improves the modulation technique of 802.11b (2.4 GHz), enabling a raw data rate of up to 54 Mbit/s and a local coverage of approximately 38 m indoor and 140 m outdoor. [Soc12]

802.11n: In the year 2009, IEEE released the standard 802.11n that enhances the previous standards of 802.11a and 802.11g with the extensions of frame aggregation on the data link layer, 4x4 Multiple Input Multiple Output (MIMO) transmission on the physical layer, and security improvements. It operates in both frequency bands of 2.4 GHz or 5.8 GHz with a bandwidth of 20 MHz or 40 MHz. The maximum raw data rates with a frequency bandwidth of 40 MHz is 600 Mbit/s with a local coverage for both frequency bands of approximately 70 m indoor and 250 m outdoor. [Soc12]

2.2.2. Cellular Network

A cellular network provides a wireless link for mobile devices, usually available within a huge range provided by (commercial) cell towers. Developed in the early 1980s, the first cellular network only transmitted analog signals for voice calls, whereas nowadays it supports IP\textsuperscript{6}-based connectivity with a high bandwidth, low latency, and extensive

\textsuperscript{4}Industrial, Scientific and Medical (ISM)
\textsuperscript{5}Corresponding to the model from International Organization for Standardization (ISO)/Open Systems Interconnection (OSI)
\textsuperscript{6}Internet Protocol (IP)
2. Background

coverage. Thus, communication ubiquity for mobile computing comes nowadays true, being available anywhere and anytime. The following paragraphs give a brief overview for the evolution of cellular networks utilized in this dissertation.

First Generation: The 1st generation (1G) of cellular network is only an analog system for voice services to cellular phones, where the underlying architecture is a circuit-switched network. In detail, Bell Labs developed the system standard of Advanced Mobile Phone Service (AMPS) for analog mobile cellular phones [Ehr79, You79] – deployed in the year 1983. In the spectrum range of 869 - 894 MHz, cell sites transmit voice to mobiles (forward channel), whereas in the spectrum range of 824 - 849 MHz mobiles transmit voice to cell sites (reverse channel) [Com81]. The main limitations are its low data rates of about 2 kbit/s and limited services. [PSMM04, Akh09, KS11, KSK14]

Second Generation: The 2nd generation (2G) of cellular network introduces the wireless technology of Global Systems for Mobile Communications (GSM) in the year 1991, still used today in many countries all over the world. The main focus of GSM was the transmission of voice and data with digital signals – still based on a circuit-switched architecture. It provides higher spectrum efficiency, better data services, and a seamless roaming in different countries compared to 1G. In detail, next generations of cellular network are only an evolution to the architecture of GSM. GSM – more precisely P-GSM – technically uses the frequency band of 890-915 MHz for the reverse channel and of 935-960 MHz for the forward channel. Each channel is segmented into 124 carrier frequencies of 200 kHz and each carrier frequency is further fragmented into eight time slots. As a result, GSM enables eight voice or data calls at the same time, increasing the capacity of the network. The data rate for each channel increased to 9.6 kbit/s – and with an improved channel encoding to 14.4 kbit/s. The main limitations of 2G are a lack of security and its low data rate for web browsing. [PSMM04, Akh09, KSK14]

The 2.5th generation introduces different techniques like High Speed Circuit Switched Data (HSCSD) or General Packet Radio Services (GPRS) in the year 1995. HSCSD enhances circuit-switched data by improving the encoding methods and aggregating multiple time slots for data transmission. Theoretically, the data rate increases to 115.2 kbit/s with eight combined channels. By providing circuit-switched data and packet-switched data, GPRS enables simultaneously a phone call and data transmission, further increasing the data rate to theoretically 171.2 kbit/s.

The 2.75th generation changes the modulation scheme in the year 1999, called Enhanced Data rates for Global Evolution (EDGE)\(^7\). It further increases the data rates per

\(^7\)The original name was Enhanced Data rates for GSM Evolution
time slot with an unchanged baud rate (sending more bits per symbol) to 59.2 kbit/s. Realistically, a mobile phone gets maximally four time slots for the downlink and two time slots for the uplink, resulting in 263.8 kbit/s and 118.4 kbit/s, respectively.

**Third Generation:** The 3rd generation (3G) of cellular network divides the core network into a circuit-switched part and packet-switched part in the year 2000. It enables IP-based services, higher data rates, increased system capacity, and worldwide roaming. 3G is a set of international-accepted standards, where ITU\(^8\) defined the requirements of 3G in IMT-2000\(^9\). The 3GPP\(^10\) organization developed Universal Terrestrial Mobile System (UMTS) as a European variant. It utilizes the frequency bands of 800/900 MHz, 1.7 - 1.9 GHz, and 2.5 - 2.69 GHz with a carrier frequency per channel of 5 MHz. UMTS achieves a data rate of 144 kbit/s in rural regions, 384 kbit/s in suburban regions, and 2 Mbit/s in low range outdoor. The main limitations of 3G are its still lower data rates compared to Wi-Fi at this time (cf. Subsection 2.2.1), high latency of at least 30 ms, high complexity of the terminals, and high cost of the infrastructure. [DA97, ITU97, PSMM04, Akh09, KSK14]

The 3.5th generation improves 3G with lower latency, higher data rates, and extended capacity by applying the protocols High-Speed Downlink Packet Access (HSDPA) and High-Speed Uplink Packet Access (HSUPA) combined in High-Speed Packet Access (HSPA). The key features are, for instance, shared channel and multi-code transmission, higher-order modulation, and fast link adaptation and scheduling. The maximum channel rate for HSDPA is 14.4 Mbit/s and for HSUPA 5.8 Mbit/s. HSPA evolution (HSPA+) further increases the data rates to 21 Mbit/s for the downlink and 11 Mbit/s for the uplink. The subsequent improvements like a combination of higher order modulation with spatial multiplexing enable a theoretical peak data rate for the downlink of 672 Mbit/s and for the uplink of 168 Mbit/s. [fG00]

### 2.2.3. Bluetooth Network

Bluetooth is a packet-based protocol that implements a Wireless Personal Area Network (WPAN) communication with a master-slave structure over a short range based on low-power consumption, invented by Ericsson in the year 1994. The Bluetooth Special Interest Group specifies nowadays the Bluetooth standards, where IEEE specified a Bluetooth standard in IEEE 802.15.1, not maintaining it anymore. Bluetooth transmits signals in the ISM frequency band of 2.4 GHz with a bandwidth of 1 MHz. The

---

\(^8\)International Telecommunication Union (ITU)
\(^9\)International Mobile Telecommunications-2000 (IMT-2000)
\(^10\)3rd Generation Partnership Project (3GPP)
2. Background

range of a Bluetooth communication depends on a power-class, where Class 1 has a maximum permitted power of 100 mW and a typical range of 100 m, Class 2 of 2.5 mW and 10 m, Class 3 of 1 mW and 1 m, and Class 4 of 0.5 mW and 0.5 m.

**Bluetooth 1:** The Bluetooth versions 1.0, 1.1\(^{11}\), and 1.2\(^{11}\) improve step-wise the new standard by fixing errors and introducing, for instance, non-encrypted channels or faster discovery and connection, having a raw data rate of up to 1 Mbit/s. [Soc05]

**Bluetooth 2:** The Bluetooth versions 2.0 released in the year 2004 and 2.1 released in the year 2007 introduce further improvements like Secure Simple Pairing and utilizes a combined modulation, enabling raw data rates of up to 3 Mbit/s.

**Bluetooth 3:** The Bluetooth version 3.0 released in the year 2009 has the main key feature of Alternative MAC/PHY. It enables raw data rates of up to 24 Mbit/s, where the Bluetooth radio discovers a device and initiates a connection and a parallel 802.11 radio transports the data at a high speed.

2.3. Cloud Computing

The rise of cloud computing started in the early 2000s, where, for instance, Amazon announced its Elastic Compute Cloud in the year 2006 [Ama06]. Four years later, Microsoft also announced the general availability of its cloud computing platform, called Azure [Hau10]. In the late 2000s, cloud computing experienced a further increase, mainly based on the availability of high-capacity networks, low-cost devices, virtualization techniques, and service-oriented architectures. The National Institute of Standards and Technology (NIST) defines cloud computing as a model that offers to end-users “… on-demand network access to a shared pool of configurable computing resources […] rapidly provisioned and released …” [BGPCV12], where configurable resources include resources like storage, processing, or network. To this end, Subsection 2.3.1 outlines the essential characteristics, Subsection 2.3.2 the different service models, and Subsection 2.3.3 the different deployment models of cloud computing.

2.3.1. Essential Characteristics

Following the definition from NIST, an end user utilizes cloud computing as an on-demand self-service, where he or she autonomously allocates and releases resources

\(^{11}\) IEEE specified version 1.1 as IEEE 802.15.1-2002 and version 1.2 as IEEE 802.15.1-2005
2.3. Cloud Computing

from a shared pool of configurable resources without an intervention from the cloud provider. An end user allocates and releases capabilities from a cloud provider via a network access, utilizing its client platform like a mobile phone or workstation. A cloud provider uses a multi-tenant model and pools its computing resources with dynamic re-/assignment of physical and virtual resources to serve the demands from multiple end users. Typically, an end user does not know the exact location of its allocated resources (location independence), defining only the resource’s location at a higher level of abstraction like a region. An allocation and release of resources from a cloud provider take place rapidly and elastically, scaling automatically up and down based on the user demand. For an end user, available resources seem to be virtually unlimited, provided in any quantity at any time. A system of cloud computing automatically controls, meters, and optimizes the usage of its resources, providing transparency to its provider and to end users. A cloud provider typically charges fees for utilized resources based on a “pay-as-you-go” model. [AFG+10, BGPCV12]

2.3.2. Service Models

Cloud computing offers different service models, where common service models are Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS), increasing step-wise the level of abstraction. The service model IaaS provides to an end user the “plain” resource like processing, storage, or network. An end user can run arbitrary software on the resources like an operating system on a processing resource. To this end, an end user has the highest flexibility on the total software stack associated with the responsibility to manage the total software stack. The service model PaaS provides to an end user the ability to deploy software onto the infrastructure from cloud computing, where a cloud provider supports a set of programming languages and tools. To this end, an end user does not manage the software stack provided for software development but controls the environment for deployment as well as the deployed software. Last, the service model SaaS provides to an end user the utilization of applications supported by a cloud provider, where the supported applications are accessible from various interfaces like a Web browser. To this end, an end user does not manage the total software stack required to execute an application and just utilizes an application as and when required. [AFG+10, BGPCV12]

2.3.3. Deployment Models

Cloud computing offers different deployment models, where common deployment models are a private cloud, a community cloud, a public cloud, and a hybrid cloud. The
2. Background

deployment model of a private cloud provides an infrastructure exclusively used by a single organization, where an organization, a third-party, or a combination owns, operates, manages, and/or hosts the cloud infrastructure internally or externally. The deployment model of a community cloud provides an infrastructure exclusively used by a specific community from organizations – e.g., sharing a common concern, where one or more organization(s) from a community, a third party, or a combination owns, operates, manages, and/or hosts the cloud infrastructure internally or externally. The deployment model of a public cloud provides an infrastructure for the general public, where a company owns, operates, manages, and hosts the cloud infrastructure internally. The deployment model of a hybrid cloud bounds together two or more different infrastructures from a private cloud, community cloud, and/or public cloud that remain unique but provide portability. [AFG+10, BGPCV12]

2.4. Technological Trends

Technological trends like Big Data, Internet of Things (IoT), or Cyber-Physical System (CPS) pave the way for powerful computing resources at the edge of a network, to rapidly manage and process a large amount of data. For instance, computing resources from cloud computing (cf. Section 2.3) reside in distant data centers at a few locations all over the world, being too far-away for related applications.

Commonly, the four dimensions of volume, velocity, variety, and veracity distinguish solutions for Big Data from conventional IT. Big Data has to manage, process, and validate (veracity) rapidly (velocity) a large amount (volume) of structured and unstructured data (variety). Jewell et al. characterize in [JBD+14] solutions for Big Data as a real-time complex processing. Thus, the utilization of computing resources at a distant data center from cloud computing is not always preferable, requiring in-network computing resources where data arises. [Cen12, JBD+14]

IoT includes a wide variety of physical objects from everyday life, integrating the physical world into computer-assisted systems. Augmented by sensors, actuators, and connectivity, an everyday object becomes a so-called “smart object” that autonomously collects, exchanges, and retrieves data with other smart objects and the Internet. This advanced interconnection improves the efficiency, accuracy, and economic benefit from everyday objects, generating a large amount of data at an exponential rate. This generation of a large amount of data introduces new challenges like the demand of very low end-to-end latency and very high network bandwidth. Thus, it requires distant computing resources from cloud computing together with in-network computing resources where data arises. [BLMN13, AFGM+15, CZ16]
2.5. Mobile Cloud Computing

With the embedding of sensing, communicating, and processing capabilities to everyday objects in the physical world, CPS bridges the cyber-world with the physical world by monitoring and controlling the physical world with a computer-assisted system. A CPS influences the interaction and control of the physical world by humans, coupling embedded real-time systems of distributed sensors and actuators with controls. Thus, it requires the distant computing resources from cloud computing together with in-network computing resources. [RLSS10, KM15]

Regarding the technological trends of Big Data, IoT, and CPS, the requirement for closer computing resources at the edge of a network is omnipresent. For instance, fog computing offers powerful computing resources like processing or storage at the edge of a network [BMZA12]. It rapidly processes data where it arises due to its location in the near proximity to end devices and its dense geographical distribution. Compared to distant computing resources from cloud computing (cf. Section 2.3), fog computing connects an end device with its in-network resources via a link with a higher bandwidth and a lower latency. [BMZA12, VRM14, YLL15]

2.5. Mobile Cloud Computing

Mobile Cloud Computing (MCC) integrates the powerful computing resources from cloud computing (cf. Section 2.3) into the environment of mobile computing (cf. Section 2.1) via wireless networks (cf. Section 2.2) to overcome the limitations of a (resource-poor) mobile device (cf. Subsection 2.1.3). The service of code offloading from MCC is a potential technique that augments (resource-poor) devices from mobile computing with powerful, remote resources from cloud computing. The idea of code offloading is to execute resource-intensive parts from an application on a powerful, remote resource on behalf of a mobile device. The related benefits for a mobile device are shorter execution times and lower energy consumption for an application execution, paying a monetary cost for the utilization of the remote resource. To this end, Subsection 2.5.1 gives a brief overview of code offloading within MCC before Subsection 2.5.2 characterizes its typical architectures of two tiers and multiple tiers. Afterwards, Subsection 2.5.3 discusses requirements to a framework for code offloading before Subsection 2.5.4 outlines related challenges.

2.5.1. Overview

The driving force behind an offloading of code from a mobile device to a powerful, remote resource is the resource poverty of mobile devices. The hardware of a mobile
2. Background

![Diagram of a typical two-tier architecture from Mobile Cloud Computing](image)

Figure 2.1.: A typical two-tier architecture from Mobile Cloud Computing, where a smart phone communicates via the Internet with the cloud.

device is unavoidably resource-poor in relation to static workstations or server instances (cf. Subsection 2.1.3). Cloud services offer on-demand computing resources with a high performance, availability (24/7), scalability, elasticity, and accessibility (anywhere at anytime) at a low cost (cf. Section 2.3). By offloading computation from a resource-poor device to remote resources, the advantages for a mobile device are, for instance, a reduced consumption of power increasing its battery life. However, the mobility in mobile computing blurs the classic model of client-server (cf. Subsection 2.1.2), where, for instance, a client statically executes predefined parts of an application on a well-known remote server based on Remote Procedure Calls (RPCs). Due to the mobility in mobile computing, the current situation on a client about user, device, application, network, and environment changes constantly. Thus, a client requires a more dynamic system for a distributed execution of application parts. Dependent on information from the current situation, the system should adaptively react and reassign application parts during the execution to the local resources of the client and to a remote resource like a server instance at a nearby fog or a distant cloud to fit the current situation. Especially, uncertain connectivity from a wireless network like a cellular network (cf. Subsection 2.2.2) requires high adaptation, reacting rapidly on its high-varying performance and reliability. [Sat96, Sat10, TAY10]

2.5.2. Architectures

In general, code offloading supports manifold architectures, where the most popular architecture consists of two tiers (cf. Figure 2.1): A mobile device (first tier) and the cloud (second tier). Categorized by the number of tiers within the architecture, this kind of architecture is called a two-tier architecture. Beside a two-tier architecture, there are also multiple tiers possible in today’s MCC (cf. Figure 2.2), where multiple tiers surround a mobile device and participate in code offloading like smart wearable
2.5. Mobile Cloud Computing

Figure 2.2.: A multi-tier architecture from Mobile Cloud Computing, where multiple resources like smart wearables, smart phones, ..., and server instances at the cloud surround a smart phone, communicating via different networks with each other. (first tier), smart phone (second tier), ..., and the cloud (last tier). As a result, this kind of architecture is called a multi-tier architecture.

**Two-tier Architecture:** Figure 2.1 shows the most popular architecture with two tiers from Mobile Cloud Computing. It consists of a mobile device like a smart phone (first tier), wireless and fixed networks, and remote resources like server instances at the cloud (second tier). In detail, a mobile device communicates via a wireless network with the Internet and thus, with remote resources located in a data center at the cloud. A wireless network is either a Wi-Fi network or a cellular network from a network provider that peers its wireless network with the Internet at Internet exchange nodes like the DE-CIX in Frankfurt\(^\text{12}\), providing connectivity to the infrastructures of cloud providers. The network within the Internet and to the cloud infrastructures is typically a fixed network, providing high bandwidth, low latency, and high reliability.

\(^{12}\text{https://www.de-cix.net/en/locations/germany/frankfurt} \)
2. Background

**Multi-tier Architecture:** Figure 2.2 shows an architecture with multiple tiers from Mobile Cloud Computing. It consists of multiple classes of highly distributed heterogeneous resources like smart wearables, smart phones, tablets, smart cars, workstations, server instances at the fog, and server instances at the cloud. In detail, different kind of (mobile and fixed) networks like a Bluetooth network, a Wi-Fi network, a cellular network, or the Internet connect the multiple resources with each other. Thus, multiple resources surround a mobile device like a smart phone, differing significantly in terms of capabilities like energy, computation, and communication. The capabilities from the multiple resources increase typically from very limited (cf. smart wearable) to virtually unlimited (cf. server instances at the cloud). An exemplary scenario for an architecture with multiple tiers consists of a smart watch that a Bluetooth link connects to a smart phone. The smart phone again communicates via a Wi-Fi link to a resource on a smart car that provides connectivity via a 3G mobile communication to an in-network server at the edge of the network. The 3G mobile communication also provides connectivity to a server instance at the cloud, where the network from the peering node to the cloud is a fixed network.

**2.5.3. Framework**

To provide code offloading for an architecture with two or multiple tiers, participating resources like the mobile device and the remote resource require an efficient framework that enables a dynamic distribution of application parts.

For the mobile device, a framework has to answer the following Five Ws and one H: What does it offload, who does the offloading, when does the offloading happen, where does it offload to, why does it offload, and how to offload. The question *what does it offload* defines the type of an application part like a thread or a method. The question *who does the offloading* defines the layer within the software stack of a mobile device that is responsible for the offloading like at application layer, at run-time layer, or at system layer. The question *when does the offloading happen* defines the point in time of the decision making like at development-time or at run-time. The question *where does it offload to* defines the number of remote resources that are available for code offloading like resources at the cloud, the fog, and/or further resources from smart devices. The question *why does it offload* defines the decision problem of code offloading like increasing energy efficiency and/or execution speed and/or decreasing monetary cost. The question *how to offload* defines the behavior of a mobile device during code offloading like an execution of a mechanism to handle failures.

For the remote resource, a framework has to answer the following Five Ws and one H:
What does it execute, who does the execution, when does the execution happen, where does it execute on, why does it execute, and how to execute. The question *what does it execute* defines the software stack on a remote resource like a virtualization environment, a mobile operating system, or a lightweight runtime environment. The question *who does the execution* defines the re-distribution from a remote resource to another remote resource or to the client due to, for instance, a long-lasting disconnection. The question *when does the execution happen* defines the point in time of the start for an execution on a remote resource like for an execution of time-uncritical parts. The question *where does it execute on* defines the number of remote executions for code offloading like a parallel execution of an application part. The question *why does it execute* defines the monetary cost charged for an execution on a remote resource like a “pay-as-you-go” model from cloud computing. The question *how to execute* defines the performance of a remote resource like the different kind of virtual machines offered in today’s cloud computing.

### 2.5.4. Challenges

Combining the two different fields of mobile computing and cloud computing, code offloading from MCC faces a number of challenges for a distribution of application parts between a mobile device and a remote resource. Due to the resource poverty of a mobile device like a smart phone, code offloading should sparingly use resources like the limited resources of a mobile device, keeping the overhead introduced to the system low. For instance, a resource-efficient monitoring of information related to the current situation is essential for an effective distribution of an application. Due to an uncertain connectivity of a wireless network like high-varying bandwidth, latency, and reliability, code offloading should provide a seamless connectivity to remote resources, handling dynamically network failures like disconnections. Regarding the low bandwidth and high latency of a Wide Area Network (WAN) link compared to, for instance, a Wi-Fi link, code offloading should utilize computing resources not only from distant data centers at the cloud but also from nearby in-network data centers at the fog. Even more, code offloading should also utilize closer computing resources from static workstations or smart devices like a smart car. Due to the advantages from cloud computing or fog computing like elasticity and scalability, code offloading should adapt dynamically the usage of computing resources at the cloud or at the fog to current workload by an adaptive provisioning and de-provisioning of computing resources. Moreover, code offloading should also turn its attention to challenges inherited from cloud computing and fog computing like challenges related to trust, security, or privacy, because a remote
2. Background

Execution of application parts from an end user can contain user-related information. Summarizing, a resource-efficient situation-aware system for code offloading is essential to face the outlined challenges.

2.6. Summary

Mobile computing described in Section 2.1 provides a portable environment that possesses capabilities for sensing, processing, and communicating of data, where a transmission of data happens via a wireless link. A mobile computing environment enables its end user to have information at its fingertips, anywhere at anytime, accessing data in the Internet mainly on the classic architecture of client-server. The issue of “mobility” in mobile computing, however, also limits the usage of a mobile computing environment, restricted by its lower performance like the limited battery life or the varying wireless connectivity compared to a static workstation. To weaken the limitations of a mobile computing environment, steady improvements to its hardware and communication like the underlying technology of a mobile wireless network increased its performance significantly. The current versions of wireless networks described in Section 2.2 provide a wireless link from a Wi-Fi network, a cellular network, or a Bluetooth network with a high performance like a high bandwidth, a low latency, a high reliability, and a high coverage, enabling the communication ubiquity. Based on this communication ubiquity, a mobile computing environment utilizes the field of cloud computing described in Section 2.3 that provides powerful services in infrastructures at a few locations all over the world. The essential characteristics of cloud computing are an on-demand provisioning of powerful, virtually unlimited computing resources like processing or storage, automatically charged in a “pay-as-you-go” manner. The utilization of computing resources from cloud computing happens typically based on different service models like IaaS, PaaS, and SaaS and deployment models like private cloud, community cloud, public cloud, or hybrid cloud. Beside computing resources in a distant cloud infrastructure, technological trends like Big Data, IoT, and CPS described in Section 2.4 moved the distant cloud closer to end terminals, offering nearby computing resources at the edge of a network like fog computing. The integration of cloud computing – and, for instance, fog computing – into mobile computing created the field of mobile cloud computing described in Section 2.5, providing new areas of application like code offloading. Code offloading augments a resource-poor device from mobile computing with powerful, remote resources at the fog or at the cloud. It dynamically executes resource-intensive parts from an application on remote resources dependent on the current situation of the user, device, application, network, and envi-
2.6. Summary

Due to the vast number of available resources for code offloading in today’s landscape of MCC, different architectures are common like a two-tier architecture or a multi-tier architecture. To this end, a related framework for code offloading between two or multiple tiers arises a number of challenges like resource efficiency or situation awareness, requiring a resource-efficient situation-aware system to offload code.
Chapter 3
System Overview

This chapter presents code offloading in the landscape of Mobile Cloud Computing. In general, code offloading augments a (resource-poor) mobile device with (powerful) remote resources by executing (resource-intensive) parts of an application remotely. To this end, Section 3.1 describes the underlying system model for code offloading with multiple tiers before Section 3.2 outlines the general problem statement to be solved for offloading code. Based on the system model and the problem statement, Section 3.3 outlines the system components involved in code offloading, including an offloading client, an offloading service, and a communication network. As each system component can fail at any point in time, Section 3.4 describes the underlying failure model for this dissertation and Section 3.5 the key requirements for an offloading system. Finally, Section 3.6 summarizes the main facts of code offloading presented in this chapter.

3.1. System Model

Today’s landscape of Mobile Cloud Computing comprises multiple classes of highly distributed heterogeneous resources that differ in performance characteristics and cost implications (cf. Subsection 2.5.2 and especially Figure 2.2). To this end, this dissertation formulates a general system model for code offloading based on multiple resources (multi-tier architecture). The general system model for a multi-tier architecture also includes the system model for the classic two-tier architecture presented in Subsection 2.5.2. The general system model is as follows:

Multiple highly distributed heterogeneous resources are available for code offloading, where each resource $\Xi(\xi)$ with $\xi = 0, 1, \ldots, n_\xi - 1$ is capable of executing portable code (p-code) with the help of a runtime environment. The performance characteristic $P_{\text{ppwr}}(\xi)$ is the processing power of a resource $\Xi(\xi)$, indicating how many p-code (e.g., instructions) the resource executes per second.

Utilizing other resources, the performance characteristics $E_{\text{send}}(\xi)$, $E_{\text{wait}}(\xi)$, $E_{\text{recv}}(\xi)$, and $E_{\text{exec}}(\xi)$ for a resource $\Xi(\xi_i)$ are energy factors of sending bytes, waiting in idle mode,
3. System Overview

receiving bytes, and executing p-code, indicating how much energy the operation consumes per second. Providing its computing power to other resources, the cost implication \( C_{\text{exec}}(\xi_r) \) for a resource \( \Xi(\xi_r) \) is the monetary cost of a resource, indicating how much monetary cost a resource charges per executed p-code.

A resource \( \Xi(\xi_i) \) and a resource \( \Xi(\xi_r) \) communicate with each other via a bidirectional link \( \Lambda(\xi_i; \xi_r) \). For a link \( \Lambda(\xi_i; \xi_r) \), the performance characteristics \( B_{up}(\xi_i; \xi_r) \) and \( B_{down}(\xi_i; \xi_r) \) are the up and down bandwidth of a link, indicating how many bytes a link transmits per second. The performance characteristics \( L_{up}(\xi_i; \xi_r) \) and \( L_{down}(\xi_i; \xi_r) \) are the up and down latency of a link, indicating the network delay of a link in seconds. The cost implications \( C_{\text{send}}(\Lambda(\xi_i; \xi_r)) \) and \( C_{\text{recv}}(\Lambda(\xi_i; \xi_r)) \) are the monetary cost of a link for sending and receiving bytes, indicating how much monetary cost a link charges per sent or received byte.

A mobile application \( A \) consists of application parts \( A_\alpha \) with \( \alpha = 0, 1, \ldots, n_\alpha - 1 \) like methods. To offload an application part \( A_\alpha \) from a resource \( \Xi(\xi_i) \) to a resource \( \Xi(\xi_r) \), the execution of the application part \( A_\alpha \) must only rely on the processing power of resources, not accessing further capabilities from a resource like a local sensor. Thus, the application parts \( A_\alpha \) consist of application parts \( A_{\alpha_0} \) that can be offloaded ("offloadable") and application parts \( A_{\alpha_1} \) that must be executed locally.

Each application part \( A_\alpha \) comprises an input execution state \( I_{\text{state}}(A_\alpha) \) like method parameters, portable code \( P_{\text{code}}(A_\alpha) \) like method instructions, and an output execution state \( O_{\text{state}}(A_\alpha) \) like the return value of the method. An execution of \( P_{\text{code}}(A_\alpha) \) only depends on \( I_{\text{state}}(A_\alpha) \) and transforms \( I_{\text{state}}(A_\alpha) \) into \( O_{\text{state}}(A_\alpha) \). The load metric \( P_{\text{exe}}(I_{\text{state}}(A_\alpha), P_{\text{code}}(A_\alpha)) \) indicates the number and kind of p-code a resource \( \Xi(\xi) \) has to actually execute from \( P_{\text{code}}(A_\alpha) \) for an execution. Furthermore, the load metrics \( I_{\text{size}}(A_\alpha) \) and \( O_{\text{size}}(A_\alpha) \) indicates the total number of bytes for an input execution state \( I_{\text{state}}(A_\alpha) \) and an output execution state \( O_{\text{state}}(A_\alpha) \), respectively.

3.2. Problem Statement

The call graph \( G(V, E) \) of an application \( A \) represents the call stack of the application parts \( A_\alpha \) (cf. Figure 3.1). Each vertex \( A_{\alpha_v} \in V \) represents an application part and each edge \( e(A_{\alpha_u}, A_{\alpha_v}) \in E \) with \( A_{\alpha_u}, A_{\alpha_v} \in V \) represents an invocation of the application part \( A_{\alpha_u} \) from the application part \( A_{\alpha_v} \). A resource \( \Xi(\xi_t) \) executes an application \( A \) by invoking its application parts \( A_\alpha \) and determines the optimal resource within \( \Xi(\xi) \) for an application part \( A_{\alpha_o} \in V \) that is offloadable by minimizing the cost function \( f_w \):

\[
\min_{\xi} f_w(A_{\alpha_o}, \Xi(\xi)) \tag{3.1}
\]
3.2. Problem Statement

where the cost function $f_w$ is the weighted sum of the execution time $T$, the energy consumption $E$, and the monetary cost $C$ of the application part $A_{\alpha o}$:

$$f_w(A_{\alpha o}, \Xi(\xi)) = w_t \cdot T(A_{\alpha o}, \Xi(\xi)) + w_e \cdot E(A_{\alpha o}, \Xi(\xi)) + w_c \cdot C(A_{\alpha o}, \Xi(\xi))$$

with the weight $w_t$ for the execution time $T$, the weight $w_e$ for the energy consumption $E$, and the weight $w_c$ for the monetary cost $C$. Utilizing a weighted sum for the cost function $f_w$, an end user or an offloading system can specify its current preference of the execution time, energy consumption, and monetary cost. For instance, if a mobile device is plugged in, an offloading system sets automatically the weight $w_e$ to 0, neglecting temporarily the energy consumption of the mobile device and benefiting from shorter execution times through code offloading.

The execution time $T(A_{\alpha o}, \Xi(\xi))$ depends on the resource $\Xi(\xi)$ that executes the application part $A_{\alpha o}$. In case of a local execution of the application part $A_{\alpha o}$ on the resource $\Xi(\xi_l)$, the execution time $T(A_{\alpha o}, \Xi(\xi))$ with $\Xi(\xi) = \Xi(\xi_l)$ evaluates to:

$$T(A_{\alpha o}, \Xi(\xi)) = T_{local}(A_{\alpha o}, \Xi(\xi_l)) = \frac{P_{\text{exe}}^{\text{pcode}}(I_{\text{state}}(A_{\alpha}), P_{\text{pcode}}(A_{\alpha}))}{P_{\text{ppwr}}(\Xi(\xi_l))}$$

In case of a remote execution of the application part $A_{\alpha o}$ on the resource $\Xi(\xi_r)$, the

![Figure 3.1.: An exemplary call graph $G(V, E)$ for an application $A$, where the set $V$ of vertexes contains the application parts $A_\alpha$ of an application $A$ and the set $E$ of edges contains the invocations of the application parts.](image-url)
3. System Overview

execution time $T(A_{ao}, \Xi(\xi))$ with $\Xi(\xi) = \Xi(\xi_r) \neq \Xi(\xi_l)$ sums up the execution time $T_{send}(A_{ao}, \Xi(\xi_r))$ of sending the input execution state $I_{state}(A_{ao})$ to the resource $\Xi(\xi_r)$, $T_{wait}(A_{ao}, \Xi(\xi_r))$ of waiting in idle mode until the resource $\Xi(\xi_r)$ executed the p-code $P_{pcode}(I_{state}(A_{ao}), P_{pcode}(A_{ao}))$, and $T_{recv}(A_{ao}, \Xi(\xi_r))$ of receiving the output execution state $O_{state}(A_{ao})$ from the resource $\Xi(\xi_r)$:

$$T(A_{ao}, \Xi(\xi)) = T_{remote}(A_{ao}, \Xi(\xi_r)) + \begin{cases} &T_{send}(A_{ao}, \Xi(\xi_r)) + T_{wait}(A_{ao}, \Xi(\xi_r)) + T_{recv}(A_{ao}, \Xi(\xi_r)), \end{cases}$$

with

$$T_{send}(A_{ao}, \Xi(\xi_r)) = \frac{I_{size}(A_{ao})}{B_{up}^{\Lambda(\xi_l:\xi_r)}} + L_{up}^{\Lambda(\xi_l:\xi_r)}$$

$$T_{wait}(A_{ao}, \Xi(\xi_r)) = \frac{P_{exe}(I_{state}(A_{ao}), P_{pcode}(A_{ao}))}{P_{ppwr}(\Xi(\xi_r))}$$

$$T_{recv}(A_{ao}, \Xi(\xi_r)) = \frac{O_{size}(A_{ao})}{B_{down}^{\Lambda(\xi_l:\xi_r)}} + L_{down}^{\Lambda(\xi_l:\xi_r)}$$

The energy consumption $E(A_{ao}, \Xi(\xi))$ also depends on the resource $\Xi(\xi)$ that executes the application part $A_{ao}$ and evolves in case of a local execution of the application part $A_{ao}$ on the resource $\Xi(\xi)$ with $\Xi(\xi) = \Xi(\xi_l)$ to:

$$E(A_{ao}, \Xi(\xi)) = E_{local}(A_{ao}, \Xi(\xi)) = T_{local}(A_{ao}, \Xi(\xi)) \cdot E_{exe}^{\Xi(\xi)}$$

In case of a remote execution of the application part $A_{ao}$ on the resource $\Xi(\xi_r)$ with $\Xi(\xi) = \Xi(\xi_r) \neq \Xi(\xi_l)$, the energy consumption $E(A_{ao}, \Xi(\xi))$ sums up the energy consumption $E_{send}(A_{ao}, \Xi(\xi_r))$ of sending the input execution state $I_{state}(A_{ao})$ to the resource $\Xi(\xi_r)$, $E_{wait}(A_{ao}, \Xi(\xi_r))$ of waiting in idle mode until the resource $\Xi(\xi_r)$ executed the p-code $P_{exe}(I_{state}(A_{ao}), P_{pcode}(A_{ao}))$, and $E_{recv}(A_{ao}, \Xi(\xi_r))$ of receiving the output execution state $O_{state}(A_{ao})$ from the resource $\Xi(\xi_r)$:

$$E(A_{ao}, \Xi(\xi)) = E_{remote}(A_{ao}, \Xi(\xi_r)) = E_{send}(A_{ao}, \Xi(\xi_r)) + E_{wait}(A_{ao}, \Xi(\xi_r)) + E_{recv}(A_{ao}, \Xi(\xi_r))$$

with

$$E_{send}(A_{ao}, \Xi(\xi_r)) = T_{send}(A_{ao}, \Xi(\xi_r)) \cdot E_{exe}^{\Xi(\xi)}$$

$$E_{wait}(A_{ao}, \Xi(\xi_r)) = T_{wait}(A_{ao}, \Xi(\xi_r)) \cdot E_{exe}^{\Xi(\xi)}$$

$$E_{recv}(A_{ao}, \Xi(\xi_r)) = T_{recv}(A_{ao}, \Xi(\xi_r)) \cdot E_{exe}^{\Xi(\xi)}$$

64
3.3. System Components

The monetary cost $C(A_{\alpha o}, \Xi(\xi))$ also depends on the resource $\Xi(\xi)$ that executes the application part $A_{\alpha o}$. It only applies in case of a remote execution, utilizing the resource $\Xi(\xi_r)$ with $\Xi(\xi) = \Xi(\xi_r) \neq \Xi(\xi_l)$ and the link $\Lambda_{\xi_l+\xi_r}$ to the resource $\Xi(\xi_r)$. The monetary cost $C(A_{\alpha o}, \Xi(\xi))$ sums up the monetary cost $C_{\text{recv}}(A_{\alpha o}, \Xi(\xi_r))$ of receiving the input execution state $I_{\text{state}}(A_{\alpha o})$ from the resource $\Xi(\xi_l)$, $C_{\text{exec}}(A_{\alpha o}, \Xi(\xi_r))$ of executing the p-code $P_{\text{pcode}}(I_{\text{state}}(A_{\alpha o}), P_{\text{pcode}}(A_{\alpha o}))$ on the resource $\Xi(\xi_r)$, and $C_{\text{send}}(A_{\alpha o}, \Xi(\xi_r))$ of sending the output execution state $O_{\text{state}}(A_{\alpha o})$ to the resource $\Xi(\xi_l)$:

$$C(A_{\alpha o}, \Xi(\xi)) = C_{\text{remote}}(A_{\alpha o}, \Xi(\xi_r))$$

$$= C_{\text{recv}}(A_{\alpha o}, \Xi(\xi_r)) +$$

$$C_{\text{exec}}(A_{\alpha o}, \Xi(\xi_r)) +$$

$$C_{\text{send}}(A_{\alpha o}, \Xi(\xi_r))$$

with

$$C_{\text{recv}}(A_{\alpha o}, \Xi(\xi_r)) = I_{\text{size}}(A_{\alpha o}) \cdot C_{\text{recv}}^{\Lambda(\xi_l;\xi_r)}$$

$$C_{\text{wait}}(A_{\alpha o}, \Xi(\xi_r)) = P_{\text{pcode}}(I_{\text{state}}(A_{\alpha o}), P_{\text{pcode}}(A_{\alpha o})) \cdot C_{\text{exec}}^{\Xi(\xi_r)}$$

$$C_{\text{send}}(A_{\alpha o}, \Xi(\xi_r)) = O_{\text{size}}(A_{\alpha o}) \cdot C_{\text{send}}^{\Lambda(\xi_l;\xi_r)}$$

Summarizing, the problem for a resource $\Xi(\xi_l)$ that distributes an application part $A_{\alpha o}$ to resources that offer their computing power is to determine the right resource $\Xi(\xi)$ for an execution of $A_{\alpha o}$ by minimizing the cost function $f_w$ (cf. Equation 3.1).

### 3.3. System Components

Related to the system model and the problem statement described above, a distribution of an application from a resource like a smart phone to another resource like a server instance at the cloud includes three main components (cf. Figure 3.2): An offloading client that offloads application parts (cf. Subsection 3.3.1), an offloading service...
3. System Overview

that executes offloaded application parts (cf. Subsection 3.3.2), and a communication network that connects both with each other (cf. Subsection 3.3.3).

3.3.1. Offloading Client

To distribute application parts to other resources, a resource runs an offloading client upon its local hardware. It provides the functionality required on the client side for code offloading to other resources. The offloading client is capable to execute portable code from an application part on the local hardware and to monitor a related execution of the portable code. Based on monitoring a local execution of an application part, the offloading client is capable to identify an application part as offloadable and to gather both its input execution state and portable code. At the invocation of an application part that is offloadable, the offloading client minimizes the cost function in Equation 3.1 to determine the optimal resource for an execution under the current situation. To obtain the parameters required for Equation 3.1, the offloading client retrieves the energy factors, samples the links to remote resources, and queries remote resources for the cost implications (cf. Section 3.1). In case of a local execution, the offloading client just continues the local execution of the application on the local hardware. In case of a remote execution, the offloading client is capable to send the input execution state and the related p-code from the application part to the remote resource, to wait in idle mode for the end of a remote execution of the application part, and to receive the output execution state from the remote resource. It is also capable to install the received output execution state on the local hardware and to continue subsequently the execution of the application.

3.3.2. Offloading Service

To execute an application part distributed by an offloading client, a resource runs an offloading service upon its local hardware. The offloading service provides the required functionality on the service side for an execution of offloaded code. To this end, the offloading service is capable to receive an input execution state and the related portable code from an offloading client, to install both on its local hardware, and to start an execution of the portable code based on the received input execution state. At the end of an execution for a distributed application part, the offloading service is capable to gather the output execution state of the execution and to send the gathered output execution state to the offloading client.
3.3.3. Communication Network

To enable resources for communicating with each other, a communication network connects an offloading client with an offloading service. A typical resource that distributes application parts to other resources is a (resource-constrained) mobile device like a smart phone, equipped with multiple connectivity for data transfer like a Wi-Fi network, a cellular network, or a Bluetooth network (cf. Section 2.2). Based on a Bluetooth link, a mobile device communicates directly via a wireless link with other resources. Based on a Wi-Fi link, a mobile device communicates via a wireless link with a Wi-Fi AP that again provides either a tethered connection or a wireless connection to other resources. Furthermore, a Wi-Fi AP typically provides via a WAN connectivity to the Internet. Last, based on a cellular link, a mobile device communicates with the Internet via a mobile communication network like the Third Generation (3G) and via fixed networks like the backbone network of a network provider.

3.4. Failure Model

Regarding the system components and their interplay for code offloading, each system component can fail at any time influencing differently the process of code offloading. The underlying failure model of this dissertation includes crash failures with eventually recovery of the offloading client, the offloading service, and the communication network.

For the offloading client, an occurrence of a crash failure disturbs a current execution of an application with its application parts. At a disturbance of a current execution of an application, an end user has to restart the application execution. Restarting an application in case of an unexpected interruption, recovery mechanisms of an operating system have to handle an occurrence of such failures. Due to recovery mechanisms implemented on an operating system, crash failures on an offloading client are out of scope of this dissertation and thus not considered in the design of a related system for code offloading. Please note that despite the occurrence of a crash failure on the offloading client during a remote execution of an application part, an end user has to pay for the utilization of the offloading service and the communication network.

For the offloading service, an occurrence of a crash failure also disturbs a current execution of an application part. Executing an application part on behalf of an offloading client, the quality of recovery mechanisms influence the efficiency of an offloading service. Owing to a failure near the end of a remote execution, for instance, a total restart of a remote execution almost doubles the waiting time, energy consumption, and monetary cost for an offloading client. To this end, the design of a related system
for code offloading considers crash failures for the offloading service.

For the communication network, an occurrence of a crash failure on a link that connects an offloading client and an offloading service disturbs the communication between both. Disturbing the communication between an offloading client and an offloading service also influences the efficiency of an offloading client. Owing to the mobility of an offloading client, for instance, an interference due to incomplete network coverage that last for several seconds or longer delays a sending of an input execution state or a receive of an output execution state. To this end, the design of a related system for code offloading considers crash failures for the communication network.

3.5. System Requirements

The realization of a system for code offloading that distributes application parts from an offloading client like a (resource-poor) mobile device to an offloading service like a (powerful) remote server instance at the cloud arises the following key requirements:

Adaptation: Due to the mobility of mobile devices, the current situation for code offloading changes constantly. Thus, a system for code offloading should adapt dynamically to varying parameters related to device, user, environment, application, or network by re-considering a made offload decision.

Efficiency: Due to the resource poverty of mobile devices, a system for code offloading should use sparingly local resources on an offloading client and links from a communication network, keeping the overhead introduced low. The sparing utilization of resources and links includes, for instance, an efficient monitoring of the current situation and the transmission of only required data.

Experience: Due to the general correlation of success for a system and the user experience, a system for code offloading should keep the burden for end users and application developers low, relying on least interventions from both developers and end users. A high level of automation combined with minimal information where necessary convince developers and end users, creating acceptance and applicability for a system.

Feasibility: For the feasibility of code offloading, a system for code offloading should guarantee the equality of a distributed execution for an application on an offloading client and offloading services with a monolithic execution for an application on an
offloading client. A location-transparent execution of applications results in equal execution states, no matter whether it is a monolithic execution or a distributed execution.

**Heterogeneity:** Due to the heterogeneity of participating offloading clients and offloading services, a system for code offloading should support different hardware platforms like smart phones, laptops, or server instances at the fog or at the cloud. The abstraction from different Instruction Set Architectures (ISAs) like x86 vs. ARM and 32-bit vs. 64-bit is fundamentally to support a vast number of participants.

**Scalability:** Due to the availability of multiple offloading services at different tiers, a system for code offloading should scale horizontally and vertically. To this end, it scales not only offloading services at one layer like the cloud but also supports multiple offloading services from different tiers like the fog and the cloud. The support of multi-tier architectures on behalf of only the classic two-tier architectures increases the efficiency of code offloading.

**Seamless Execution:** Due to the distributed nature of code offloading, a system for code offloading should enable a seamless execution of distributed application parts, handling automatically the occurrence of failures. An applicable failure handling increases not only the efficiency but also the acceptance of code offloading.

### 3.6. Summary

To provide code offloading in the landscape of Mobile Cloud Computing, the *system model* described in Section 3.1 presents a general system model for a distribution with multiple resources. It includes performance characteristics and cost implications for resources that distribute application parts to other resources and resources that offer the processing power to other resources. Based on the system model, the *problem statement* described in Section 3.2 minimizes a cost function to determine the optimal resource for an execution of an application part that is offloadable. The cost function is a weighted sum of the execution time, energy consumption, and monetary cost for an application part on a resource. Thus, it regards the trade-off between the cost for a local execution and for a remote execution. The local cost comprises the execution time and energy consumption for a local execution on an offloading client, whereas the remote cost comprises the execution time, energy consumption, and monetary cost for a remote execution on an offloading service. Both the offloading client and the offloading service are parts of the *system components* described in Section 3.3,
3. System Overview

communicating via a communication network with each other. The communication network typically consists of a wireless network and fixed networks, where an offloading client offloads computation via the communication network to an offloading service. Following Finagle’s Law of Dynamic Negatives “If anything can go wrong, it will.” [Chi63], each system component might fail. To this end, the failure model for code offloading presented in Section 3.4 involves crash failures with eventually recovery for an offloading client, an offloading service, and the communication network. Resulting from previous discussions, the system requirements described in Section 3.5 include key features and functionality of a system for code offloading.
Chapter 4
Efficient Code Offloading with Annotations

This chapter presents a basic distribution for code offloading with annotations. It supports the offloading of computation from an offloading client running on a (resource-poor) mobile device to an offloading service running on a (powerful) remote resource. As the mobile device runs the system for the basic distribution on its hardware platform, the system has to use sparingly local resources, keeping the overhead introduced low. It also has to keep the burden on application developers and end users low, being applicable with a high user experience. To this end, Section 4.1 first gives a brief overview of a basic distribution between a mobile device and a remote resource before Section 4.2 gives the system overview for the basic distribution, including the system model, the problem statement, and the system components. Due to the utilization of a runtime-layer distribution for code offloading, Section 4.3 describes the typical software stack and functionality for a runtime environment, utilized by the offloading client and offloading service. The offloading timeline in Section 4.4 outlines the interplay between an offloading client and an offloading service to realize the basic distribution for computation offloading. Providing the corresponding functionality for an offloading client and an offloading service, Section 4.5 describes the offloading framework required for the basic distribution. To evaluate the performance of the basic distribution, Section 4.6 presents the implementation of prototypes based on the Java platform, where this dissertation utilizes different Java implementations. Afterwards, Section 4.7 presents the evaluation of the basic distribution based on three prototypes, including the evaluation setup and the evaluation results. Last, Section 4.8 summarizes the main facts of the efficient code offloading with annotations presented in this chapter.

4.1. Basic Distribution

The key feature of “mobility” from mobile computing is simultaneously its main challenge, requiring techniques like code offloading. It also influences the design and per-
4. Efficient Code Offloading with Annotations

formance of a mobile device and of the systems running on a mobile device. Compared to stationary computing devices like workstations or server instances, a mobile device is a resource-poor computing environment related to performance like execution speed and battery. As a result, energy efficiency plays an important role for a mobile device and thus, for its systems like a distribution system for code offloading.

In the literature, approaches for code offloading propose either an automatic partitioning of an application at development-time (cf. CloneCloud from Chun et al. [CIM+11]), an annotation of application parts by an application developer (cf. MAUI from Cuervo et al. [CBC+10]), or a manual distribution of application parts by an application developer based, for instance, on a Remote Procedure Call (cf. RPC from Kristensen [Kri10]). An automatic partitioning of an application at development-time keeps an application developer out of business but puts a high complexity on a distribution system. It requires a full-automatic system-layer distribution that plans a huge set of possible future situations in advance. An annotation of application parts that are feasible for a distribution increases the burden on an application developer but reduces the complexity of a distribution system. It requires an half-automatic runtime-layer distribution that profits from application knowledge of an application developer. Last, a manual distribution of application parts has the highest burden on an application developer but the lowest complexity on a distribution system. It requires a developer-manual application-layer distribution that fully depends on the knowledge from an application developer.

To this end, our deployment of a distribution system for code offloading faces the trade-off between the overhead introduced to a system and the interventions required from an application developer. It utilizes a runtime-layer distribution with the annotation of application parts that are offloadable by the application developer like Cuervo et al. [CBC+10]. This little application-specific knowledge from an application developer goes far [FSS02], reducing noticeably the complexity of our distribution system. In detail, it relies on an annotation-based instrumentation of the portable code (instructions) from an application part based on a runtime environment. An application developer annotates at development-time an application part that is offloadable, marking it as a suitable candidate for a remote execution. A suitable candidate does not rely on further resources like a local sensor except the processing power for an execution of its portable code. Furthermore, it possesses a low size of information required for a remote execution, a long running remote execution, and a low size of output information received after the remote execution. During the compilation of the source code to platform-independent portable code at development-time, an offload-aware compiler introduces offload-specific portable code for application parts annotated as offloadable.
from an application developer. At run-time, the offload-specific portable code invokes at its execution an offload-aware controller that afterwards handles a distribution of the application part that is offloadable.

Summarizing, the annotation-based instrumentation of the portable code (instructions) for a distribution system keeps both the overhead introduced to a mobile device and the burden on an application developer low. In detail, our deployment invokes automatically an offload-aware controller for application parts that are feasible for a distribution based on the offload-specific portable codes introduced by the offload-aware compiler. The automatic invocation avoids, for instance, a monitoring of system information while a mobile device executes an application. Just relying on the annotation of an application part that is feasible for a distribution, our deployment hides the distributed and parallel nature of a remote execution from an application developer by automatically distributing application parts without further interventions. Nevertheless, it keeps an application developer aware of a possible distribution benefiting from application-specific knowledge from the application developer.

4.2. System Overview

The basic distribution offloads computation in the two-tier architecture (cf. Subsection 2.5.2), where each tier has one resource. The first tier consists of a (resource-poor) mobile device $\Xi(\xi_k)$ and the second tier consists of a (powerful) server instance at the cloud $\Xi(\xi_m)$. Thus, it reduces the system model of multiple tiers with multiple resources into two tiers with a resource each. The performance characteristics and cost implications for the two resources stay the same from the general system model defined in Section 3.1. The same applies to the definition for an application, where the basic distribution utilizes the model defined in Section 3.1.

Like the system model, the problem statement for the basic distribution also reduces from multiple tiers with multiple resources into two tiers with a resource each. Thus, it still minimizes the cost function $f_w$ from Equation 3.1 but only for two resources resulting in the trade-off between cost for a local execution on resource $\Xi(\xi_k)$ and cost for a remote execution on resource $\Xi(\xi_m)$:

$$\min I \cdot f_w(A_{\alpha_o}, \Xi(\xi_k)) + (1 - I) \cdot f_w(A_{\alpha_o}, \Xi(\xi_m)) \quad (4.1)$$

where $I$ equals 1 in case of a local execution and $I$ equals 0 in case of a remote execution.

The system components for the basic distribution corresponds to the system components described in Section 3.3 – namely an offloading client (cf. Subsection 3.3.1),
4. Efficient Code Offloading with Annotations

Figure 4.1.: The system components for the basic distribution, where an offloading client offloads computation via a communication network to an offloading service. Moreover, a code service disseminates portable code of applications to clients and services.

...
4.3. Runtime-layer Offloading

The basic distribution presented in this chapter utilizes a runtime-layer distribution with annotations to offload computation from an offloading client to an offloading service. As both the offloading client and the offloading service extends the basic functionality provided by a runtime environment, this section introduces the typical functionality provided by a runtime environment to execute portable code on a resource. To this end, Subsection 4.3.1 first gives an overview of the software stack and functionality of a runtime environment, before Subsection 4.3.2 describes the functionality for an offloading client and Subsection 4.3.3 for an offloading service. Last, Subsection 4.3.4 gives an overview of the Java platform with its Java Runtime Environment (JRE), extended in this dissertation for code offloading.

4.3.1. Overview

Figure 4.2a shows the software stack of a resource like a smart phone, a laptop, or a server machine, running on the hardware platform. The hardware platform typically consists of multiple components like a Central Processing Unit (CPU), a Random Access Memory (RAM), storage, and a Network Interface Card (NIC). These underlying components of the hardware platform differ in capabilities and properties like the ISA...
of a CPU (e.g., an x86 vs. ARM vs. PowerPC architecture with a 32-bit vs. 64-bit instruction set). To manage the hardware platform and enable the execution of software components, a resource executes an operating system like Windows, Linux, or Android, specialized to the components of the hardware platform. Due to different capabilities and properties of the hardware platform as well as the operating system, an application developer has to explicitly compile its application for a combination of a hardware platform and an operating system. To avoid an explicit compilation and abstract from an actual combination of a hardware platform and an operating system, the operating system executes a runtime environment like the Java Runtime Environment (JRE). The runtime environment enables an execution of applications written in platform-independent portable code, where an application consists of (multiple) parts like threads, components, or methods.

To execute the platform-independent portable code of an application part on a resource (cf. Figure 4.2b), the runtime environment has an application part loader. The application part loader loads an application part by extracting its portable code and forwarding it to the portable code compiler. The portable code compiler compiles the platform-independent portable code to platform-dependent machine code and forwards it to the machine code executor. The machine code executor executes the machine code on the actual hardware platform, where the execution of an application part can require further application parts. In this case, the machine code executor invokes the required application part, again starting its loading, compilation, and execution.

To offload an application part from a hardware platform (e.g., mobile device) to another hardware platform (e.g., server instance at the cloud) based on a runtime environment, the offloading client as well as the offloading service extend the above-described functionality of a basic runtime environment.

4.3.2. Offloading Client

Figure 4.3 shows the runtime environment of the offloading client, where a decision maker, a monitor, and a communicator extend the functionality required to distribute an execution of application parts to an offloading service. To this end, the application part loader forwards the platform-independent portable code from an application part to the decision maker. The decision maker decides whether to execute an application part on the local hardware platform or on the remote hardware platform of an offloading service. To determine the execution side of an application part, the decision maker minimizes the cost function \( f_w \) from Equation 4.1, where the decision requires parameters about the current situation like network quality. As a result, the monitor
monitors the parameters required for a decision and forwards them to the decision maker to determine the execution side. In case of a local execution of an application part, the decision maker forwards the platform-independent portable code to the portable code compiler, entering the process of compilation and execution on the local hardware platform. In case of a remote execution of an application part, the decision maker forwards the platform-independent portable code and the input execution state to the communicator. The communicator is responsible for the communication to an offloading service, sending the portable code and the input execution state to the offloading service. During a remote execution of an application part on an offloading service, the communicator waits for the end of a remote execution, receiving the output execution state of a remote execution. Receiving an output execution state of a remote execution from an offloading service, the communicator forwards it to the machine code executor. Afterwards, the machine code executor installs the output execution state of a remote execution on the local hardware platform and continues the execution of the application just as for a local execution of the application part.

### 4.3.3. Offloading Service

Figure 4.4 shows the runtime environment of the offloading service, where a communicator extends the functionality required to execute an application part on behalf of an offloading client. To this end, the communicator receives the platform-independent portable code and the input execution state of an application part from an offloading client. Possessing the portable code and the input execution state required for a remote
4. Efficient Code Offloading with Annotations

Figure 4.4.: The extended runtime environment for the offloading service enabling code offloading by utilizing a communicator.

execution, it forwards the portable code to the portable code compiler and the input execution state to the machine code executor. The portable code compiler compiles the platform-independent portable code to platform-dependent machine code and forwards it to the machine code executor. The machine code executor first installs the received input execution state, before it starts an execution of the machine code on its hardware platform based on the input execution state. At the end of an execution of an application part on the offloading service, the machine code executor forwards the output execution state to the communicator that sends it to the waiting offloading client. Afterwards, the offloading service waits for further requests from offloading clients, starting again a remote execution of an application part.

4.3.4. Java Platform

The Java programming language is a general-purpose programming language with key properties like an object-oriented design, safety, concurrency, and portability. Taking a closer look at the portability, the famous slogan “write once, run everywhere” [Tim97] from Sun Microsystems emphasizes the design goal of platform independence of the Java programming language. In detail, a Java application compiled once from an application developer runs without a recompilation on any combination of hardware platforms and operating systems that support an adequate Java platform. Achieving this platform independence of the Java programming language, a Java platform compiles the code of a Java application to Java bytecode instructions. The Java bytecode instructions are an intermediate representation of code that runs on any architecture...
4.3. Runtime-layer Offloading

compared to architecture-specific machine code. Due to the intermediate representation of Java bytecode instructions, a Java platform executes the cross-platform instructions on a run-time environment, compiling the intermediate representation of Java bytecode instructions to architecture-specific machine code at run-time.

A Java platform is responsible for the development of a Java application written in the Java programming language as well as the execution of its cross-platform instructions. For the execution of the cross-platform instructions, an end user utilizes a JRE, particularly implemented for a combination of a hardware platform and an operating system. The JRE comprises a set of components required for an execution like an implementation of a Java Virtual Machine (JVM) and Java class libraries. An application developer utilizes a Java Development Kit (JDK) that extends the functionality of a JRE with development tools. Based on the development tools like the Java Compiler `javac`, an application developer compiles Java source code to Java bytecode instructions at development-time. At run-time, a Just-In-Time (JIT) compiler from a JVM of a JRE compiles on demand the intermediate Java bytecode instructions to native machine code. Due to an extensive set of Java class libraries, a Java application can access particular features from hardware platforms or operating systems.

To this end, a Java platform abstracts from the actual hardware platform and software stack, providing a platform-independent execution of applications written in the Java programming language. At development-time, it compiles a Java application to the intermediate representation of Java bytecode instructions that it further compiles to architecture-specific machine code at run-time. To provide a platform-independent execution of Java bytecode instructions, a Java platform consists of a Java Virtual Machine, a predefined set of Java bytecode instructions, and Java class libraries.

**Java Virtual Machine:** Although the Java bytecode instructions are platform-independent, each particular combination of a hardware platform and an operating system requires its own implementation of a Java Virtual Machine (JVM). To ensure a semantic interpretation of Java bytecode instructions in the same way, different implementations of a JVM have to meet the formal requirements of the JVM specification [LYBB15]. Due to the utilization of a single specification, all implementations are interoperable no matter of underlying hardware or software. Thus, each JVM is able to execute a Java application due to the execution of the intermediate Java bytecode instructions.

Figure 4.5 shows the internal architecture for a JVM to execute an application written in the Java programming language. Typically, a Java application consists of (multiple) Java classes, where a Java class is an organizational unit possessing a number of Java bytecode instructions as well as auxiliary information for its processing like a symbol.
4. Efficient Code Offloading with Annotations

Figure 4.5.: Overview of the internal architecture for a Java Virtual Machine. [Venry]

table. At development-time, the Java compiler `javac` compiles a Java class to a binary format, representing the Java class independently from the underlying platform. At run-time, a JVM dynamically loads a Java class – that confirms to the Java class file format – from a Java application with the help of a class loader. Beside loading classes from a Java application, the class loader is also responsible for locating a Java class library, reading its content and loading the related classes within the library. Notice that the class loader only loads a Java class on demand, once a Java application calls a corresponding Java class. Possessing the Java bytecode instructions for a Java class, a JVM executes the Java bytecode instructions with the help of an execution engine. The execution engine corresponds to an abstract computing machine that translates at runtime the Java bytecode instructions to native machine code instructions based on, for instance, a JIT compiler. Afterwards, it executes the machine code instructions on the host operating system. The execution engine stores required information like translated instructions, instantiated objects, method parameters, or intermediate results in the runtime data areas. The runtime data areas include one method area, one heap, pc (program counter) registers, Java stacks, and native method stacks. A JVM has only one method area and one heap, shared by all Java threads from a Java application. It stores information from a loaded Java class file like type information in the method area and allocates memory in the heap for new objects like class instances or arrays. Each Java thread has its own pc register and Java stack. A pc register indicates the execution of the next Java bytecode instruction for a Java method and the Java stack.
stores state information for an execution of a Java method. The state information includes method parameters, local variables, return value, intermediate computations, and further data. In detail, the Java stack comprises for each Java method a stack frame containing the state information for a Java method. A JVM pushes a new stack frame at the call of a Java method onto the Java stack and pops the current stack frame at the end of a Java method discarding its state information. A Java method completes either by returning properly or by throwing an exception. Last, a JVM has native method stacks to store information about the state for native methods executed through a native method interface. The native method interface provides the capability for a JVM to execute platform-dependent native methods on host operating systems.

**Java Bytecode Instructions:** The Java bytecode instructions are platform-independent instructions, representing the instruction set of a JVM. A Java bytecode consists of one or more bytes. The first byte identifies uniquely the instruction (opcode) and the further bytes, if any, are parameters for the instruction. Thus, there are in total 255 Java bytecode instructions available, of which a JVM uses currently 198 instructions (cf. Appendix A). In detail, they provide functionality like loading and storing of data (cf. Table A.1), arithmetic operations, logic operations, and type conversions (cf. Table A.2), and object creation, object manipulation, operand stack management, control transfer, method invocation, method return, exception throwing, and monitor-based concurrency (cf. Table A.3). Out of the other 57 instructions, the JVM specification reserves 54 for future use and set 3 as permanently unimplemented. [LYBB15]

The Java bytecode instructions operate on certain data types, categorized into a set of *primitive types* and a *reference type*, holding primitive values and reference values, respectively. The primitive types are `boolean` (8-bit signed), `byte` (8-bit signed), `char` (16-bit signed), `short` (16-bit signed), `int` (32-bit signed), `long` (64-bit signed), `float` (32-bit signed), and `double` (64-bit signed). The reference type is a reference (32-bit or 64-bit) for class types, interface types, array types, and so on.

**Java Class Libraries:** Due to the platform independence of a Java platform, a Java application cannot rely on specific functions of a platform-native library. To this end, the Java class libraries provide to a Java application the common functionality of modern operating systems. They are a comprehensive, well-known set of reusable functions, relying heavily on the underlying hardware platform and operating system. At run-time, a JVM loads dynamically the Java class libraries on demand to realize common tasks like networking or file handling. For an application developer, the Java class libraries are an abstract interface to platform-dependent tasks.
4. Efficient Code Offloading with Annotations

4.4. Offloading Timeline

First of all, an application developer implements a Java application that consists at least of a Java class with a main method (cf. Subsection 4.3.4). Commonly, a Java application consists of multiple Java classes containing multiple Java methods (application parts). A Java method typically encapsulates a required functionality of a Java application. Possessing a Java method that is feasible for a remote execution, an application developer annotates it with the annotation \texttt{Offloadable} (cf. Listing 4.1):

```java
public class HelloWorld {
    public static void main(String[] args) {
        // Prints the String "Hello, World!" to the console
        System.out.println("Hello, World!");

        float flt = dstrbtlMthd(args);
        System.out.println("Computation finished: " + flt);
    }

    @Offloadable
    public static float dstrbtlMthd(String[] args) {
        float result = 0.0F;
        // Long running computation
        ...
        return result;
    }
}
```

Listing 4.1: The Java source code of an extended “Hello, World”-application that contains a Java method that is offloadable.

After the implementation of a Java application and a (possible) annotation of Java methods as “offloadable”, an application developer compiles the Java source code of an application with the program javac, the JAVA Compiler. It translates Java source code from an application into platform-independent Java bytecode instructions. To this end, we extended javac to introduce at development-time offload-specific Java bytecode instructions for a Java method annotated with \texttt{offloadable}. During this translation of Java source code into Java bytecode instructions, it introduces an offload-specific Java bytecode instruction at the start of a Java method annotated with \texttt{offloadable} and another one at each end of the Java method (cf. Subsection 4.5.1).

After the compilation of a Java application and a dissemination via a code service to mobile devices, an end user can start an execution of a Java application on its mobile device. Please note that usually an end user downloads and installs an application on its mobile device from a code service only once, before the first execution of an application (cf. Figure 4.6). Through a start of a Java application, an operating system on a mobile device runs a JRE extended for the basic distribution by an offloading framework (cf.
4.4. Offloading Timeline

Figure 4.6.: Offloading timeline of the basic distribution for an application part between
an offloading client and an offloading service via a communication network.
Both download and install the application from a code service.

Section 4.5). Beside typical components of a JRE for an execution of a Java application
(cf. Subsection 4.3.4), the extended JRE also starts the additional components of the
offloading client required for the basic distribution (cf. Subsection 4.5.2). Afterwards,
it executes the Java application by invoking its main method that again invokes further
Java methods, communicating with each other.

After the invocation of a Java method on the offloading client, an execution of
platform-independent Java bytecode instructions requires a compiler that translates
Java bytecode instructions to platform-dependent machine code during run-time. Due
to the introduction of offload-specific Java bytecode instructions at development-time,
we extended the standard Java compiler of a Java Virtual Machine (cf. Subsec-
tion 4.5.1). The extension comprises the translation of offload-specific Java bytecode
instruction to platform-dependent machine code at run-time. The execution of trans-
lated machine code invokes the execution controller of the offloading client (cf. Sec-
tion 4.5.2). Subsequently, the execution controller invokes the offload controller of the
offloading client, informing the offload controller of the call of a Java method annotated with \textit{offloadable} by an application developer.

After the invocation of the offload controller on the offloading client, the offload controller decides whether to distribute the called Java method to an offloading service (cf. Figure 4.6). To this end, it minimizes the cost function in Equation 4.1, requiring information about the application, the network, the user, the device, and the remote resource (cf. Figure 4.7b). The information about the application comprises the input execution state of the Java method, the execution time of the Java method on the local resource and on the remote resource, as well as the size of the output execution state of the Java method. To retrieve the input execution state, the execution controller on the offloading client gathers the input execution state from the local resource, having direct access to the Java stack (cf. Subsection 4.5.2). The input execution state comprises the method parameters, the member values of the class declaring the method, and the static members of all classes reachable. To retrieve the execution time on the local and remote resource, the app profiler on the offloading client builds a profile model based on past invocations. To this end, it requires the input execution state together with the Java bytecode instructions of the called Java method. The app profiler retrieves the Java bytecode instructions from the execution controller that has access to the declaring class of a Java method. Based on the profile model built, the offload controller on the offloading client predicts the execution time for an execution on the local resource and on the remote resource. The same applies to the prediction for the size of the output execution state, where the app profiler builds another profile model based on past invocations. For this, it requires the input execution state together with the size of the output execution state. The information about the network comprises the link bandwidth and link latency to the remote resource. Possessing the link bandwidth and link latency, the offload controller on the offloading client determines the duration for sending the input execution state to the remote resource as well as for receiving the output execution state from the remote resource. To retrieve the link bandwidth and link latency, the network monitor on the offloading client monitors both to the remote resource. The information about the user comprises the user-defined weights for the execution time, the energy consumption, and the monetary cost. To retrieve the user-defined weights, the user interface on the offloading client provides an interface for the user to enter the weights. The information about the device comprises factors for the energy consumption and for the performance of the mobile device. To retrieve the factors for the energy consumption, the device interface on the offloading client provides an interface for the device manufacturer to enter the device-specific factors. To retrieve the factor for the performance, the device benchmark on the offloading client
4.4. Offloading Timeline

(a) Detailed timeline for a remote execution of an offloadable application part.

(b) Flow of decision making for code offloading of an offloadable application part.

Figure 4.7.: Overview of (a) the timeline and (b) decision making for code offloading.

benchmarks the local performance of executing Java bytecode instructions. Last, the information about the remote resource comprises factors for its monetary cost and for its performance. To retrieve the factors for the monetary cost and the performance, the service monitor on the offloading client creates a resource request, requesting the factors from an offloading service. Afterwards, it forwards the resource request to the service connector on the offloading client. The service connector handles the communication to the remote resource and thus, sends the resource request to the offloading service.

Upon the receipt of a resource request at the client connector on the offloading service, the client connector requests the factors of the performance and the monetary cost from the offload controller (cf. Subsection 4.5.3). The offload controller on the offloading service retrieves the factor for the performance from the device benchmark and the factors of the monetary cost from the provider interface. To retrieve the factor for the performance, the device benchmark on the offloading service benchmarks the local performance of executing Java bytecode instructions. To retrieve the factors for the monetary cost, the provider interface on the offloading service provides an interface for the provider of the remote resource to enter the resource-specific factors. Possessing the factors for the performance and the monetary cost of the remote resource, the offload controller on the offloading service forwards the factors to the requesting client connector. The client connector on the offloading service first creates a resource response that contains the factors, before it sends the resource response to the service connector of the offloading client.
4. Efficient Code Offloading with Annotations

After the offload controller on the offloading client possesses all of the information required to minimize the cost function, it determines the execution side of the called Java method. In case of a local execution of the Java method, the offload controller on the offloading client just continues the local execution of the Java method by returning its invocation that again returns the invocation of the execution controller. In case of a remote execution of the Java method (cf. Figure 4.6), the offload controller forwards the input execution state of the Java method to the service connector and waits for a response from it. Receiving the input execution state from the offload controller, the service connector creates an offload request – containing the input execution state of the Java method – and sends it to the client connector on the offloading service.

Upon the receipt of an offload request on the client connector of the offloading service, the client connector forwards the input execution state to the offload controller (cf. Subsection 4.5.3). The offload controller on the offloading service forwards the input execution state to the execution controller that installs the state on the computing resource and starts the execution of the Java method by calling the Java method with its method parameters contained in the input execution state.

After the execution of the offload request on the offloading service, the execution of the offload-specific Java bytecode instruction invokes the execution controller. The execution controller gathers the output execution state of the Java method. It comprises the information of an input execution state, the return value of the Java method, and the value for the register with the “ByteCode Index” (bci) for the execution. The bci register indicates the index of the Java bytecode instruction executed next. It holds at the end of the execution the Java bytecode instruction executed last like a return instruction. Possessing the output execution state of the Java method, the execution controller forwards it to the offload controller that again forwards it together with the monetary cost of the execution to the client connector. The client connector creates an offload response – containing the output execution state and the monetary cost – and sends it to the service connector of the offloading client.

Upon the receipt of an offload response on the service connector of the offloading client, the service connector forwards the output execution state and the monetary cost to the offload controller (cf. Subsection 4.5.2). The offload controller forwards the output execution state to the execution controller and pays the monetary cost charged from the offloading service. The execution controller installs the information in the output execution state on the Java stack and jumps to the last Java bytecode instruction of the Java method. Due to the jump to the last Java bytecode instruction, the local execution returns the Java method with the return value contained in the output execution state and continues the execution of the Java application.
4.5. Offloading Framework

The offloading framework provides the functionality for the basic distribution between an offloading client and an offloading service on the Java platform. The offloading framework requires an extension of the standard Java compiler of the Java platform to translate the offload-specific Java bytecode instructions at development-time as well as at run-time (cf. Subsection 4.5.1). Moreover, the offloading client (Subsection 4.5.2) as well as the offloading service (Subsection 4.5.3) of the offloading framework extend the functionality provided by a JRE.

4.5.1. Extended Java Compiler

The JAVA Compiler (javac) is responsible to translate the Java source code of a Java application into platform-independent Java bytecode instructions at development-time (cf. Subsection 4.3.4). As an application developer marks Java methods feasible for a remote execution with the annotation \textit{offloadable}, javac has to know this offload-specific annotation and how to handle it. To this end, we extended javac to instrument Java bytecode instructions for an efficient identification of annotated Java methods. For an annotated Java method, it introduces the offload-specific Java bytecode instructions \texttt{offload} and \texttt{offload\_end} during the translation of Java source code into Java bytecode. In detail, it inserts \texttt{offload} at the start of the method body as the first executed Java bytecode instruction. Knowing the start of an execution of a Java method that is offloadable, the Java bytecode instruction \texttt{offload\_end} marks the end of its execution. Thus, javac inserts it before each end of the method body. Execution ends are a return Java bytecode instruction – \texttt{return}, \texttt{ireturn}, \texttt{lreturn}, \texttt{freturn}, \texttt{dreturn}, or \texttt{areturn} – or a Java bytecode instruction throwing an exception like \texttt{throw} (cf. Appendix A). Executing \texttt{offload\_end} right before the last Java bytecode instruction of a Java method, the offloading framework gathers the execution state just before the execution of the last instruction destroys the execution state.

4.5.2. Offloading Client

Figure 4.8 shows the runtime environment of the offloading client, providing the client-side functionality for the basic distribution. It consists of an offload compiler, an execution controller, a state generator, an app profiler, a network monitor, a network interface, a user interface, a device interface, a device benchmark, a service monitor, a service connector, and an offload controller.
4. Efficient Code Offloading with Annotations

**Offload Compiler:** The standard Java compiler is responsible to translate platform-independent Java bytecode instructions into platform-dependent machine code at runtime (cf. Subsection 4.3.4). As `javac` introduces the offload-specific Java bytecode instructions `offload` and `offload_end` at development-time, a standard Java compiler of a Java implementation has to know how to translate these Java bytecode instructions to machine code. To this end, we extended the standard Java compiler to an offload compiler. It translates the two offload-specific Java bytecode instructions into offload-specific machine code that internally invokes the execution controller at its execution.

**Execution Controller:** The execution controller of the offloading client is an interface between the execution engine and the offloading components within a JVM (cf. Subsection 4.3.4). An execution of the assembler code for the offload-specific Java bytecode instructions `offload` and `offload_end` invokes the execution controller. Only at an execution of `offload`, the execution controller invokes the offload controller to inform it of the call of a Java method that is feasible for a remote execution. The main tasks of the execution controller are the installation of an output execution state after a remote execution of a Java method and the manipulation of an execution of a Java method. Moreover, it also interacts with the state generator to obtain an input execution state and an output execution state for a called Java method.
To install an output execution state after an execution of a Java method on a remote resource, the execution controller replaces the member values of the declaring Java class with the related values contained in an output execution state. It also pushes the return value contained in an output execution state on the stack frame of the Java stack. Notice that the execution controller only replaces the member values and pushes the return value, because further values on the stack frame of the Java stack have only a local scope and thus, are only valid during an execution of the Java method. Due to the immediate return of an execution of the Java method after the Java bytecode instruction `offload_end`, the execution engine destroys the created stack frame.

To manipulate an execution of a Java method, the execution controller changes the values of the registers from the execution engine (cf. Subsection 4.3.4), holding state information during an execution of a Java method. For instance, it has a register that holds the “ByteCode Index” (bci) indicating an index to the next Java bytecode instruction executed by the execution engine (cf. pc registers). By setting the bci register to the last Java bytecode instruction of a Java method, the execution engine immediately returns the call of a Java method.

**State Generator:** The state generator of the offloading client is an interface between the runtime data areas and the offloading components within a JVM (cf. Subsection 4.3.4). To this end, the main tasks of the state generator are the marshaling of Java data types, the gathering of execution state information before an execution of a Java method, the gathering of execution state information after an execution of a Java method, and the gathering of Java bytecode instructions of a Java method.

To marshal Java data types, the state generator distinguishes between Java primitives, Java objects, and Java arrays (cf. Subsection 4.3.4). For a Java primitive, it just copies the Java identifier of the data type together with related data into a platform-independent representation. For a Java object, it visits each member value of the Java class and stores the Java primitives. In case of the member value of the Java object corresponds itself to a Java object or a Java array, the state generator marshals recursively this member value. For an one-dimensional Java array, the state generator visits each entry and stores the related data of a Java primitive or a Java object. For a multi-dimensional Java array, it visits recursively each entry of each dimension, reaching at the end an one-dimensional Java array that it marshals.

To gather the execution state information before the execution of a called Java method, the state generator copies the parameters of a Java method from the Java stack and from the Java heap (cf. Section 4.3.4). On the execution of the first Java bytecode instruction `offload`, the execution engine initialized a new stack frame on
the Java stack for the invocation of a Java method. The stack frame contains the parameters of a Java method in ascending order, where the values of Java primitives are on the stack and the values of Java objects or Java arrays are on the heap linked by a reference on the stack frame. To this end, the state generator directly copies the values of a Java primitive from the stack frame on the Java stack, whereas it first marshals the Java object or Java array from the Java heap linked by a reference. Beside the parameters of a Java method, the first value at the stack frame on the Java stack is a reference to the this object\textsuperscript{1}, corresponding to the declaring Java class of the Java method. Visiting the declaring Java class, the state generator directly copies its member values. Last, the state generator requires the static members of all classes reachable. It visits all loaded Java classes from the class loader stored in a list within the JVM (cf. Section 4.3.4), copying the static members of the reachable Java classes. Thus, it possesses the information required to create an input execution state.

To gather the execution state information after the execution of a Java method, the state generator first creates an input execution state. Afterwards, it further requires the return value for the Java method, corresponding to the last value on the stack frame of the Java stack due to the execution of the Java bytecode instruction \texttt{offload_end}. In case the last value on the stack frame of the Java stack is a Java primitive, the state generator just copies it. In case of the last value on the stack frame of the Java stack is a Java object or a Java array, the state generator marshals the Java object or Java array referenced by the last value on the stack frame. Thus, it possesses the information required to create an output execution state.

To gather the Java bytecode instructions of a called Java method, the state generator visits the declaring Java class of a Java method through the reference to the this object on the frame stack on the Java stack. The declaring Java class of the Java method possesses the Java bytecode instructions of a Java method. Thus, the state generator directly copies the Java bytecode instructions from the declaring Java class.

\textbf{App Profiler:} The offload controller of the offloading client invokes the app profiler to retrieve two profile models built from past invocations of a Java method (history-based approach). One model profiles the execution time for a local execution of a Java method on the hardware platform, where the other model profiles the size of the output execution state after an execution of a Java method. To build the two profile models, the app profiler retrieves from the execution controller after a local

\textsuperscript{1}In case of a static call of a Java method (e.g., \texttt{invokestatic}), the reference to the this object is missing on the related Java stack. Due to the static invocation of the Java method, however, the member values of the declaring Java class correspond to the static member values of the declaring Java class, accessed through the class definition.
4.5. Offloading Framework

execution of a Java method its input execution state, its output execution state, and its execution time. Afterwards, the app profiler creates a data sample for each profile model. The first data sample consists of the input execution state together with the execution time. The second data sample consists of the input execution state together with the output execution state. Creating data samples after a local execution for a Java method, the training set on the offloading client increases over time. Thus, the app profiler trains from time to time new profile models due to an increased set of training data. Notice that the offloading client does not rely on a specific algorithm to profile the execution time or the size, being out of scope of this dissertation. In detail, the basic distribution utilizes two k-Nearest Neighbors\(^2\) algorithms like in [MF10] for regression-based machine learning, building and training both models.

**Network Monitor:** The offload controller of the offloading client invokes the network monitor to retrieve the link bandwidth and the link latency from the offloading client to an offloading service and vice versa. To this end, the network monitor probes the uplink and downlink in five iterations each, where an iteration comprises a ping followed by a packet with 10 kB of data. Afterwards, the network monitor takes the average, possessing the actual link bandwidth and link latency.

**Network Interface:** The offload controller of the offloading client invokes the network interface to retrieve the factors for the monetary cost of the network. The factors are the monetary cost for a utilization of the communication network to and from an offloading service. To this end, the network interface provides an interface – more precisely a configuration file, where a network provider defines the static or dynamic factors of the monetary cost of the communication network.

**User Interface:** The offload controller of the offloading client invokes the user interface to retrieve the user-defined weights for the execution time, the energy consumption, and the monetary cost of the cost function (cf. Section 3.2). To this end, the user interface provides a graphical interface – more precisely three bars, where an end user of a mobile device defines its preference for each weight. Based on the weights defined by an end user, the end user influences the solution of the cost function.

**Device Interface:** The offload controller of the offloading client invokes the device interface to retrieve static factors for the energy consumption. The static factors are the energy consumption for executing code on the hardware platform, sending and

\(^2\)For a detailed explanation of the k-Nearest Neighbor algorithm, please take a look at [WFHP16].
4. Efficient Code Offloading with Annotations

receiving bytes via the communication network to and from an offloading service, and waiting in idle mode during a remote execution. To this end, the device interface provides an interface – more precisely a configuration file, where a device manufacturer defines the static factors of the energy consumption.

Device Benchmark: The offload controller of the offloading client invokes the device benchmark to retrieve the performance factor of executing Java bytecode instructions. To this end, the device benchmark benchmarks the computing speed of a hardware platform by measuring the execution time for a predefined number of Java bytecode instructions. In detail, the benchmark comprises measurements of Java bytecode instructions for the manipulation of Java primitives (Integer, Long, Float, and Double), Java objects, and Java arrays (cf. Subsection 4.3.4). Based on the measurements of Java bytecode instructions, the device benchmark calculates the performance factor, possessing a comparative value for the performance of hardware platforms.

Service Monitor: The offload controller of the offloading client invokes the service monitor to retrieve the factors of the performance and the monetary cost for a remote execution on an offloading service. To this end, it creates a resource request that is actually a packet containing only an identifier for requesting the factors. After the creation of the resource request, the service monitor forwards it to the service connector of the offloading client and waits for a resource response from the offloading service. A resource response is a packet with a related identifier and the requested factors.

Service Connector: The offload controller and the service monitor of the offloading client invoke the service connector to handle the communication between the offloading client and an offloading service. To this end, it receives the input execution state from the offload controller, creating an offload request and sending it to the client connector on an offloading service. Afterwards, the service connector waits for a receive of an offload response at the end of a remote execution. Receiving an offload response from a client connector on an offloading service, the service connector on the offloading client extracts the output execution state and the monetary cost of a remote execution, forwarding both to the offload controller. Beside sending of offload requests and receiving of offload responses, the service connector receives resource requests from the service monitor. To this end, it sends a resource request to the client connector on an offloading service and waits for a receive of a resource response. Afterwards, the service connector on the offloading client extracts the factors of the performance and the monetary cost of an offloading service and forwards both to the offload controller.
4.5. Offloading Framework

**Offload Controller:** The execution controller of the offloading client invokes the offload controller at the execution of the offload-specific Java bytecode instruction `offload`. This indicates the call of a Java method that is feasible for a remote execution. Thus, the offload controller minimizes the cost function from Equation 4.1, requiring the information about the application, the network, the user, the device, and the offloading service. For the information about the application, the offload controller invokes the execution controller retrieving the input execution state of the called Java method. Furthermore, it invokes the app profiler retrieving the profile models for the execution time on the hardware platform and for the size of the output execution state. Based on the profile models built from the app profiler, the offload controller predicts the execution time on the hardware platform and the size of the output execution state based on the input execution state of the called Java method. To retrieve the execution time on an offloading service, the offload controller multiplies the execution time predicted for the local hardware platform by a speedup factor. It calculates the speedup factor based on the factor for the performance of the local hardware platform and of the offloading service. For the information about the network, the offload controller invokes the network monitor retrieving the link bandwidth and the link latency to an offloading service. For the information about the user, the offload controller invokes the user interface retrieving the user-defined weights for the execution time, the energy consumption, and the monetary cost. For the information about the device, the offload controller invokes the device interface retrieving the factors for the energy consumption and for the performance of the hardware platform. For the information about the offloading service, the offload controller invokes the service monitor retrieving the factors for the monetary cost and for the performance of the offloading service.

By minimizing the cost function with the retrieved information, the offload controller decides whether it executes the called Java method on the hardware platform of the offloading client or distributes the execution of the called Java method to the hardware platform of the offloading service. In case of a local execution on the offloading client, the offload controller just returns its invocation – and thus, the invocation of the execution controller – continuing the execution of the Java method locally on the offloading client. In case of a remote execution on the offloading service, the offloading client forwards the input execution state to the service connector. Afterwards, it waits for the output execution state and the monetary cost of a remote execution on the offloading service from the service connector. Receiving the output execution state and the monetary cost from the service connector, the offload controller forwards the output execution state to the execution controller. It also pays the monetary cost charged from the offloading service for the remote execution of the Java method.
4. Efficient Code Offloading with Annotations

4.5.3. Offloading Service

Figure 4.9 shows the runtime environment of the offloading service, providing the service-side functionality for the basic distribution. It consists of an offload compiler, an execution controller, a state generator, a provider interface, a device benchmark, a client connector, and an offload controller.

**Offload Compiler:** The offload compiler of the offloading service corresponds to the offload compiler on the offloading client (cf. Subsection 4.5.2), translating both offload-specific Java bytecode instructions into offload-specific machine code.

**Execution Controller:** The execution controller of the offloading service corresponds to the execution controller on the offloading client (cf. Subsection 4.5.2). Only the execution of the offload-specific Java bytecode instruction `offload_end` invokes the execution controller that invokes the offload controller, informing it about the execution end of a Java method and forwarding the output execution state to it.

**State Generator:** The state generator on the offloading service provides the same functionality as the state generator on the offloading client (cf. Subsection 4.5.2),
extended with the functionality to install an execution state for an invocation of a Java method. To install an execution state for an invocation of a Java method, the state generator replaces the member values of the declaring Java class with the related values contained in the execution state. Notice that the state generator only replaces the member values, because the offload controller calls a Java method with the right parameters contained in the execution state. For the replacement of the member values, the state generator visits the declaring Java class through the reference to the this object on the stack frame on the Java stack. Afterwards, it replaces the member values in memory with the values contained in the execution state.

**Provider Interface:** The offload controller of the offloading service calls the provider interface to retrieve dynamic factors. The dynamic factors are the monetary cost for executing code on the hardware platform, and sending and receiving bytes via the communication network to and from an offloading client. To this end, the provider interface provides an interface – more precisely text entry boxes, where a provider of a remote resource defines the resource-specific factors for the monetary cost.

**Device Benchmark:** The device benchmark of the offloading service corresponds to the device benchmark on the offloading client (cf. Subsection 4.5.2), benchmarking the performance of the hardware platform.

**Client Connector:** According to the basic distribution (cf. Section 4.4), the receive of a resource request or an offload request invokes the client connector of the offloading service. By receiving a resource request from the service connector of an offloading client, the client connector on the offloading service invokes the offload controller to retrieve the factors for the performance and the monetary cost of executing, sending, and receiving. Retrieving the factors from the offload controller, the client connector creates a resource response – containing the factors – and sends it to the requesting service connector on the offloading client. By receiving an offload request from the service connector of the offloading client, the client connector on the offloading service extracts the input execution state from the offload request. Afterwards, the client connector forwards it to the offload controller that starts the execution of the Java method. After the execution of the Java bytecode instructions of the Java method, the offload controller invokes the client connector, forwarding the output execution state and the monetary cost of the remote execution. The client connector creates an offload response – containing the output execution state and the monetary cost of the remote execution – and sends it to the service connector of the offloading client.
4. Efficient Code Offloading with Annotations

**Offload Controller:** The client connector of the offloading service invokes the offload controller due to the receive of an offload request. To this end, the client connector forwards the input execution state of a Java method to the offload controller. The offload controller forwards it to the execution controller that installs the input execution state for an invocation of the Java method (cf. Subsection 4.5.2). Afterwards, the offload controller starts the execution of the Java method. At the end of an execution of a Java method, the execution controller invokes the offload controller and forwards the output execution state gathered due to the execution of the offload-specific Java bytecode instruction `offload_end`. Based on the time elapsed for the execution of the Java method, the offload controller calculates the monetary cost for the execution. Possessing the output execution state and the monetary cost for the execution of a Java method, the offload controller forwards both to the client connector.

4.6. Implementation

The basic distribution is the basis for the concepts and algorithms presented in this dissertation. Thus, we implemented the functionality required for the basic distribution on the Java platform (cf. Subsection 4.3.4). For the Java platform, there are different implementations available, where this dissertation deploys a prototype for code offloading based on the Jikes Research Virtual Machine (cf. Subsection 4.6.1), the Open Java Development Kit (cf. Subsection 4.6.2), and the Android Open-Source Project (cf. Subsection 4.6.3). Beside the implementation of prototypes based on the Java platform, this dissertation utilizes two self-designed measurement boards (cf. Subsection 4.6.5) that measure the energy consumption of the offloading clients.

4.6.1. Jikes Research Virtual Machine

The Jikes Research Virtual Machine (RVM) is an open-source implementation of a JVM, written in the Java programming language (meta-circular). Despite the meta-circular implementation style, the Jikes RVM is self-hosted, where it executes its Java bytecode instructions on itself. In November 1997, the development of the Jikes RVM started as an internal research project at IBM’s Thomas J. Watson Research Center. At 2012, the Jikes RVM won the ACM SIGPLAN Software award [SIG12]. The citation of the award says that “The high quality and modular design of Jikes has made it easy for researchers to develop, share, and compare advances in programming language implementation.” At the year 2015, the Jikes RVM possesses a large community of researchers and scientific publications, spawning more than 200 papers and at least 40
4.6. Implementation

Figure 4.10.: The software stack from the Android OS. [And16a, And16b]

dissertations at almost 100 universities [Mac16]. Focusing on the research of new techniques, the Jikes RVM offers a rapid prototyping for a JVM, supporting the PowerPC and the Intel x86 32-bit ISAs with a Linux operating system. As Java class libraries, it utilizes either the implementation from Apache Harmony or from GNU Classpath.

4.6.2. Open Java Development Kit

The Open Java Development Kit (OpenJDK) is a free and open-source implementation of the Java Platform, Standard Edition (Java SE). The main components of the OpenJDK project are an implementation of the JVM named HotSpot, the Java class libraries, and the Java compiler javac. The OpenJDK project implements its components mainly in the C++ programming language and the Java programming language. Back to 2006, Sun Microsystems announced that Java would become open-source and released main parts like the Java HotSpot virtual machine as free software. To provide a JDK as free software, Sun Microsystems released at the year 2007 the Java class libraries as open-source code. Today, the Java platform of OpenJDK is the standard Java installation on many Linux distributions like Ubuntu.
4. Efficient Code Offloading with Annotations

4.6.3. Android Open-Source Project

The Android Open-Source Project (AOSP) is a mobile Operating System (OS) and application environment for mobile devices like smart phones or tablets. It supports multiple architectures like ARM, x86, and MIPS\(^3\) architecture in 32-bit and 64-bit variants. Initially developed by Android Inc. and later bought by Google in the year 2005, it provides an open-source software stack. The commercial version Android OS from Google had 1.4 billion active devices in September 2015 [Goo15]. Thus, it is the most widespread (mobile) OS. Due to the open nature of AOSP, it has a large community of developers that utilizes AOSP for own projects. The software stack of AOSP runs above a device hardware and comprises a Linux kernel, a Hardware Abstraction Layer (HAL), an Android platform that consists of the Android runtime, the native libraries, and the Android framework, the Binder Inter-Process Communication (IPC), and the applications (cf. Figure 4.10). Depending on the actual hardware platform of a mobile device, AOSP utilizes a Linux kernel specialized with important additions (cf. the Binder IPC driver in Figure 4.10) for a (mobile) embedded platform. Through the Linux kernel and the HAL, AOSP accesses device resources like camera, audio, or network functions. To this end, the HAL provides a standard interface for software hooks between the higher-level Android platform and the lower-level drivers. The Android platform consists of the Android runtime, the native libraries, and the Android framework. The Android runtime includes the Dalvik Virtual Machine (DVM) and Java-compatible core libraries to execute applications written in the Java programming language. The DVM contains a JIT compiler that executes dex-code (Dalvik EXecutable), compiled from Java bytecode at development-time. With Android version 5.0, Google replaced the JIT compiler with an Ahead-Of-Time (AOT) compiler that compiles the entire code of an application into machine code at the installation of an application. The Java-compatible core libraries correspond to a subset of the discontinued Apache Harmony. Since Android version 7.0, OpenJDK replaces Apache Harmony. The native libraries provide access to particular features of a hardware platform. For instance, OpenGL ES provides an application programming interface to render graphics on an embedded device. The Android framework summarizes system services like the Window Manager or the MediaPlayer Service that access the hardware platform of a mobile device. In detail, AOSP groups system services under the System Server and the Media Server (cf. Figure 4.10). The System Server contains services involved in the system like the Window Manager. The Media Server contains services involved in playing and recording media like the MediaPlayer Service. The Binder

\(^3\)Microprocessor without Interlocked Pipeline Stages (MIPS)
IPC provides mechanism for a communication across process boundaries, enabling an interaction between system services as well as between system services and higher-level applications. Actually, there are two sources for applications in the software stack. The first source is the pre-installation of applications, providing key capabilities of mobile devices like calling or browsing. The second source is the user-driven installation of applications, customizing the functionality of a mobile device with third-party applications. To this end, Android offers the Android Software Development Kit (SDK), supporting an environment for an open development of applications. It is a comprehensive set of development tools like a debugger or software libraries for application developers to deploy applications written in the Java programming language. Until December 2014, Google provided the Android SDK based on a plugin – namely the Android Development Tools (ADT) plugin – for the Eclipse Integrated Development Environment (IDE). Nowadays, the Android Studio from Google is the primary IDE for the deployment of Android applications.

4.6.4. Modifications

#### Jikes RVM and OpenJDK:

The changes to the Jikes RVM (cf. Subsection 4.6.1) and to the OpenJDK (cf. Subsection 4.6.2) comprise modifications to the JAVA Compiler (javac), the implementation of the JVM, and the Java core class libraries (cf. Subsection 4.3.4). The changes to javac include the integration of the offload-specific annotation and the offload-specific Java bytecode instructions. The integration enables the compilation of offload-specific Java source code to offload-specific Java bytecode instructions at development-time (cf. Subsection 4.5.1). The changes to the implementation of the JVMs include the execution of the offload-specific Java bytecode instructions and the functionality required for the offloading client and for the offloading service. Thus, the JVM provides either the execution as an offloading client or as an offloading service. The changes to the Java core class libraries include the integration of the offload-specific annotation in the java.lang package.

#### AOSP:

The modification to the AOSP comprises changes to the Android runtime and an additional system service to the Android application framework (cf. Subsection 4.6.3). The changes to the Android runtime include the integration of the offload-specific annotation, the offload-specific Java bytecode instructions, and the functionality required for the offloading client (cf. Subsection 4.5.2). To this end, we adapted among others the DVM corresponding to an implementation of a JVM that executes the Dalvik Bytecode on the hardware platform. In detail, we enabled the DVM to
execute the two offload-specific Dalvik bytecodes `offload` and `offload_end`. Moreover, we enabled it to gather an input execution state of a Java method, install an output execution state received from an offloading service, and skip a local execution. Through the execution of `offload`, the DVM informs an additional system service, namely the Offload Manager, from the Android application framework of the current execution of a Java method that is offloadable. Like other managers (e.g., Activity Manager or Location Manager) from the Android application framework, the Offload Manager also possesses a category in the Android settings. Via this Android setting, an end user easily controls the system service of the Offload Manager like starting the service, stopping the service, or changing the user-defined weights.

### 4.6.5. Measurement Boards

To measure the energy consumption of an offloading client that offloads computation to an offloading service, this subsection provides an insight about the self-designed measurement boards for this dissertation. Due to mobile and stationary measurements of the hardware platforms from a smart phone (cf. Section C.1), a netbook (cf. Section C.2), and a laptop (cf. Section C.3), this dissertation designs two different
4.6. Implementation

Figure 4.12.: The measurement board for the Dell Inspiron Mini 10v and the Lenovo ThinkPad T61, consisting of a self-designed board and a 16-bit Analog Data Acquisition Board from Meilhaus Electronic [Ele17].

measurement boards, namely one for the Samsung Galaxy Nexus and another one for the Dell Inspiron Mini 10v & Lenovo ThinkPad T61.

**Samsung Galaxy Nexus:** A Raspberry Pi and a self-designed extension board measures the power consumption of the battery-operated smart phone (cf. Figure 4.11). The Raspberry Pi runs a real-time operating system – a Linux with RT PREEMPT patch – polling concurrently samples from a 12 bit Analog-to-Digital Converter (ADC) on the extension board. The ADC samples the continuous voltage $U$ and current $I$ of the device under test, namely the Samsung Galaxy Nexus. To this end, the ADC measures the voltage drop along a 0.01 Ohm shunt resistor, connected in series with the device under test. The measured voltage drop is proportional to the current $I$ flowing through the shunt (Ohm’s law $R = \frac{U}{I}$). Furthermore, the extension board measures the voltage $U$ directly at the device under test. Last, the extension board amplifies the voltage of the shunt resistor with the help of an instrumentation amplifier for precise measurements. The corresponding design of hardware and software are open-source\(^4\).

**Dell Inspiron Mini v10 & Lenovo ThinkPad T61:** A self-designed board together with the ME-Jekyll ME-4610 PCI 16-bit Analog Data Acquisition Board from Meilhaus Electronic (cf. [Ele17]) measures the power consumption of the power-supplied

\(^4\)github.com/duerrfk/rpi-powermeter
netbook and laptop (cf. Figure 4.12). The self-designed board amplifies the voltage drop along a 0.1 Ohm shunt resistor with the help of an operational amplifier for precise measurements. It connects the shunt resistor in series with the device under test, either the Dell Inspiron Mini 10v or the Lenovo ThinkPad T61. Moreover, it also measure the voltage $U$ directly at the device under test. Both the voltage and the voltage drop are analog inputs to the 16-bit Analog Data Acquisition Board that does a high-precision 16 bit/500 kHz A/D conversion. As the analog inputs from the Data Acquisition Board have a voltage range of ±10 V, the self-designed board has a preceding voltage divider for the power input of 19 V/20 V from the netbook/laptop.

4.7. Evaluation

To evaluate the performance of the basic distribution, a mobile device executes a Java application on an unaltered Java Runtime Environment (JRE) and on an offload-aware JRE. To this end, Subsection 4.7.1 describes the evaluation setup before Subsection 4.7.2 presents the evaluation results.

4.7.1. Setup

The different implementations of the prototype for the basic distribution categorize the evaluation setup (cf. Section 4.6), where this subsection describes the setup for the prototypes of the Jikes RVM, the OpenJDK, and the AOSP.

Jikes Research Virtual Machine

For the Jikes RVM, the netbook Dell Inspiron Mini 10v (cf. Section C.2) and the laptop Lenovo ThinkPad T61 (cf. Section C.3) execute the offloading client of the Jikes RVM prototype, representing two heterogeneous mobile devices. The netbook is a resource-poor and energy-efficient device, whereas the laptop is a more powerful device with higher energy consumption. To measure the power consumption of the netbook or of the laptop, we utilized the measurement board described in detail in Subsection 4.6.5. The desktop computer HP Compaq 8200 Elite (cf. Section C.4) executes the offloading service of the Jikes RVM prototype, being the most powerful device and having a power connection. As an underutilized desktop computer in the vicinity executes the offloading service, it does not charge a monetary cost for its utilization (cf. $C_{exec}^{E_{\xi}}$ in Section 3.1). The netbook and the laptop communicate via a 3.5 Generation (3.5G) mobile communication network to the desktop computer with a varying bandwidth of about 1 Mbit/s and a varying latency of round about
100 ms. The Huawei E1750 Surf Stick (cf. Section C.6) provides the link to the mobile communication network. It does not charge a monetary cost for its utilization (cf. \(C_{\text{send}}^{A(\xi_l;\xi_r)}\) and \(C_{\text{recv}}^{A(\xi_l;\xi_r)}\) in Section 3.1), because an end user has today typically a flat-rate data plan. The application evaluated on the netbook and on the laptop is the Chesspresso application (cf. Section B.2), where the end user sets the weight for the time \(w_t\), the energy \(w_e\), and the cost \(w_c\) to 1 (cf. Section 3.1).

**Open Java Development Kit**

For the OpenJDK, the evaluated mobile devices are also the resource-poor netbook and the more powerful laptop (cf. Section C.2 and Section C.3). Both mobile devices execute the offloading client of the OpenJDK prototype for the basic distribution. We also measured the power consumption of the netbook or of the laptop with the measurement board described in detail in Subsection 4.6.5. The desktop computer (cf. Section C.4) executes the offloading service of the OpenJDK prototype being the most powerful device and having a power connection. For the monetary cost charged by the offloading service (cf. \(C_{\text{exec}}^{\Xi(\xi_r)}\) in Section 3.1), we consider the price charged by Amazon\(^5\) for an \(m3\).\textit{medium} machine type from its cloud, causing a monetary cost of 0.070 \$/hr. For the communication network, the netbook and the laptop communicate via a Wi-Fi link to the desktop computer, not charging a monetary cost for its utilization (cf. \(C_{\text{send}}^{A(\xi_l;\xi_r)}\) and \(C_{\text{recv}}^{A(\xi_l;\xi_r)}\) in Section 3.1). The Wi-Fi link has a varying bandwidth of round about 3 Mbit/s and a varying latency of round about 5 ms, provided by the Linksys WRT54GL wireless router (cf. Section C.7). The applications evaluated on the netbook and on the laptop are a chess game (cf. Section B.2) and a text-to-voice application (cf. Section B.2), differing in the computational complexity and the communication required for code offloading. For the user-defined weights, the end user sets the weight \(w_t\) for the time, \(w_e\) for the energy, and \(w_c\) for the cost to 1 (cf. Section 3.1).

**Android Open-Source Project**

For the AOSP, the evaluated mobile device is the Samsung Galaxy Nexus (cf. Section C.1), executing the offloading client of the AOSP prototype. This smart phone is a resource-constrained device with a high energy consumption for the execution of resource-intensive application parts. To measure the power consumption on the smart phone, we utilize the measurement board described in detail in Subsection 4.6.5. A t2.micro instance from the Amazon Web Services (AWS) Elastic Compute Cloud (EC2) (cf. Section C.5) executes the offloading service from the OpenJDK prototype, being

\(^{5}\text{https://aws.amazon.com/ec2/pricing/?nc1=h_ls}\)
4. Efficient Code Offloading with Annotations

the most powerful device. As a result, we consider the price charged by Amazon\textsuperscript{6} for a t2.micro instance from its cloud, causing a monetary cost of 0.012 $ per hour (cf. $C_{\text{exec}}^{\zeta(r)}$ in Section 3.1). For the communication network, the smartphone communicates via a Wi-Fi network and fixed networks to the t2.micro instance, not charging a monetary cost for the utilization (cf. $C_{\text{send}}^{\Lambda(\zeta;\xi)}$ and $C_{\text{recv}}^{\Lambda(\zeta;\xi)}$ in Section 3.1). The networks have a varying bandwidth of about 2.5 Mbit/s and a varying latency of round about 10 ms. The Linksys WRT54GL wireless router provides the Wi-Fi AP to the Internet (cf. Section C.7). The mobile application evaluated on the smartphone consists of resource-friendly and resource-intensive parts, where the resource-friendly parts are graphical user input and output. The resource-intensive parts are a best chess move engine (cf. Section B.3), a human face recognition engine (cf. Section B.4), and a text-to-voice engine (cf. Section B.5), differing in the computational complexity and the communication required for code offloading. Beside the standard applications always started on an Android OS, the end user only starts the implemented application for a better comparison and sets the weight $w_t$ for the time, $w_e$ for the energy, and $w_c$ for the cost to 1 (cf. Section 3.1).

4.7.2. Results

The different implementations of the prototypes for the basic distribution also categorize the evaluation results (cf. Section 4.6), where this subsection describes the results for the prototypes of the Jikes RVM, the OpenJDK, and the AOSP.

Jikes Research Virtual Machine

The netbook consumes in idle mode 8.40 W on average (cf. power consumption at 35 s in Figure 4.13). Executing the Chesspresso application totally on the netbook with an unaltered Jikes RVM, this local execution takes in total 29.542 s and consumes in total 294.07 J, resulting in a power consumption of 9.95 W on average (cf. Figure 4.13). Now, executing the Chesspresso application on the netbook with the Jikes RVM enabled for the basic distribution, it takes in total 16.825 s (cf. Figure 4.13). The remote execution starts at 14.59 s and ends at 18.69 s, taking 4.10 s. The netbook consumes for the basic distribution in total 178.00 J resulting in a power consumption of 10.58 W on average. Thus, an execution of the Chesspresso application on the netbook benefits significantly from the basic distribution, taking 12.717 s less time and consuming 116.07 J less energy. Compared to the power consumption of the netbook, the absolute power consumption on the laptop is in total higher due to the smaller energy efficiency. The laptop

\textsuperscript{6}https://aws.amazon.com/ec2/pricing/?nc1=h_1s
4.7. Evaluation

Figure 4.13.: Power consumption on the netbook for a local execution and the basic distribution of the Chesspresso application.

Figure 4.14.: Power consumption on the laptop for a local execution and the basic distribution of the Chesspresso application.
Consumes in idle mode 27.29 W on average (cf. power consumption at 10 s Figure 4.14). Executing the Chesspresso application totally on the laptop with an unaltered Jikes RVM, this local execution in total takes 8.841 s and consumes 408.97 J, resulting in a power consumption of 46.25 W on average (cf. Figure 4.14). Now, executing the Chesspresso application on the laptop with the Jikes RVM enabled for the basic distribution, it takes in total 6.737 s. The remote execution starts at 2.695 s and ends at 6.963 s taking 4.268 s. The laptop consumes for the basic distribution in total 244.89 J resulting in a power consumption of 36.34 W on average. Thus, an execution of the Chesspresso application on the laptop also benefits from the basic distribution, taking 2.104 s less time and consuming 164.08 J less energy.

Summarizing, the basic distribution increases on both devices the energy efficiency of the devices as well as the execution speed of the Chesspresso application.

OpenJDK

Chess Game: For the chess game, each mobile device plays 10 rounds with the different opening moves from Figure B.2 and a chess difficulty of 2 (cf. Section B.3).

For the netbook, Table 4.1 summarizes the execution time, energy consumption, and monetary cost for a local execution and the basic distribution after 10 rounds of chess moves for each opening move. A local execution of the chess game totally on the netbook takes the longest time and consumes the most energy for every opening move. In detail, the execution time for a local execution is between 134.70 s and 345.03 s with an energy consumption between 1450.38 J and 3715.22 J. Comparing the time taken and the energy consumed of a local execution with the basic distribution (cf. Table 4.1), the basic distribution of the Java method that is offloadable from the chess game reduces both the execution time and the energy consumption significantly. In detail, it requires at least 24.44 s and 241.10 J and at most 45.31 s and 433.51 J. Due to the utilization of the offloading service, it reduces the execution time by up to 86.87% and energy consumption by up to 88.33%, causing monetary cost between 0.05273 $ and 0.12995 $.

For the laptop, Table 4.1 also summarizes the execution time, energy consumption, and monetary cost for a local execution and the basic distribution after 10 rounds of chess moves for each opening move. Overall, the results for the evaluation on the laptop follow the same characteristics as the results for the evaluation on the netbook. Due to the fact that the laptop is a more powerful device with a higher energy consumption compared to the netbook, the laptop halves the time taken and more than duplicates the energy consumed for a local execution of the chess game, still taking the longest
Table 4.1: Overview of execution time, energy consumption, and monetary cost on the netbook and on the laptop for the evaluation of the chess game with different opening moves.
4. Efficient Code Offloading with Annotations

time and consuming the most energy (cf. Table 4.1). In detail, the execution time for a
local execution is between 69.019s and 178.608s with an energy consumption between
3187.30 J and 8248.12 J. Despite the higher performance of the laptop, the benefits
of the basic distribution to the offloading service are still notable (cf. Table 4.1).
Compared to a local execution, the basic distribution reduces the execution time and
the energy consumption by up to 74.87% and 81.58%. It requires at least 23.912s and
890.93 J and at most 44.889s and 1519.67 J with a monetary cost of at least 0.04997 $
and at most 0.12966 $.

Summarizing, the basic distribution increases on the netbook and on the laptop the
energy efficiency of the devices as well as the execution speed of the chess game.

**Text-to-Voice Application:** For the text-to-voice application, each mobile device
transforms either from the beginning the next 15 words ($Idx=0\ Cnt=15$) or from the
word at position 15 the next 20 words ($Idx=15\ Cnt=20$) from ThinkAir or MAUI with
the voice “Alan” to voice (cf. Section B.5).

For the netbook, Table 4.2 shows the execution time, energy consumption, and
monetary cost for a local execution and the basic distribution, transforming 15 or 20
words from ThinkAir or MAUI to voice. A local execution transforms the words to
voice totally on the netbook, taking the longest time (minimum: 65.532s; maximum:
85.024s) and consuming the most energy (minimum: 705.64 J; maximum: 915.52 J)
compared to the basic distribution (cf. Table 4.2). As the text-to-voice application
is on the netbook a good candidate for a distribution, the basic distribution reduces
significantly the execution time and energy consumption by up to 51.48% and 54.00%
(cf. Table 4.2). In detail, it takes at least 32.519s and at most 41.472s, consumes at
least 332.97 J and at most 423.29 J, raises at least 0.03867 $ and at most 0.05560 $.

For the laptop, Table 4.2 shows the execution time, energy consumption, and mon-
eyary cost for a local execution and the basic distribution, transforming the words to
voice. A local execution of the text-to-voice application takes between 26.083s and
33.731s and consumes between 280.86 J and 363.21 J (cf. Table 4.2). Due to the fact
that, on the one hand, the laptop is a more powerful device with a higher energy con-
sumption and, on the other hand, the text-to-voice application has a big size of the
output execution state, a local execution on the laptop is better than a remote execu-
tion on the offloading service. As a result, the basic distribution does not distribute
the Java method that is offloadable of the text-to-voice application. Regarding a naive
basic distribution that still distributes the Java method that is offloadable, it increases
the execution time (minimum: 29.304s; maximum: 38.508s) by up to 16.16%, the
energy consumption (minimum: 298.35 J; maximum: 392.98 J) by up to 10.09%, and
<table>
<thead>
<tr>
<th></th>
<th>ThinkAir:</th>
<th>ThinkAir:</th>
<th>MAUI:</th>
<th>MAUI:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Idx=0 Cnt=15</td>
<td>Idx=15 Cnt=20</td>
<td>Idx=0 Cnt=15</td>
<td>Idx=15 Cnt=20</td>
</tr>
<tr>
<td><strong>Local Execution</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution Time</td>
<td>65.53</td>
<td>84.45</td>
<td>65.93</td>
<td>85.02</td>
</tr>
<tr>
<td>Energy Consumption</td>
<td>705.64</td>
<td>909.34</td>
<td>709.88</td>
<td>915.52</td>
</tr>
<tr>
<td>Monetary Cost</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td><strong>Basic Distribution</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution Time</td>
<td>32.53</td>
<td>41.47</td>
<td>32.52</td>
<td>41.25</td>
</tr>
<tr>
<td>Energy Consumption</td>
<td>332.97</td>
<td>423.29</td>
<td>333.98</td>
<td>421.17</td>
</tr>
<tr>
<td>Monetary Cost</td>
<td>0.04144</td>
<td>0.05560</td>
<td>0.03867</td>
<td>0.05498</td>
</tr>
<tr>
<td><strong>Laptop</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ThinkAir:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Idx=0 Cnt=15</td>
<td>26.08</td>
<td>33.73</td>
<td>26.55</td>
<td>33.15</td>
</tr>
<tr>
<td>Idx=15 Cnt=20</td>
<td>26.55</td>
<td>33.15</td>
<td>26.55</td>
<td>33.15</td>
</tr>
<tr>
<td>Energy Consumption</td>
<td>280.86</td>
<td>363.21</td>
<td>285.89</td>
<td>356.96</td>
</tr>
<tr>
<td>Monetary Cost</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td><strong>Naive Distribution</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution Time</td>
<td>29.30</td>
<td>37.91</td>
<td>29.99</td>
<td>38.51</td>
</tr>
<tr>
<td>Energy Consumption</td>
<td>298.35</td>
<td>385.29</td>
<td>307.03</td>
<td>392.98</td>
</tr>
<tr>
<td>Monetary Cost</td>
<td>0.04107</td>
<td>0.05479</td>
<td>0.03803</td>
<td>0.05178</td>
</tr>
</tbody>
</table>

Table 4.2.: Overview of execution time, energy consumption, and monetary cost on the netbook and on the laptop for the evaluation of the text-to-voice application.
4. Efficient Code Offloading with Annotations

Figure 4.15.: Power consumption on the smart phone for a local execution and the basic distribution of the mobile application.

causes monetary cost between 0.03803 $ and 0.05479 $ compared to a local execution.

Summarizing, the basic distribution increases on the netbook and on the laptop the energy efficiency of the device as well as the execution speed of the chess game. The basic distribution only offloads the Java method that is offloadable from the text-to-voice application on the netbook, benefiting from computation offloading. On the laptop, the basic distribution does not offload the Java method that is offloadable from the text-to-voice application due to its big size of the output execution state. The evaluation results of a naive basic distribution that still offloads the Java method that is offloadable would decrease the energy efficiency of the device as well as the execution speed of the text-to-voice application on the laptop, highlighting the importance of proper decisions for computation offloading.

Android Open-Source Project

Figure 4.15 shows the power consumption on the smart phone for a local execution of the application on its hardware platform in airplane mode. On average, the smart phone consumes 1.107 W in idle mode, 3.612 W for utilizing the cellular link, and 2.412 W during the execution of the resource-intensive application parts. Once an end user starts the application on the smart phone, the smart phone starts the execution of the best chess move engine with configuration 0 of the chess board (cf. Figure B.2i) and a chess difficulty of 2. The best chess move engine takes in total 37.216 s caused by
its high code complexity. After a short period of time for user input and output (0.5 s), the application executes the human face recognition engine with a minimum scale of 1, a maximum scale of 100, and an image size of 1258 pixels times 1024 pixels. It takes 11.431 s caused by its moderate code complexity. After another period of time for user input and output (0.5 s), the application executes the text-to-voice engine that finishes the execution. The text-to-voice engine transforms all words from OWN with the voice “Alan” (cf. Section B.5). The text-to-voice engine takes 18.624 s caused by a higher code complexity than the human face recognition engine but a lower code complexity than the best chess move engine. In total, the local execution of the application on the smart phone takes 68.853 s and consumes 173.159 J.

Figure 4.15 also shows the power consumption on the smart phone for the basic distribution of the application via a Wi-Fi link to the Internet and thus, to an offloading service at the cloud. The Wi-Fi link between the smart phone and the server instance at the cloud has a bandwidth of 1.5 MB/s and a latency of 8.605 ms. As a result, the basic distribution executes the best chess move engine, the human face recognition engine, and the text-to-voice engine on the offloading service, reducing its execution time to 4.208 s, to 7.654 s, and to 2.084 s, respectively. In total, the basic distribution results in an execution time of 15.705 s and an energy consumption of 45.301 J.

Summarizing, the basic distribution increases significantly on the smart phone the energy efficiency of the device as well as the execution speed of the application.

4.8. Summary

To offload computation from a (resource-poor) mobile device to a (powerful) remote resource like a server instance at the cloud, the basic distribution described in Section 4.1 provides an efficient code offloading with annotations. An efficient code offloading uses sparingly resources from both a mobile device and a remote resource relying on least interventions from both an application developer and an end user. To this end, it utilizes an annotation-based instrumentation of portable code, requiring a little application-specific knowledge from an application developer. The system overview described in Section 4.2 specifies the system model, the problem statement, and the system components of the basic distribution between a mobile device and a remote resource (two-tier architecture). As features like a dynamic adaptation, resource efficiency, or a seamless execution are necessarily for the basic distribution, it utilizes a runtime-layer offloading described in Section 4.3. It runs a runtime environment on the hardware platform of participating resources to abstract from actual capabilities and properties from different hardware platforms and software stacks in the landscape.
of Mobile Cloud Computing (MCC). In detail, the offloading client and the offloading service extends the functionality provided from a runtime environment like the Java platform to enable an efficient code offloading based on portable code. The offloading timeline described in Section 4.4 for the basic distribution between an offloading client and an offloading service via a communication network describes each step from an identification of application parts for a remote execution to a receive of an output execution state after a remote execution. Each step described in the timeline from the basic distribution requires specific functionality on the offloading client and on the offloading service provided by the offloading framework described in Section 4.5. The functionality provided by the offloading framework are interfaces for the device manufacturer, end user, and resource provider, benchmark of the hardware, controllers for the distribution and execution, monitors for the network and service, connectors to the client and the service, an application profiler, a state generator, and a compiler. To analyze the interplay of the functionality required for the basic distribution, this dissertation deploys different implementations described in Section 4.6 of a prototype based on the Jikes Research Virtual Machine, the OpenJDK (Java Development Kit), and the Android Open-Source Project. Furthermore, each prototype performs an evaluation described in Section 4.7 of the performance for the basic distribution between a mobile device like a smart phone, netbook, or laptop to a server instance at a desktop or the Amazon cloud. In summary, the evaluation results for the basic distribution of an application like a chess game, a face recognition application, or a text-to-voice application outperform significantly a related execution of the applications locally on the hardware platform of the mobile devices.
Chapter 5

Robust Code Offloading through Safe-point’ing

A mobile device might suffer from crash failures of the communication network that last for longer periods of seconds to minutes during the basic distribution presented in detail in Chapter 4. To make the basic distribution robust against crash failures, this chapter presents a robust code offloading of application parts through safe-point’ing. To this end, Section 5.1 describes a preemptable distribution of application parts between an offloading client on a (resource-poor) mobile device and an offloading service on a (powerful) remote resource. The preemptable distribution enables an offloading client to react on failures by interrupting a remote execution on an offloading service and continuing the remote execution locally based on the execution state of the offloading service. First, Section 5.2 gives the system overview for the preemptable distribution, including the system model, the problem statement, and the system components, before Section 5.3 highlights a related offloading timeline. The basic idea of the preemptable distribution is the creation and transmission of safe-points of a remote execution, where a safe-point conceptually corresponds to a snapshot of the state of a remote execution. The offloading framework described in Section 5.4 enables an offloading client on a mobile device to continue a remote execution locally from the safe-point received most recently, containing the intermediate state of the remote execution on an offloading service. To evaluate the benefit and overhead of the preemptable distribution compared to the basic distribution, Section 5.5 presents the evaluation of a prototype, including the evaluation setup and the evaluation results. Last, Section 5.6 summarizes the main facts of a robust code offloading through safe-point’ing presented in this chapter.

5.1. Preemptable Distribution

The basic distribution – presented in detail in Chapter 4 – of applications parts comprises an offloading client that distributes application parts, an offloading service that executes the distributed application parts, and a (wireless) link that connects both
(communication network). Corresponding to the failure model in Section 3.4, the offloading service as well as the (wireless) link suffer from crash failures. Multiple evaluations in the literature show that mobile connections are error-prone, causing (temporary) disconnections between an offloading client and an offloading service. For instance, Ding et al. [DWC+13] measure the signal strength of a 3G mobile communication network from smart phones with Android OS based on a data set from 3785 volunteers worldwide. The measurements show that over 80% (60%) of the end users have a poor signal strength for over 15% (32%) of their active usage time. As end users of smart phones move frequently in an area with a poor coverage of cellular networks, Ding et al. [DWC+13] identify geographic variations of network coverage as the major reason for temporary disconnections.

Approaches proposed in the literature only consider very limited strategies to handle failures during a remote execution of application parts. Popular strategies either start a local re-execution of an application part or wait for a re-connection to the remote site in case of failures. Both strategies have adverse effects for the efficiency of the energy consumption as well as for the execution time of an application: On the one hand, a distribution plus a re-execution of an application part consume more energy and take more time than a local execution of the application part. Aborting a distribution of an application part due to a failure wastes the energy consumed and the time spent so far for the distribution plus the waiting in idle mode for the end of the remote execution. On the other hand, a distribution plus waiting for a re-connection block an offloading client for an (arbitrary) long time. Moreover, an offloading client again consumes energy and spends time while waiting, possible beyond the point in time, where a local execution would be more efficient.

In order to solve the dilemma of either waiting for a re-connection or starting a local re-execution in case of a failure, we propose the approach of a preemptable distribution. The preemptable distribution enables an offloading client to re-use the intermediate state of a remote execution on an offloading service for a local continuation in case of a failure. To this end, an offloading service creates from time to time the intermediate states for an execution of an application part and transmits these intermediate states to the offloading client. In case of a failure, the offloading client does not need to either wait for a re-connection or start a local re-execution, because it now continues the execution locally based on the intermediate state received most recently. We call such an intermediate state of a remote execution a safe-point, containing the information required on an offloading client for a continuation of the partial remote execution of an application part. The challenge of the preemptable distribution is to determine the right times of creating and transmitting safe-points such that it increases the energy
efficiency on the offloading client and the execution speed of an application. In detail, if the preemptable distribution utilizes too many safe-points, the communication overhead and thus, the energy consumption increases. If it utilizes too few safe-points, the risk of falling back to a very old intermediate state in case of a failure increases ultimately. At worst, it converges into the basic distribution without safe-points, just waiting for a re-connection or starting a local re-execution. The computation complexity of an application part together with the size of a safe-point influence the decision for creating and transmitting a safe-point. For instance, if a local re-execution of an application part is inexpensive with regard to energy, fewer safe-points are sufficient. To this end, the preemptable distribution estimates the energy saved by a safe-point based on online measurements to calculate the right time for safe-points such that it reduces effectively the energy consumption of an offloading client in average.

Summarizing, we make the following contributions: (1) A framework for the preemptable distribution based on safe-points for applications written in the Java programming language; (2) an adaptive algorithm with online measurements that schedules the creation and transmission of safe-points such that it minimizes the energy consumption and increases the robustness under failures; (3) an implementation of the offloading framework for the preemptable distribution on the Java platform; and (4) an evaluation with state of the art mobile devices that shows the efficiency of the preemptable distribution based on safe-points.

### 5.2. System Overview

The preemptable distribution increases the robustness of the basic distribution by re-using intermediate states of a remote execution. Thus, the system model and the system components of the preemptable distribution are same as before (cf. Section 4.2).

Beside the minimization of the cost function $f_w$ for the basic distribution, the preemptable distribution additionally increases the robustness of an application under crash failures. While increasing the robustness of an application, the execution time and energy consumption should stay small in cases with failures. Regarding the cost function $f_w$ of Equation 4.1, the creation and transmission of safe-points influence the execution time $T(A_{\alpha_o}, \Xi(\xi))$, the energy consumption $E(A_{\alpha_o}, \Xi(\xi))$, and the monetary cost $C(A_{\alpha_o}, \Xi(\xi))$ of a remote execution. We assume that safe-point'ing is an additional feature offered by the provider of an offloading service and, for instance, financed by advertising (cf. Google with Android OS), not charging additional monetary cost. However, safe-point'ing increases the execution time $T_{\text{wait}}(A_{\alpha_o}, \Xi(\xi_v))$ on an offloading client due to the time for creating safe-points on an offloading service. It also causes
5. Robust Code Offloading through Safe-point’ing

the execution time \( T_{sfpt}(A_{\alpha o}, \Xi(\xi_r)) \) on an offloading client due to receiving safe-points from an offloading service. Thus, the execution time \( T(A_{\alpha o}, \Xi(\xi)) \) is defined as:

\[
T(A_{\alpha o}, \Xi(\xi)) = T_{remote}(A_{\alpha o}, \Xi(\xi)) = T_{send}(A_{\alpha o}, \Xi(\xi)) + T_{wait}(A_{\alpha o}, \Xi(\xi)) + T_{sfpt}(A_{\alpha o}, \Xi(\xi)) + T_{recv}(A_{\alpha o}, \Xi(\xi))
\]

with

\[
T_{wait}(A_{\alpha o}, \Xi(\xi)) = \frac{P_{exec}(I_{state}(A_{\alpha o}), P_{pcode}(A_{\alpha o}))}{P_{pwr}(\Xi(\xi))} + \sum_s T_{crtn}(SP_i)
\]

\[
T_{sfpt}(A_{\alpha o}, \Xi(\xi)) = \sum_s^N (SP) \left( \frac{SP_{size}(A_{\alpha o})}{E_{down}^{\Xi(\xi)}} + L_{down}^{\Xi(\xi)} \right)
\]

Please note that the time \( T_{crtn}(SP_i) \) spent for creating a safe-point depends on the current execution state of the remote execution and thus, is unknown in advance.

Beside increasing the execution time on an offloading client, safe-point’ing also consumes energy, causing the energy consumption \( E_{sfpt}(A_{\alpha o}, \Xi(\xi)) \) for receiving safe-points. Please note that the energy consumption \( E_{wait}(A_{\alpha o}, \Xi(\xi)) \) includes the increase of the energy consumption caused by creating safe-points on the remote side. Thus, the energy consumption \( E(A_{\alpha o}, \Xi(\xi)) \) is defined as:

\[
E(A_{\alpha o}, \Xi(\xi)) = E_{remote}(A_{\alpha o}, \Xi(\xi)) = E_{send}(A_{\alpha o}, \Xi(\xi)) + E_{wait}(A_{\alpha o}, \Xi(\xi)) + E_{sfpt}(A_{\alpha o}, \Xi(\xi)) + E_{recv}(A_{\alpha o}, \Xi(\xi))
\]

with

\[
E_{sfpt}(A_{\alpha o}, \Xi(\xi)) = T_{sfpt}(A_{\alpha o}, \Xi(\xi)) \cdot E_{recv}(A_{\alpha o}, \Xi(\xi))
\]

As the total number \( N(SP) - 1 \) of created and received safe-points as well as the time for creating each safe-point \( T_{crtn}(SP_i) \) are unknown in advance, \( T_{wait}(A_{\alpha o}, \Xi(\xi)) \), \( T_{sfpt}(A_{\alpha o}, \Xi(\xi)) \), \( E_{wait}(A_{\alpha o}, \Xi(\xi)) \), and \( E_{sfpt}(A_{\alpha o}, \Xi(\xi)) \) are also unknown in advance.

5.3. Offloading Timeline

Figure 5.1a shows the preemptable distribution with safe-point’ing of an application part between an offloading client and an offloading service, where no failures occur during the remote execution. Compared to the basic distribution of an application
5.3. Offloading Timeline

(a) No failures during the remote execution.

(b) Failures during the remote execution.

Figure 5.1.: The preemptable distribution of an application part between an offloading client and an offloading service via a communication network, where (a) no failures occur during the remote execution and (b) failures occur during the remote execution, where the offloading client continues the execution locally.

Part (cf. Section 4.4), an offloading service creates and transmits safe-points to an offloading client during an execution of an application part. In detail, in case of a remote execution, the offload controller invokes a safe-point generator to create a safe-point on the offloading client (cf. Subsection 5.4.1). A safe-point includes the parameters of the method, all reachable values of heap objects, declared class member objects, and static objects, corresponding to the input execution state of the basic distribution. After the creation of a safe-point on the safe-point generator, the offload controller forwards the created safe-point to the service connector and waits for the output execution state. The service connector creates the offload request that contains among others the safe-point and sends it to the offloading service.

Due to the receive of an offload request, the offloading service installs the contained information and calls the Java method, starting a remote execution just like for the basic distribution. To execute the Java method, the Java Runtime Environment (JRE) invokes the offload compiler to translate the platform-independent Java bytecode instructions into platform-dependent machine code. During the translation of Java bytecode instructions into machine code, the offload compiler instruments the machine code with breakpoints to interrupt the execution of the Java method at certain points detailed below. The introduced breakpoints invokes the execution controller that again invokes the offload controller. Due to the invocation of the offload controller, it decides whether to create a safe-point at this point in time and transmit it to the offloading client. During the execution of the Java method on the offloading service, the offload
controller might create several safe-points, but not transmitting each. It only transmits a safe-point to the offloading client if the energy consumption on the offloading client to locally reach the execution state contained in a safe-point is higher than a transmission of the safe-point from the offloading service to the offloading client.

While the offloading service executes the Java method, the offloading client runs a failure handler. The failure handler decides whether to preempt the distribution and continue the remote execution on the offloading service locally based on the safe-point received most recently (cf. Figure 5.1b). To this end, the failure handler on the offloading client monitors via the service connector the link to the offloading service as well as the remote service itself (cf. Subsection 4.5.2). In case of the failure handler detects a connection failure or a node failure, it invokes the offload controller to continue the remote execution locally on the offloading client.

5.4. Offloading Framework

The preemptable distribution extends the functionality provided from the offloading framework of the basic distribution (cf. Section 4.5), adapting the offloading client (cf. Subsection 5.4.1) and the offloading service (cf. Subsection 5.4.2).

5.4.1. Offloading Client

Figure 5.2 shows the runtime environment on the offloading client that enables the preemptable distribution with safe-point’ing. Compared to the runtime environment of the basic distribution (cf. Subsection 4.5.2), the changes comprise the offload controller and the state generator – now named safe-point generator – and a failure handler.

**Offload Controller:** The execution controller on the offloading client invokes the offload controller due to the execution of the Java bytecode instruction `offload`, marking the call of the Java method as feasible for a distribution. To this end, the offload controller on the offloading client invokes the safe-point generator to create a safe-point of the current execution state. The generated safe-point corresponds to the input execution state of the basic distribution. Thus, the safe-point generator forwards it after the creation to the offload controller that again forwards it to the service connector just like for the basic distribution. The failure handler on the offloading client invokes the offload controller due to the occurrence of a connection failure to or a node failure of the offloading service. To this end, the offload controller invokes the safe-point generator to retrieve the execution state last known of the remote execution. The
5.4. Offloading Framework

Figure 5.2.: Overview of the runtime environment on the offloading client enabling the preemptable distribution based on safe-point’ing with an adapted offload controller and an added safe-point generator and failure handler.

The safe-point generator generates the execution state last known based on the safe-point created locally (input execution state) together with the safe-point(s) received from the offloading service. Retrieving the execution state last known, the offload controller forwards it to the execution controller that installs it and continues the execution of the remote execution locally.

Safe-point Generator: The safe-point generator on the offloading client extends the state generator from the basic distribution (cf. Subsection 5.4.1) by providing functionality for safe-point’ing. It creates the first safe-point before the execution of a Java method that is offloadable. This first safe-point corresponds to the input execution state of the basic distribution that contains all of the information required for a remote execution. Furthermore, the safe-point generator is responsible to generate the execution state last known of the remote execution on the offloading service. To this end, it combines all incremental safe-points received from the offloading service into a single safe-point. This combined safe-point contains the execution state last known from a remote execution like all variables required and modified during a remote execution.

Failure Handler: The failure handler on the offloading client decides whether to preempt the distribution of the Java method and continue its remote execution on the
5. Robust Code Offloading through Safe-point’ing

Figure 5.3.: Overview of the runtime environment on the offloading service enabling the preemptable distribution based on safe-point’ing with an adapted offload compiler, execution controller, and offload controller and an added safe-point generator.

Offloading service locally based on the execution state last known. To this end, it monitors via the service connector the link to the offloading service as well as the remote service itself by sending periodically heartbeat messages. Due to the periodically pinging of the offloading service, the failure handler invokes the offload controller in case of receiving no heartbeat message after a timeout period. Receiving no heartbeat message after a timeout period corresponds to the occurrence of a failure.

5.4.2. Offloading Service

Figure 5.3 shows the runtime environment on the offloading service that enables the preemptable distribution with safe-point’ing. Compared to the runtime environment of the basic distribution (cf. Subsection 4.5.3), the changes comprise the offload compiler, the execution controller, the offload controller, and the safe-point generator.

Offload Compiler: The offload compiler on the offloading service inserts breakpoints at each Java bytecode instruction, where an execution of the Java bytecode instruction branches an execution of a Java method. Java bytecode instructions that branch an execution of a Java method are goto, goto_w, jsr, jsr_w, ret, tableswitch, and
5.4. Offloading Framework

lookupswitch (cf. Appendix A). These Java bytecode instructions are particularly well-suited for safe-pointing, because the execution state at these branching instructions has a small footprint of memory. As variables with a local scope – e.g., inside the body of a loop – are typically not visible after the branch, a safe-point does not include such variables with a local scope and thus, has a small footprint of memory.

Furthermore, the offload compiler on the offloading service inserts additional Assembler code to efficiently keep track of modifications to the Java stack, the Java heap, and static Java objects during an execution of a Java method (cf. Subsection 4.3.4). To track modifications during an execution, the offload compiler inserts the additional Assembler code at each Java bytecode instruction that modifies a value of the Java stack, the Java heap, or a static Java object. Java bytecode instructions that change a value of the Java stack are istore, lstore, fstore, dstore, astore, istore_0, istore_1, istore_2, istore_3, lstore_0, lstore_1, lstore_2, lstore_3, fstore_0, fstore_1, fstore_2, fstore_3, dstore_0, dstore_1, dstore_2, dstore_3, astore_0, astore_1, astore_2, astore_3, iastore, lastore, fastore, dastore, aastore, bastore, castore, and sastore (cf. Appendix A). The Java bytecode instruction that changes a value of the Java heap is putfield and the Java bytecode instruction that changes a value of a static Java object is putstatic (cf. Appendix A). At execution, the inserted Assembler code flips a certain Bit in the Java header of the corresponding Java object, where a Bit flipped in the Java header indicates the modification of a Java object during an execution.

Execution Controller: Each time the execution of a Java method reaches the additional Assembler code inserted by the offload compiler for Java bytecode instructions that branches an execution of a Java method, the inserted Assembler code invokes the execution controller on the offloading service. On the one hand, the execution controller counts the number of Assembler code executed since the safe-point created last and, on the other hand, invokes the offload controller to create and transmit a safe-point to the offloading client at the current state of an execution.

Offload Controller: The execution controller on the offloading service invokes the offload controller due to the execution of the additional Assembler code inserted by the offload compiler for Java bytecode instructions that branches an execution of a Java method. The offload controller decides whether to create a safe-point at this point in time and transmit it to the offloading client by evaluating the energy savings for the offloading client through receiving a safe-point.

In order to evaluate the energy savings of a safe-point, the offload controller estimates
5. Robust Code Offloading through Safe-point ing

the energy consumption on the offloading client to reach the execution state contained in the safe-point through a local execution. To this end, it estimates the execution time required on the offloading client to execute the Assembler code counted since the last safe-point by the execution controller. Multiplying the estimated execution time with the energy factor for executing portable code (cf. Section 3.1), the offload controller calculates the energy consumption spent on the offloading client for this partial execution. In detail, the estimation of the energy consumption for a local execution includes the following parameters: (1) The number \( P_{asm}^{exe}(S_i(A_{\alpha_o}), P_{pcode}(A_{\alpha_o})) \) with \( S_0(A_{\alpha_o}) = I_{state}(A_{\alpha_o}) \) of Assembler code executed since the last safe-point to reach the execution state contained in the current safe-point. The offload controller obtains the number \( P_{asm}^{exe}(S_i(A_{\alpha_o}), P_{pcode}(A_{\alpha_o})) \) from the execution controller that counts the number of executed Assembler code based on the additional Assembler code inserted by the offload compiler. (2) The performance characteristic \( \Xi(\xi_k) \) that indicates how many Assembler code the offloading client \( \Xi(\xi_k) \) executes per second. (3) The performance characteristic \( E_{exec}(asm) \) of the offloading client \( \Xi(\xi_k) \) that indicates an energy factor of how much energy the execution of Assembler code consumes per second. Please note that an offload request contains both performance characteristics of the offloading client. Thus, the energy consumption for a local execution \( E_{exec}(S_i(A_{\alpha_o})) \) is defined as:

\[
E_{exec}(S_i(A_{\alpha_o}), P_{pcode}(A_{\alpha_o})) = \frac{P_{asm}^{exe}(S_i(A_{\alpha_o}), P_{pcode}(A_{\alpha_o}))}{\Xi(\xi_k)} \cdot E_{exec}(asm) \tag{5.1}
\]

Besides the estimation of the energy consumption on the offloading client for a local execution, the offload controller on the offloading service also estimates the energy consumption for a transmission of the safe-point to the offloading client. To this end, it invokes the safe-point generator to create a safe-point at the current state of the execution. Based on the created safe-point, the offload controller determines the size of it and multiplies the size with the energy factor for receiving bytes (cf. Section 3.1). In detail, the estimation of the energy consumption for a transmission includes the following parameters: (1) The size \( S_i^{size} \) in bytes of the safe-point. In order to calculate the size \( S_i^{size} \) of a safe-point, the safe-point generator creates a safe-point that the offloading service might not sent if it is not beneficial for the offloading client. (2) The performance characteristic \( B_{down}^{\Xi(\xi_k;\xi_m)} \) and \( L_{down}^{\Xi(\xi_k;\xi_m)} \) for a link \( \Lambda(\xi_k;\xi_m) \) between the offloading service \( \Xi(\xi_m) \) and the offloading client \( \Xi(\xi_k) \) that indicates the number of bytes the link transmits per second and the network delay of the link in seconds, respectively (cf. Section 3.1). (3) The performance characteristic \( E_{recv}(asm) \) that indicates an energy factor for receiving bytes on the offloading client (cf. Section 3.1). Please note
that an offload request contains the three performance characteristics of the offloading client. Thus, the energy consumption for a transmission $E_{\text{trnsmt}}(S_i(A_{\alpha}))$ is defined as:

$$E_{\text{trnsmt}}(S_i(A_{\alpha})) = \left( \frac{S_i}{P_{\text{ppwr}}(\text{asm})} + L_{\text{down}} \right) \cdot E_{\text{recv}}$$ (5.2)

Finally, the offload controller only sends the created safe-point to the offloading client if the transmission of the safe-point saves energy on the offloading client compared to a local execution: $E_{\text{exec}}(S_i(A_{\alpha}), P_{\text{pcode}}(A_{\alpha})) > E_{\text{trnsmt}}(S_i(A_{\alpha}))$. Therefore, the offloading service ensures for a safe-point that the energy consumption for a transmission is lower than for a local execution.

Algorithm 5.1 summarizes the decision making for safe-points on the offload controller. Keeping the overhead on the offloading service low, the offload controller only generates a new safe-point if $P_{\text{exe asm}}(S_i(A_{\alpha}), P_{\text{pcode}}(A_{\alpha}))$ is greater than a time threshold $T_{\text{exe thrshld}}$ (cf. Line 3). As a result, the time threshold $T_{\text{exe thrshld}}$ defines the rate at which the offloading service generates potential safe-points for a transmission, avoiding a too-frequent generation of safe-points for a code sequence with closely-spaced breakpoints. Furthermore, it only sends a safe-point if the sum of the energy consumption for a transmission $E_{\text{trnsmt}}(S_i(A_{\alpha}))$ plus an energy threshold $E_{\text{trnsmt thrshld}}$ is smaller than for a local execution $E_{\text{exec}}(S_i(A_{\alpha}), P_{\text{pcode}}(A_{\alpha}))$ (cf. Line 9). Please note that the energy threshold $E_{\text{trnsmt thrshld}}$ functions as a parameter for the accuracy of the estimations to include, for instance, bandwidth or latency variations in the network link.

Last, the execution controller on the offloading service invokes the offload controller due to the execution of the Assembler code for the Java bytecode instruction `offload_end`. To this end, the offload controller always invokes the safe-point generator to create a last safe-point due to the end of the remote execution of the Java method. The offload controller forwards the safe-point created last to the client connector that transmits it to the offloading client.

**Safe-point Generator:** The safe-point generator on the offloading service is responsible for an efficient creation of safe-points. Thus, the safe-point generator utilizes its own serialization and deserialization of Java objects. Utilizing its own serialization and deserialization, it does not require from an application developer the implementation of an individual serialization and deserialization for each Java class based on the Java interface `Serializable`. Not relying on the Java interface `Serializable` keeps the generation of safe-points from Java classes transparent to application developers. To this end, the safe-point generator implements a memory module for the Java Virtual
5. Robust Code Offloading through Safe-point’ing

Constant: \( JBC_{\text{brnch}} \) # Branching Java ByteCode (JBC) instructions
Constant: \( JBC_{\text{ofldnd}} \) # Java bytecode instruction offload_end
Input: \( JBC_{\text{nxt}} \) # Java bytecode instruction executed NeXT
Input: \( T_{\text{thrshld}} \) # Current time threshold, initialized to \( T_{\text{thrshld}} \)

```
1: if \( JBC_{\text{nxt}} \in JBC_{\text{brnch}} \) then
2: \( t_{\text{tmp}} := \frac{E_{\text{exec}}(S_i(A_{\alpha_o}),P_{\text{pcode}}(A_{\alpha_o}))}{P_{\text{power}}(\text{asm})}; \)
3: if \( t_{\text{tmp}} > T_{\text{cur}} \) then
4: \( T_{\text{thrshld}} := T_{\text{thrshld}} + T_{\text{exe}}; \)
5: \( S_i := \text{gnrtSfPnt}(); \) # Generate a safe-point ...
6: \( S_{i\text{size}} := \text{sizeof}(S_i); \) # ... and determine the size of it
7: \( E_{\text{transmt}}(S_i(A_{\alpha_o})) := \left( \frac{S_{i\text{size}}}{P_{\text{down}}(\xi_k)} + L_{\text{down}}^A(\xi_k - \xi_m) \right) \cdot E_{\text{recv}}(\xi_k); \)
8: \( E_{\text{exec}}(S_i(A_{\alpha_o}),P_{\text{pcode}}(A_{\alpha_o})) := t_{\text{tmp}} \cdot E_{\text{exec}}(\text{asm}); \)
9: if \( E_{\text{exec}}(S_i(A_{\alpha_o}),P_{\text{pcode}}(A_{\alpha_o})) > \left( E_{\text{transmt}}(S_i(A_{\alpha_o})) + E_{\text{thrshld}} \right) \) then
10: \( \text{sndSfPnt}(S_i); \) # Send safe-point to the offloading client
11: \( T_{\text{thrshld}} := T_{\text{exe}}; \)
12: end if
13: end if
14: else if \( JBC_{\text{nxt}} == JBC_{\text{ofldnd}} \) then
15: \( S_i := \text{gnrtSfPnt}(); \) # Generate last safe-point ...
16: \( \text{sndSfPnt}(S_i); \) # ... and send it to the offloading client
17: end if
```

Algorithm 5.1: Algorithm of the decision making for safe-point’ing from the offload controller on the offloading service.
Machine (JVM) that transforms Java primitives, objects, and arrays into a platform-independent representation and vice versa (cf. Subsection 4.3.4). In order to minimize the size of a safe-point, the safe-point generator only serializes objects required and modified since the safe-point transmitted last (incremental safe-point'ing). It identifies the required and modified Java objects by visiting all Java objects that are reachable from an execution of a Java method and inspects their flag of modification. The set of reachable objects includes all visible static objects, members from the declaring class of the method, local values on the stack, and accessible objects on the heap (cf. Subsection 4.3.4). As the offload compiler inserts additional Assembler code that flips a Bit in the Java header on a modification of a related Java object, the offloading service only monitors modifications at the level of Java bytecode instructions. In detail, it cannot detect modifications outside the JVM, for instance, due to an execution of a native method. However, the tracking of modifications based on Bit flipping keeps the overhead introduced to the runtime environment low.

5.5. Evaluation

To evaluate the performance of the preemptable distribution compared to the basic distribution, a mobile device executes a Java application on an unaltered JRE (local execution), on an offload-aware JRE (basic distribution), and on a preemptable-aware JRE (preemptable distribution). To this end, we extended the implementation of the Jikes RVM prototype for the basic distribution (cf. Subsection 4.6.1) to provide the functionality required for the preemptable distribution. The extensions to the Jikes RVM prototype comprise changes to the implementation of the JVM to provide the functionality required for the preemptable distribution on the offloading client and on the offloading service. Next, Subsection 5.5.1 describes the evaluation setup before Subsection 5.5.2 presents the evaluation results.

5.5.1. Setup

The evaluation setup for the preemptable distribution extends the evaluation setup for the basic distribution on the Jikes RVM described in Subsection 4.7.1 as follows:

The evaluation compares the preemptable distribution with two basic approaches found in the literature (cf. Section 9.2). Both approaches provide a non-preemptable distribution based on different strategies at the occurrence of a failure. The first approach – named Basic Distribution with a Re-Execution (BDwRE) – starts a re-execution of the Java method on the offloading client as soon as the failure occurs. As
5. Robust Code Offloading through Safe-point’ing

only the input execution state is available at the occurrence of a failure, the offloading client must re-execute the complete Java method locally again. The second approach – named Basic Distribution waiting for a Re-Connection (BDwRC) – waits for a re-connection to the offloading service at the occurrence of a failure to receive the last safe-point of a remote execution from the offloading service.

5.5.2. Results

Now, this subsection describes the results for the evaluation of the preemptable distribution. As the creation and transmission of safe-points introduce some overhead, the evaluation compares first the different approaches in a scenario, where no failures occur during a distribution of a Java method. Afterwards, it compares the different approaches in scenarios, where failures occur during a distribution at different points in time, highlighting the benefits of the creation and transmission of safe-points.

Distributing the Java method without failures: The creation and transmission of safe-points for the preemptable distribution introduces a certain overhead that only pays off in case of the occurrence of failures. Thus, for a distribution of the Java method without the occurrence of a failure, we expect that the preemptable distribution is less efficient due to the usage of safe-points than the other approaches not using safe-points. Note that in a scenario without failures, the approaches BDwRE and BDwRC behave identically and perform like the basic distribution, because the only difference of the approaches is the reaction on failures. To evaluate the overhead introduced by safe-point’ing, we compare the preemptable distribution with the basic distribution – corresponding to BDwRE and BDwRC – and both approaches with a local execution in a scenario without failures. Ideally, the overhead introduced by safe-point’ing does not affect the overall performance of the preemptable distribution significantly.

To give an insight of the introduced overhead, we start with the comparison of the different approaches on the netbook, where Subsection 4.7.2 describes the evaluation results for a local execution (29.542s; 294.07J) and the basic distribution (16.825s; 178.00J) of the Chesspresso application on the Jikes RVM. To this end, Figure 5.4 shows the power consumption on the netbook for the preemptable distribution of the Chesspresso application. It takes in total 16.746s, where the remote execution takes 4.500s and the netbook receives five safe-points. During the preemptable distribution, the netbook consumes in total 181.45J resulting in a power consumption of 10.84W on average. Thus, the preemptable distribution and basic distribution perform similar for the scenario without the occurrence of failures, virtually resulting in the same
5.5. Evaluation

Now, we compare the different approaches on the laptop, giving an insight of the introduced overhead, where Subsection 4.7.2 also describes the evaluation results for a local execution (8.841s; 408.97 J) and the basic distribution (6.737s; 244.89 J) of the Chesspresso application on the Jikes RVM. To this end, Figure 5.5 shows the power consumption on the laptop for a preemptable distribution of the Chesspresso application. It takes in total 6.914s, where the remote execution takes 4.428s and the laptop receives five safe-points. During the preemptable distribution, the laptop consumes in total 253.85 J resulting in a power consumption of 36.71 W on average. Thus, the preemptable distribution and basic distribution also perform on the laptop similar for the scenario without the occurrence of failures, virtually resulting in the same execution time and power consumption. The relative difference of the preemptable distribution compared to the basic distribution for the consumed energy is only 3.66%.

Summarizing, the relative difference of the basic distribution and the preemptable distribution is very small in a scenario without the occurrence of failures. The reason for the small overhead introduced by safe-point’ing is primarily the small number and size of received safe-points (cf. Algorithm 5.1). As a result, the preemptable distribution still saves a lot of energy compared to a local execution of the Chesspresso application.
5. Robust Code Offloading through Safe-point’ing

Distributing the Java method with failures: The creation and transmission of safe-points for the preemptable distribution pay off in case of the occurrence of failures, where we consider now different scenarios with the occurrence of failures. Thus, for a distribution of the Java method with the occurrence of failures, we expect the preemptable distribution to be more efficient due to the usage of safe-points than the other approaches not using safe-points. To evaluate the benefits of the preemptable distribution with safe-points, we compare all approaches in multiple scenarios, where disconnections happen at different points in time.

As BDwRC waits until a re-connection in case of a failure, it performs equally than the basic distribution described in Subsection 4.7.2 if a disconnection together with a re-connection occurs before the end of a remote execution. In such a scenario, BDwRC and the preemptable distribution receive the last safe-point at the end of the remote execution from the offloading service. Receiving the last safe-point from the offloading service, BDwRC (basic distribution) performs better than the preemptable distribution, because the preemptable distribution continues the remote execution locally based on the safe-point received last and stops the continuation in case of a re-connection. Depending on the point in time the failure occurs, the duration of the disconnection increases the energy consumption but not the execution time of the preemptable dis-
5.5. Evaluation

(a) Failure occurs at $t_e = 16.14$ s with 0 safe-points received.

(b) Failure occurs at $t_e = 16.31$ s with 1 safe-point received.

(c) Failure occurs at $t_e = 17.31$ s with 2 safe-points received.

Figure 5.6: Power consumption on the netbook for a remote execution of the Chesspresso application based on the preemptable distribution, where failures occur at different points in time ($t_e = 16.14$ s, 16.31 s, and 17.31 s) during the remote execution.
5. Robust Code Offloading through Safe-point’ing

(a) Failure occurs at $t_e = 18.54$ s with 3 safe-points received.

(b) Failure occurs at $t_e = 17.97$ s with 3 safe-points received.

(c) Failure occurs at $t_e = 19.02$ s with 4 safe-points received.

Figure 5.7.: Power consumption on the netbook for a remote execution of the Chesspresso application based on the preemptable distribution, where failures occur at different points in time ($t_e = 18.54$ s, 17.97 s, and 19.02 s) during the remote execution.
5.5. Evaluation

For instance, in a worst-case scenario for the preemptable distribution on the netbook, a disconnection occurs at 14.60s and a re-connection at 18.60s. **BDwRC** (basic distribution) consumes in total 178.00 J, whereas the preemptable distribution consumes in total 221.25 J instead of 181.45 J. The same applies to a disconnection before the end of a remote execution with a re-connection after the end of a remote execution but before the end of a local continuation (preemptable distribution). Depending on the point in time of the re-connection, the continuation of the remote execution on the netbook increases the energy consumption of the preemptable distribution but not the execution time. Actually, the point in time of the re-connection determines whether **BDwRC** and the preemptable distribution stay below the time for a local execution. In contrast to **BDwRC**, the duration of the local continuation and thus, the number of received safe-points determine for the preemptable distribution whether it stays below the time for a local execution. For instance, if the local continuation finishes before 30.00s and the disconnection is still long in coming, the preemptable distribution stays below the time for a local execution instead of **BDwRC**. To this end, Figure 5.6 and Figure 5.7 show in total six scenarios, where a disconnection occurs at 16.14s, 16.31s, 17.31s, 18.54s, 17.97s, and 19.02s and the offloading client receives 0, 1, 2, 3, 3, and 4 safe-points. Regarding the worst-case scenarios for the preemptable distribution, a re-connection happens just before the local continuation finishes at 45.63s, 42.55s, 39.15s, 36.02s, 30.55s, and 25.14s, where it consumes 81.68J, 76.43J, 62.42J, 51.65J, 39.34J, and 21.55J more energy compared to **BDwRC**.

**BDwRE** starts a re-execution in case of a failure, where only failures are interesting that occur before the remote execution finishes and last for a longer period. Otherwise, **BDwRE** and the preemptable distribution perform similar apart from safe-point’ing, because both start a local continuation – **BDwRE** from the start and the preemptable distribution from the safe-point received last – and receive the last safe-point at the end of the remote execution. Regarding the scenario with a disconnection at 16.14s (cf. Figure 5.6a), the preemptable distribution does not receive a safe-point due to the short time of the remote execution. Thus, **BDwRE** and the preemptable distribution perform equally, taking 42.23s and consuming 467.34 J. For a disconnection at 16.31s, the preemptable distribution only generates one intermediate safe-point before it continues the execution locally. It takes 39.29s and consumes 437.83 J compared to 42.54s and 434.41 J for **BDwRE**. Due to the short period of the remote execution of 2.3s, the preemptable distribution consumes still more energy than **BDwRE** or a local execution (294.07 J). The preemptable distribution generates more safe-points due to a longer period of the remote execution if the disconnection occurs later. Thus, the energy efficiency of the preemptable distribution further increases while the execution
5. Robust Code Offloading through Safe-point’ing

Figure 5.8.: Power consumption on the laptop for a remote execution of the Chesspresso application based on the preemptable distribution, where a failure occurs at \( t_e = 5.424 \) s with 1 safe-point received.

Time further decreases. For a disconnection at 17.31 s, it generates two intermediate safe-points before it continues the execution locally. Based on the two intermediate safe-points, the execution time as well as the energy consumption of the preemptable distribution are smaller than BDwRE (36.03 s vs. 43.68 s and 399.59 J vs. 447.14 J), but are higher than a local execution (36.03 s vs. 29.54 s and 399.59 J vs. 294.07 J). Generating four intermediate safe-points for a disconnection at 19.02 s, the preemptable distribution now performs better than a local execution with an execution time of 22.36 s vs. 29.54 s and an energy consumption of 246.14 J vs. 294.07 J. Compared to BDwRE, the preemptable distribution reduces the execution time by 23.34 s and the energy consumption by 220.47 J.

On the laptop, the same applies for BDwRC and BDwRE as for the netbook, described above. In case of a disconnection together with a re-connection before the end of a remote execution, the only difference between BDwRC and the preemptable distribution is the local continuation of the remote execution of the preemptable distribution. As a result, the execution time of BDwRC and the preemptable distribution is equal, whereas the energy consumption of the preemptable distribution is higher due to the local continuation. For instance, in a worst-case scenario for the preemptable distribution, a disconnection occurs at 2.90 s and a re-connection at 4.40 s, where
5.5. Evaluation

Figure 5.9.: Power consumption on the laptop for a remote execution of the Chesspresso application based on the preemptable distribution, where a failure occurs at $t_e = 6.146 \text{s}$ with 2 safe-points received.

Figure 5.10.: Power consumption on the laptop for a remote execution of the Chesspresso application based on the preemptable distribution, where a failure occurs at $t_e = 6.749 \text{s}$ with 3 safe-points received.
5. Robust Code Offloading through Safe-point'ing

\(BDwRC\) consumes 244.89 J and the preemptable distribution 282.35 J (local execution: 8.841 s and 408.97 J). In case of a disconnection before and a re-connection after the end of a remote execution, the worst-case scenarios for the preemptable distribution against \(BDwRC\) contain a re-connection just before the local continuation finishes (cf. Figure 5.8 - 5.10). In detail, a re-connection happens just before the local continuation finishes at 11.87 s, 9.93 s, and 8.51 s, where the preemptable distribution consumes 135.25 J, 80.33 J, and 37.42 J more energy.

To this end, we concentrate on disconnections that occur before the remote execution ends and last for a longer period. Figure 5.8, Figure 5.9, and Figure 5.10 show three scenarios, where a disconnection occur at 5.42 s, 6.16 s, and 6.75 s and the offloading client receives 1, 2, and 3 safe-points. In general, the results for the laptop follow the same trend as for the netbook. For short periods of a remote execution until the disconnection, the preemptable distribution is more energy efficient than \(BDwRE\). Actually, the preemptable distribution generates one intermediate safe-point for a disconnection at 5.42 s that already leads to a smaller execution time and energy consumption than \(BDwRE\) (11.46 s vs. 12.56 s and 501.85 J vs. 513.12 J). Compared to a local execution (8.841 s and 408.97 J), the execution time and energy consumption of the preemptable distribution is higher. Again, longer periods of a remote execution pronounce more the savings of the energy consumption and the execution time. For a disconnection at 6.75 s, the preemptable distribution generates three safe-points resulting in an execution time of 8.14 s and an energy consumption of 318.85 J. Thus, it performs better than a local execution and \(BDwRE\) with an execution time of 8.841 s and 13.88 s and an energy consumption of 408.97 J and 533.65 J, respectively.

Summarizing, for scenarios with the occurrence of failures, the preemptable distribution with safe-points reduces the execution time on the netbook and on the laptop. The reduction of the execution time results from the creation and transmission of safe-points together with a continuation of the local execution. Furthermore, the preemptable distribution stays below the time for a local execution, whereas the two basic approaches \(BDwRC\) and \(BDwRE\) do not meet it.

5.6. Summary

To provide a robust code offloading through safe-point’ing, the preemptable distribution described in Section 5.1 extends the basic distribution described in Section 4.1. The preemptable distribution re-uses – in case of a failure – the intermediate state of a remote execution so far without abandoning the complete progress executed remotely. As a result, it increases the energy efficiency and responsiveness of the basic distribution.
5.6. Summary

under failures. The system overview described in Section 5.2 with the system model, problem statement, and system components corresponds almost to the system overview of the basic distribution, considering safe-point’ing in the problem statement. The basic idea of the preemptable distribution is the interruption and local continuation of a current distribution in case of a failure without losing the partial result calculated remotely so far and avoiding a local re-execution of the complete application part, described in the offloading timeline in Section 5.3. A safe-point contains from a remote execution all of the information required to continue a remote execution locally on the offloading client. Thus, an offloading service creates and transmits safe-points to the offloading client during a remote execution whenever it is beneficial in terms of energy consumption. The offloading framework described in Section 5.4 provides the additional functionality required for the preemptable distribution on an offloading client and an offloading service. Compared to the offloading framework for the basic distribution (cf. Section 4.5), the changes are mainly on the offload compiler, offload controller, state generator, and failure handler. To evaluate the introduced overhead and gained benefit by the preemptable distribution compared to the basic distribution with common mechanism to handle failures, the evaluation described in Section 5.5 presents the evaluation setup and the evaluation results. In detail, a prototype based on the Jikes RVM runs in multiple scenarios, where failures occur at different point in times. Although the creation and transmission of safe-points introduce communication and energy overhead, the evaluation results show that this overhead quickly pays off. Especially, in scenarios with failures, safe-point’ing leads to a lower energy consumption on the evaluated mobile device like a netbook and a laptop. Moreover, the utilization of safe-points for the preemptable distribution also improves the responsiveness of an application by staying below the corresponding time for a local execution under failures.
Chapter 6
Deadline-aware Code Offloading with Predictive Safe-point’ing

The preemptable distribution presented in detail in Chapter 5 improves the efficiency of the basic distribution under failures through the utilization of safe-points. However, the scheduling strategy that determines the point in times for the creation and transmission of a safe-point is not always optimal with regard to efficiency. To improve the efficiency of the preemptable distribution, this chapter presents a predictive distribution described in Section 6.1 that optimizes the schedule of creating and transmitting safe-points. The system overview in Section 6.2 describes the system model, problem statement, and system components for the predictive distribution. On the one hand, the predictive distribution tries to avoid the creation and transmission of safe-points that – considered afterwards – are not needed. In the optimal case, an offloading client only receives one safe-point just before the occurrence of a failure. On the other hand, the predictive distribution ensures a minimal responsiveness of an application execution by adding an additional constraint for the execution time to meet a predefined deadline for the application execution. To this end, the predictive distribution improves the preemptable distribution with an optimal schedule for safe-point’ing described in Section 6.3 by predicting the link connectivity and the remaining runtime. To evaluate the performance of the predictive distribution compared to the preemptable distribution, Section 6.4 presents the evaluation of a simulation, including the evaluation setup and the evaluation results. Last, Section 6.5 summarizes the main facts for a deadline-aware code offloading with predictive safe-point’ing presented in this chapter.

6.1. Predictive Distribution

The approach of the preemptable distribution based on safe-points – described in Chapter 5 – shows the benefits of safe-point’ing the intermediate states of a remote execution. The preemptable distribution uses a scheduling strategy of creating and transmitting
6. Deadline-aware Code Offloading with Predictive Safe-point’ing

safe-points that is not always optimal with regard to the efficiency of the energy consumed on an offloading client. To improve the basic approach of the preemptable distribution based on safe-points, we propose an optimized schedule of creating and transmitting safe-points based on the following two improvements: First, the optimized schedule tries to avoid the creation and transmission of safe-points that – considered afterwards – are not needed on the offloading client. For instance, receiving two safe-points on the offloading client, the receive of the first safe-point is needless, because no failure occurred in the meanwhile. In the optimal case, an offloading client only receives during a remote execution one safe-point to continue the execution locally just before the occurrence of a failure like a disconnection. Second, the optimized schedule ensures a minimal responsiveness of an application execution by adding an additional constraint for the execution time to meet a predefined deadline for the application execution. These two improvements related to the preemptable approach comprise the design of an adaptive algorithm for scheduling safe-points that adapts dynamically the point in time of creating and transmitting safe-points due to dynamic conditions of the communication network. On the one hand, the optimized schedule creates and transmits fewer safe-points for a communication network where communication failures are less frequent. On the other hand, it creates and transmits more safe-points for a communication network where communication failures are more frequent, being prepared for impending failures. To this end, the improved approach solves a constrained optimization problem. The problem minimizes the overhead for safe-point’ing under a deadline constraint for the execution by using a predictive approach that forecasts the link quality of the communication network and the remaining runtime of the execution.

During the remote execution of an application part that is offloadable, the offload controller on the offloading service determines at the breakpoints introduced for safe-point’ing (cf. Subsection 5.4.2) whether it creates and transmits a safe-point to the offloading client. To improve the decision making for safe-point’ing on the offload controller, it solves a scheduling problem of optimal safe-point’ing under a constraint of the maximum time for the execution using an Integer Linear Program (ILP). The solution of the scheduling problem requires a prediction of the link connectivity and a prediction of the remaining runtime for the execution. The offload controller on the offloading service should intuitively create and transmit a safe-point to the offloading client at the current breakpoint if the probability of a connection loss is high and the remaining runtime for an execution lasts too long until reaching the next breakpoint. Otherwise, it takes the risk to wait until it reaches the next breakpoint and again decides about the creation and transmission of a safe-point. Thus, the offload controller waits for the next breakpoint without the creation and/or transmission of a safe-point if either the
6.2. System Overview

probability of a connection loss is low or the probability of reaching the maximum time for the execution is high. As the creation and transmission of a safe-point is ideally equivalent to switch the execution side from the offloading service to the offloading client, the offload controller only creates and transmits a safe-point if – according to the prediction models – it does not reach the next breakpoint. Subsequently, if the offloading client receives a safe-point from the offloading service, it continues the local execution of an application part based on the safe-point received. The offloading client can theoretically re-distribute the application part – that it continued locally – again to the offloading service after the disconnection. Regarding the typical duration of a disconnection that lasts longer than the typical duration of the remaining execution for an application part, however, a re-distribution is very unlikely.

At the same time, during a remote execution of an application part, the failure handler on the offloading client monitors via the service connector the link to the offloading service as well as the remote service itself. To monitor the link and the service, the failure handler utilizes message probing as well as the receipt of safe-points. Based on the prediction of the remaining runtime for the execution, the failure handler sets a deadline until it has to receive a response or a safe-point from the offloading service to meet the constraint of the maximum time for the execution. Please note that the failure handler sets adaptively the deadline such that a local continuation of a remote execution based on a safe-point received most recently will end before the maximum time for the execution. Thus, if the failure handler does not receive a further safe-point until the deadline set, it invokes the offload controller to continue the remote execution locally based on the safe-point received most recently.

Summarizing, we make the following contributions: (1) A formulation of the problem to optimally schedule safe-point’ing under temporal disconnections of the communication network together with a given deadline constraint for the execution time; (2) an adaptive algorithm that solves the formulated problem with the help of an ILP and a prediction model for the link quality of the communication network and the remaining time of the execution; (3) an implementation of the predictive approach for the preemptable distribution; and (4) an evaluation of the efficiency of the predictive distribution based on simulations, calibrated through energy measurements on mobile devices and through connectivity traces of mobile devices.

6.2. System Overview

The predictive distribution adds to the preemptable distribution an adaptive algorithm based on prediction models for safe-point’ing. Thus, the system model as well as the
system components are same as before (cf. Section 5.2).

The scheduling problem of the predictive distribution improves the preemptable distribution with an adaptive approach for safe-point’ing based on prediction models. Regarding the problem statement of the preemptable distribution (cf. Section 5.2), the predictive distribution adds the additional constraint $T^{\text{max}}(A_{\alpha_o}, \Xi(\xi))$ for the execution time. $T^{\text{max}}(A_{\alpha_o}, \Xi(\xi))$ is the maximum time tolerated for an execution of the application part $A_{\alpha_o}$ on $\Xi(\xi)$. Due to the constraint added to the problem statement, it meets a deadline predefined for an execution of an application part, ensuring a minimal responsiveness of an application. The offload controller searches for a strategy of the creation and transmission of safe-point(s) that selects the optimal times and numbers for safe-point’ing such that the total consumed energy is minimal under the given time constraint. Formally, the objective is to minimize the cost function $f_w$ under the given constraint of the maximum time for the execution:

$$\min_{\xi} \quad f_w(A_{\alpha_o}, \Xi(\xi))$$

$$s.t. \quad T(A_{\alpha_o}, \Xi(\xi)) \leq T^{\text{max}}(A_{\alpha_o}, \Xi(\xi))$$

(6.1)

As an end user sets the constraint $T^{\text{max}}(A_{\alpha_o}, \Xi(\xi))$ of the maximum time for the execution, we set the user-defined weight $w_t$ for the execution time to 0. Moreover, we also set the monetary cost $C(A_{\alpha_o}, \Xi(\xi))$ to 0, because the question of monetary cost charged in case of a failure is left as a remaining open research question\(^1\). This simplifies the equation as follows:

$$\min_{\xi} \quad w_e \cdot E(A_{\alpha_o}, \Xi(\xi))$$

$$s.t. \quad T(A_{\alpha_o}, \Xi(\xi)) \leq T^{\text{max}}(A_{\alpha_o}, \Xi(\xi))$$

(6.2)

Summarizing, the goal is to find an optimal schedule for the creation and transmission of safe-points that solves the given constrained optimization problem under dynamic network conditions like link failures or varying bandwidth. Please note that both the number of safe-points and the points in time of the creation and transmission of safe-points influence the constrained optimization problem.

\(^1\)For instance, based on a received safe-point, an offloading client has continued the execution locally due to the occurrence of a failure and receives later on the last safe-point of the remote execution from the offloading service due to a re-connection. In case of the offloading client did not finish the execution yet, it utilizes the last safe-point and pays for the whole remote execution as well. In case of the offloading client already finished the execution, it only pays for the partial remote execution for receiving the safe-point to continue the execution locally. As a result, an offloading service may not continue a remote execution in case of the occurrence of a failure, because the offloading client will not pay for it if the result of the remote execution receives too late.
6.3. Optimal Schedule for Safe-point’ing

The goal of the optimal schedule for the creation and transmission of safe-points is to minimize the cost function under the given constraint of the maximum time for an execution. To this end, it models an execution of a Java method (application part) as an execution graph $G = (V, E, \varepsilon^v, \varepsilon^{e(u,v)}, \tau^v, \tau^{e(u,v)})$ being a vertex-weighted and edge-weighted graph. The graph-based execution model consists of a set of vertices $V$, where each $v \in V$ represents an execution block. An execution block corresponds to a sequential execution of code between two breakpoints. Please note that the offload compiler presented in Subsection 5.4.2 inserts a breakpoint before each Java bytecode instruction that branches the execution of a Java method. An edge $e(u,v) \in E$ with $u, v \in V$ defines a transition between execution blocks according to the execution flow of a Java method. Figure 6.1 shows an exemplary execution graph of a Java method that the offload compiler divided into nine execution blocks $EB_i$ with $i = 0, 1, 2, \ldots, 8$.

At run-time, a possible execution path is $EB_0 \rightarrow EB_1 \rightarrow EB_2 \rightarrow EB_6 \rightarrow EB_7$.

The execution graph $G$ further defines two weights $\varepsilon^v$ and $\tau^v$ with $v \in V$ for a vertex and two weights $\varepsilon^{e(u,v)}$ and $\tau^{e(u,v)}$ with $e(u,v) \in E$ and $u, v \in V$ for an edge. The function $\varepsilon^v : V \rightarrow \mathbb{R}$ denotes the energy consumed for an execution of the related execution block. The function $\varepsilon^{e(u,v)} : E \rightarrow \mathbb{R}$ denotes the energy consumed between two execution blocks for distributing the Java method from the offloading client to the offloading service and vice versa (i.e., switching the execution side again). The function $\tau^v : V \rightarrow \mathbb{R}$ denotes the execution time of the related execution block according to the prediction model for the execution time. The function $\tau^{e(u,v)} : E \rightarrow \mathbb{R}$ denotes the time for switching the execution from the offloading client to the offloading service and vice versa, depending on the bandwidth and latency of the network link.
6. Deadline-aware Code Offloading with Predictive Safe-point’ing

Based on the vertex-weighted and edge-weighted execution graph $G$, we formulate the problem of the optimal schedule for safe-point’ing as an Integer Linear Program (ILP). Intuitively, the ILP searches for the placement of each execution block from the execution graph $G$ either on the offloading client or on the offloading service, minimizing the cost function and not violating the constraint of the maximum time for the execution of the Java method. To define the placement of an execution block, the ILP uses a binary vector $B \in \{0, 1\}^{|V|}$. $B_v = 0$ means an execution of $v \in V$ on the offloading service and $B_v = 1$ means an execution of $v \in V$ on the offloading client. Thus, the optimal solution of the ILP is the placement vector that minimizes the cost function without a constraint violation. Using this placement vector, the following equation expresses the execution time $T(A_{\alpha_o}, \Xi(\xi))$:

$$T(A_{\alpha_o}, \Xi(\xi)) = \sum_{v \in V} (B_v \cdot \tau^v_{\Xi(\xi_k)} + (1 - B_v) \cdot \tau^v_{\Xi(\xi_m)}) + \sum_{e(u,v) \in E} |B_u - B_v| \cdot \tau_{B_u,B_v}$$

where $\tau^v_{\Xi(\xi_k)}$ is the time for a local execution if the optimal solution places the execution block $v$ on the offloading client $\Xi(\xi_k)$. Otherwise, it adds the time $\tau^v_{\Xi(\xi_m)}$ for a remote execution if the offloading service $\Xi(\xi_m)$ executes the execution block $v$. The second term defines the time for switching the execution side, where the optimal solution adds this switching time if the execution blocks $u, v \in V$ are on different sides. Beside the execution time, the ILP also requires the energy consumption $E(A_{\alpha_o}, \Xi(\xi))$:

$$E(A_{\alpha_o}, \Xi(\xi)) = \sum_{v \in V} (B_v \cdot \varepsilon^v_{\Xi(\xi_k)} + (1 - B_v) \cdot (\tau^v_{\Xi(\xi_m)} \cdot E_{wait})) + \sum_{e(u,v) \in E} |B_u - B_v| \cdot \varepsilon^{e(u,v)}$$

where $\varepsilon^v_{\Xi(\xi_k)}$ is the energy consumed for a local execution if the optimal solution places the execution block $v$ on the offloading client $\Xi(\xi_k)$. Otherwise, it adds the energy consumed of the offloading client waiting in idle mode if the offloading service $\Xi(\xi_m)$ executes the execution block $v$. The second term of the equation defines the energy consumed for switching the execution side from $u$ to $v$ that includes the creation and transmission of a safe-point. Thus, if the block placement of $u, v \in V$ differs – for instance, $u$ on the offloading client and $v$ on the offloading service, or vice versa, the optimal solution has to add the energy cost for switching the execution side.

Regarding Equation 6.3 and Equation 6.4, the execution time and the energy consumption depend on the time taken and the energy consumed for each individual
6.3. Optimal Schedule for Safe-point’ing

<table>
<thead>
<tr>
<th>IES</th>
<th>EB₀</th>
<th>EB₁</th>
<th>EB₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>int i</td>
<td>int i</td>
<td>int i</td>
<td>int i</td>
</tr>
<tr>
<td>float f</td>
<td>short s</td>
<td></td>
<td>float f</td>
</tr>
<tr>
<td>short s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>String str</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.2.: An exemplary modification of variables for each execution block.

Execution block $v \in V$. The execution blocks differ in the number and kind of the executed Java bytecode instructions (computational complexity), where the execution time of the same execution block differs on the offloading client and on the offloading service. Thus, we express the energy consumed and the time taken for an execution block $v \in V$ as follows:

$$
\tau_{\Xi(\xi)}^v = \frac{P_{\text{exe}} \cdot I_{\text{state}}(A_\alpha), P_{\text{code}}(A_\alpha)}{P_{\text{ppwr}}} \\
\xi_{\Xi(\xi)}^v = \tau_{\Xi(\xi)}^v \cdot E_{\text{exec}}^{\Xi(\xi_k)}
$$

where $P_{\text{ppwr}}$ is the processing power of a resource $\Xi(\xi)$, $E_{\text{exec}}^{\Xi(\xi_k)}$ the energy factor for executing code, and $P_{\text{exe}} \cdot I_{\text{state}}(A_\alpha), P_{\text{code}}(A_\alpha)$ the load metric (cf. Section 3.1). Please note that the offloading framework calibrates $P_{\text{ppwr}}$ through online benchmarks, where we calibrate $E_{\text{exec}}^{\Xi(\xi_k)}$ for the offloading framework through offline measurements.

Beside the energy consumed and the time taken for each individual execution block, Equation 6.3 and Equation 6.4 require the time taken and the energy consumed for a switch of the execution side. The size of a safe-point determines the consumed energy and the taken time, where the size depends on the variables modified since the safe-point transmitted last (incremental safe-point’ing). For instance, the offloading client distributes a Java method (cf. Figure 6.1), sending the input execution state (first safe-point) to the offloading service. The offloading service starts the execution of the Java method based on the first safe-point (at node input execution state). Now, the optimal schedule for safe-point’ing determines to switch the execution side from the offloading service to the offloading client between the execution blocks EB₂ and EB₆. To this end, it requires all variables modified since the offloading service started the execution (EB₀ → EB₂). Figure 6.2 shows an exemplary modification of variables, where a safe-point contains the variables i (EB₂), f (EB₂), and s (EB₀).
6. Deadline-aware Code Offloading with Predictive Safe-point’ing

As a result, the last synchronization of the execution state through safe-point’ing \( S_{i-1} \) with \( i \geq 1 \) determines the energy consumed and the time taken for a switch of the execution side. Please note that \( S_0 \) equals to the execution state contained in the first safe-point at the start of the execution on the offloading service (input execution state). The following equation defines the time taken \( \tau_{B_u,B_v}^{e=(u,v)} \) for a switch of the execution side, where the offloading service executes \( u \in V \) and the offloading client \( v \in V \):

\[
\tau_{B_u,B_v}^{e=(u,v)} = \delta(S_i, S_{i-1}) \cdot B_{down}^\Lambda(\xi_k:\xi_m) + L_{down}^\Lambda(\xi_k:\xi_m) \quad (6.6)
\]

The function \( \delta(S_i, S_{i-1}) \) calculates the size of the incremental safe-point’ing. Moreover, \( B_{down}^\Lambda(\xi_k:\xi_m) \) is the bandwidth and \( L_{down}^\Lambda(\xi_k:\xi_m) \) the latency of the link \( \Lambda(\xi_k:\xi_m) \) between the offloading client and the offloading service (cf. Section 3.1). Please note that for a switch of the execution side from the offloading client to the offloading service only the bandwidth \( B_{up}^\Lambda(\xi_k:\xi_m) \) and the latency \( L_{up}^\Lambda(\xi_k:\xi_m) \) changes.

The following equation defines the energy consumed \( \varepsilon_{B_u,B_v}^{e=(u,v)} \) for a switch of the execution side between the execution blocks \( u, v \in V \). The offloading service executes \( u \) and the offloading client \( v \), hence receiving a safe-point on the offloading client:

\[
\varepsilon_{B_u,B_v}^{e=(u,v)} = \tau_{B_u,B_v}^{e=(u,v)} \cdot E_{recv}^{\Xi(\xi_k)} \quad (6.7)
\]

where \( E_{recv}^{\Xi(\xi_k)} \) is the energy consumed for receiving bytes (cf. Section 3.1).

Summarizing, the optimal schedule solves the following ILP for the creation and transmission of safe-points on the offloading service:

\[
\begin{align*}
\min \quad & \sum_{v \in V} \left( B_v \cdot \frac{P_{exe}(I_{state}(A_\alpha), P_{pcode}(A_\alpha))}{P_{pwr}^{\Xi(\xi_k)}} \cdot E_{exec}^{\Xi(\xi_k)} + (1 - B_v) \cdot \frac{P_{exe}(I_{state}(A_\alpha), P_{pcode}(A_\alpha))}{P_{pwr}^{\Xi(\xi_m)}} \cdot E_{wait}^{\Xi(\xi_k)} \right) + \sum_{e(u,v) \in E} \left( |B_u - B_v| \cdot \left( \delta(S_i, S_{i-1}) \cdot B_{down}^\Lambda(\xi_k:\xi_m) + L_{down}^\Lambda(\xi_k:\xi_m) \right) \cdot E_{recv}^{\Xi(\xi_k)} \right) \\
\text{s.t.} \quad & \sum_{v \in V} \left( B_v \cdot \frac{P_{exe}(I_{state}(A_\alpha), P_{pcode}(A_\alpha))}{P_{pwr}^{\Xi(\xi_k)}} + (1 - B_v) \cdot \frac{P_{exe}(I_{state}(A_\alpha), P_{pcode}(A_\alpha))}{P_{pwr}^{\Xi(\xi_m)}} \right) + \sum_{e(u,v) \in E} \left( |B_u - B_v| \cdot \left( \delta(S_i, S_{i-1}) \cdot B_{down}^\Lambda(\xi_k:\xi_m) + L_{down}^\Lambda(\xi_k:\xi_m) \right) \right) \leq T^{max}(A_\alpha, \Xi(\xi))
\end{align*}
\]
6.3. Optimal Schedule for Safe-point’ing

6.3.1. Prediction of Link Connectivity

The solution of the ILP presented above depends on the knowledge of the bandwidth $B^{\Lambda(\xi_k;\xi_m)}$ and the latency $L^{\Lambda(\xi_k;\xi_m)}$ of the link $\Lambda(\xi_k;\xi_m)$ between the offloading client and the offloading service. To this end, the optimal schedule uses a prediction model of the link based on Markov chains. Researchers widely use Markov chains for the prediction of the link quality like in [ZWG+13], because a Markov chain requires a low memory space together with a low computational complexity. A Markov chain is a stochastic process with an initial state of a state space, where a transition describes the change from a state to another state based on a transition probability. A transition matrix outlines the transition probabilities of a Markov chain that satisfies the Markov property. Fulfilling the Markov property, a Markov chain predicts the future of a stochastic process solely based on its current state. Thus, the transition probability for switching from a state into another state only depends on the current state and not on the state history of past transitions. Formally, let the pair $(\Omega,F)$ be a measurable space with the sample space $\Omega$ and the \( \sigma \)-algebra $F$. The set $\Omega$ contains all possible outcomes, where an outcome represents for a single execution the output. The collection $F$ is a subset of $\Omega$ that contains the considered events, where an event is a set of zero or more outcomes. Moreover, let the triplet $(\Omega,F,P)$ be the probability space with the probability measure $P$ that returns an event’s probability and thus, is the function $P:F\rightarrow[0,1]$. Based on the measurable space $(\Omega,F)$, the filtration $\{F_t\}_{t \geq 0}$ with $F_t \subseteq F$ and $t \in T$ fulfills $t_1 \leq t_2 \implies F_{t_1} \subseteq F_{t_2}$, where the index set $T$ equals $\mathbb{R}_{\geq 0} = \{r \in \mathbb{R}| r \geq 0\}$. [Ste94, Str13]

An $(\Omega,F)$-valued stochastic process $X = (X_t, t \in T)$ fulfills the Markov property if

$$P(X_t \in A|F_k) = P(X_t \in A|X_k) \quad (6.9)$$

is true $\forall A \in F$ and $\forall k,l \in T$ with $k < l$. For a Markov chain with a discrete time where $T$ equals $\mathbb{N}^0 = \{0,1,2,\ldots\}$ and a finite space of discrete states where $F$ is a discrete set (called a Discrete-Time Markov Chain (DTMC)) the Markov property is:

$$P(X_n = x_n|X_{n-1} = x_{n-1}, X_{n-2} = x_{n-2}, \ldots, X_0 = x_0) = P(X_n = x_n|X_{n-1} = x_{n-1})$$

To build the prediction model of the link based on a Markov chain, the optimal schedule considers the bandwidth and latency of past connections from the offloading client. As a result, the offloading client does not measure in parallel the bandwidth and latency to an offloading service, keeping the (energy) overhead introduced to the system low. The optimal schedule maps the continuous values for the bandwidth and latency moni-
6. Deadline-aware Code Offloading with Predictive Safe-point’ing

tored from past connections onto \(n\) discrete quality classes of the link. For instance, an interpretation of the mapping to the five latency classes \([0\text{ ms}, 50\text{ ms}], \ (50\text{ ms}, 100\text{ ms}], \ (100\text{ ms}, 200\text{ ms}], \ (200\text{ ms}, 300\text{ ms}], \ (300\text{ ms}, \infty] \) is “very low latency”, “low latency”, “medium latency”, “high latency”, and “disconnected”. Likewise, possible quality classes for the link bandwidth are, for instance, \([0\text{ kbit/s}, 50\text{ kbit/s}], \ (50\text{ kbit/s}, 350\text{ kbit/s}], \ (350\text{ kbit/s}, 700\text{ kbit/s}], \ (700\text{ kbit/s}, 1050\text{ kbit/s}], \ (1050\text{ kbit/s}, \infty] \), interpreted as “disconnected”, “very low bandwidth”, “low bandwidth”, “medium bandwidth”, and “high bandwidth”. Based on the discrete quality classes for the bandwidth and for the latency of the link, the optimal schedule models each direction (downstream and upstream) of the link as a DTMC with a finite state space \(Q^{B,L}\) of \(n^2\) states. For instance, the finite state space for the above-described quality classes consists of 25 states. The variable \(p_{ij}\) with \(i, j = [0, 1, \ldots, m]\) and \(m = n - 1\) represents the transition probabilities between a state \(Q^{B,L}_i\) and a state \(Q^{B,L}_j\), resulting in the following transition matrix \(P\):

\[
P = \begin{pmatrix}
    p_{00} & p_{01} & \cdots & p_{0m} \\
    p_{10} & p_{11} & \cdots & p_{1m} \\
     \vdots & \vdots & \ddots & \vdots \\
    p_{m0} & p_{m1} & \cdots & p_{mm}
\end{pmatrix}
\]  

(6.10)

where the time duration spent in each state is a non-negative natural number (e.g., with a time step size of 500 ms). Moreover, we assume that the probability of a link failure and a corresponding link recovery is exponentially distributed, fulfilling the Markov property (cf. Nicholson et al. [NN08]). Based on the transition rates, we calculate iteratively the mean recurrence time \(M_{ii}\) and the mean first passage time \(M_{ij}\). The mean first passage time \(M_{ij}\) represents the number of steps expected that the stochastic process takes for reaching node \(j\) from node \(i\) for the first time:

\[
M^{(k+1)} = E + P \ast (M^{(k)} - \text{diag}(M^{(k)}))
\]  

(6.11)

with

\[
M^{(k)} = \begin{pmatrix}
    m^{(k)}_{00} & m^{(k)}_{01} & \cdots & m^{(k)}_{0s} \\
    m^{(k)}_{10} & m^{(k)}_{11} & \cdots & m^{(k)}_{1s} \\
    \vdots & \vdots & \ddots & \vdots \\
    m^{(k)}_{s0} & m^{(k)}_{s1} & \cdots & m^{(k)}_{ss}
\end{pmatrix}
\]

\[
\text{diag}(M^{(k)}) = \begin{pmatrix}
    m^{(k)}_{00} & 0 & \cdots & 0 \\
    0 & m^{(k)}_{11} & \cdots & 0 \\
    \vdots & \vdots & \ddots & \vdots \\
    0 & 0 & \cdots & m^{(k)}_{ss}
\end{pmatrix}
\]

where \(M^{(0)} = E\) and \(E\) denote the identity matrix. [Ste94]
6.3. Optimal Schedule for Safe-point’ing

Summarizing, the offload controller on the offloading service predicts the time-dependent connectivity factors of the bandwidth \( B^A(\xi_k;\xi_m) \) and the latency \( L^A(\xi_k;\xi_m) \) based on the prediction models from Markov chains. By summing up the values for the bandwidth and for the latency, it results in the time-invariant factors for the quality class \( b \) with \( b = 0, 1, \ldots, m \) of the bandwidth and \( l \) with \( l = 0, 1, \ldots, m \) of the latency:

\[
B^A(\xi_k;\xi_m) = \sum_{c=0}^{m} m_{bc}^\infty \\
L^A(\xi_k;\xi_m) = \sum_{c=0}^{m} m_{cl}^\infty
\]

6.3.2. Prediction of Remaining Runtime

Beside the prediction of the link connectivity for the solution of the ILP presented above, the solution of the ILP also depends on the prediction of the remaining runtime for an execution of a Java method on the offloading client as well as on the offloading service. To this end, the optimal schedule uses a prediction model of the execution time to satisfy the constraint of the maximum time for the execution. The prediction model predicts the path for an execution – more precisely, the sequence of execution blocks – of a Java method based on the input execution state. To build the prediction model of execution paths, the optimal schedule monitors past executions of the Java method, retrieving the sequence of blocks executed for the input execution state (history-based approach). Like the app profiler for the basic distribution (cf. Subsection 4.5.2), it utilizes a k-Nearest Neighbors algorithm to profile the sequence of blocks executed for a Java method based on the input execution state. For instance, Figure 6.3 highlights an exemplary execution graph executed at run-time for a Java method (cf. Figure 6.1)
that executes the loop $EB_3 \rightarrow EB_4 \rightarrow EB_5$ four times. Based on the prediction model of execution paths, the optimal schedule predicts the sequence of execution blocks for a Java method. During the execution of the Java method, the offload controller on the offloading service keeps track of execution blocks already executed, knowing at each breakpoint the number and kind of individual execution blocks that the offloading service or the offloading client still has to execute. As a result, it calculates the remaining time for an execution of remaining execution blocks on the offloading service or on the offloading client based on Equation 6.5.

6.4. Evaluation

The performance of the predictive distribution relies on multiple parameters like network quality, energy characteristics of mobile devices, or application complexity. To this end, we implemented a simulation in MATLAB\(^2\) to analyze the performance for a wide range of parameter combinations. To provide a realistic evaluation based on the MATLAB simulation, however, real-world measurements of the connectivity quality from communication networks, energy consumed on mobile devices, and execution times of applications calibrate the parameters used within the MATLAB simulation. Thus, the evaluation gains results under realistic circumstances, where the MATLAB simulation runs different scenarios. First, Subsection 6.4.1 describes the evaluation setup before Subsection 6.4.2 presents the evaluation results.

6.4.1. Setup

The evaluation of the optimal schedule for the predictive distribution with safe-points uses the same setup presented in Subsection 5.5.1 with the following differences.

Beside the Chesspresso application already considered for the preemptable distribution (cf. Section B.2), the evaluation further considers a face recognition application presented in detail in Section B.4. The face recognition application recognizes human faces in an image. In detail, it is a good candidate for a distribution, because it has a high complexity of computation, a huge size of the input execution state, a small size of safe-points, and a small size of the output execution state.

In order to solve the given ILP in Equation 6.8, the offloading service uses the standard solver from MATLAB that is highly optimized and quickly solves the problem in a few milliseconds keeping the overhead introduced to the offloading service small. In detail, the time overhead of the standard solver from MATLAB executed on the

\[ \text{https://www.mathworks.com/products/matlab.html} \]
6.4. Evaluation

(a) Power consumption on the laptop for a local execution of the Chesspresso application.

(b) Power consumption on the laptop for a remote execution of the Chesspresso application.

Figure 6.4.: Power consumption on the laptop for (a) a local execution and (b) a remote execution of the Chesspresso application, highlighting the timing parameters for the predictive distribution.
offloading service for solving the ILP is less than 50 ms for the Chesspresso application or for the face recognition application. The MATLAB simulation takes several timing parameters into account, basically depending on the processor speed of the device, the computational complexity of the application, and the data rate of the communication link. To this end, we calibrated the timing parameters through the execution of both applications on the Jikes RVM prototype for the preemptable distribution on both devices. Beside the timing parameters, we also measured the energy consumed for the execution of the application code on the offloading client and on the offloading service (cf. Section 4.7). The measurements of the timing parameters and the energy consumption on the physical devices provide a realistic basis for the simulation. Table 6.1 summarizes and Figure 6.4 outlines the timing parameters. In detail, \( T_{lsc} \) is the time until the execution reaches the Java method that is offloadable. \( T_{lec} \) and \( T_{rmc} \) are the time for an execution of the Java method on the offloading client and on the offloading service. \( T_{lg} \) and \( T_{lst} \) are the time for the creation of the input execution state (first safe-point) on the offloading client and the transmission of the first safe-point from the offloading client to the offloading service. Similarly, \( T_{rsg} \) and \( T_{rst} \) are the time for the creation of a safe-point on the offloading service and the transmission of a safe-point from the offloading service to the offloading client. \( T_{lsi} \) and \( T_{rsl} \) are the time for the installation of a safe-point on the offloading client and on the offloading service. Finally, \( T_h \) defines the constraint of the maximum time for the application execution, which we set lower for the Chesspresso application than for the face recognition application. We set \( T_h \) according to the execution time of the applications on the offloading client.
6.4. Evaluation

Table 6.1.: Timing parameters calibrated on real-world measurements for the evaluation of the predictive distribution with safe-points.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Chesspresso Application</th>
<th>Face Recognition Application</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Netbook</td>
<td>Laptop</td>
</tr>
<tr>
<td>$T_h$</td>
<td>Maximum time for the application execution</td>
<td>40.000 s</td>
<td>10.000 s</td>
</tr>
<tr>
<td>$t_s$</td>
<td>Granularity of the time slot</td>
<td></td>
<td>0.001 s</td>
</tr>
<tr>
<td>$T_{isc}$</td>
<td>Execution time for reaching the offloadable Java method</td>
<td>5.797 s</td>
<td>1.421 s</td>
</tr>
<tr>
<td>$T_{lmc}$</td>
<td>Execution time for a local execution of the Java method</td>
<td>27.705 s</td>
<td>7.307 s</td>
</tr>
<tr>
<td>$T_{rmc}$</td>
<td>Execution time for a remote execution of the Java method</td>
<td>4.249 s</td>
<td></td>
</tr>
<tr>
<td>$T_{lsc}$</td>
<td>Execution time for reaching the end of the application</td>
<td></td>
<td>0.778 s</td>
</tr>
<tr>
<td>$T_{lsg}$</td>
<td>Time of generating the input execution state</td>
<td>1.146 s</td>
<td>0.655 s</td>
</tr>
<tr>
<td>$T_{lst}$</td>
<td>Time of transmitting the input execution state to remote side</td>
<td>0.100 s</td>
<td></td>
</tr>
<tr>
<td>$T_{lsi}$</td>
<td>Time of installing a safe-point locally</td>
<td>0.157 s</td>
<td>0.014 s</td>
</tr>
<tr>
<td>$T_{rsg}$</td>
<td>Time of generating a safe-point on the remote side</td>
<td></td>
<td>0.029 s</td>
</tr>
<tr>
<td>$T_{rst}$</td>
<td>Time of transmitting a safe-point to the client</td>
<td>0.010 s</td>
<td></td>
</tr>
<tr>
<td>$T_{rsi}$</td>
<td>Time of installing the input execution state remotely</td>
<td>0.327 s</td>
<td></td>
</tr>
</tbody>
</table>
6. Deadline-aware Code Offloading with Predictive Safe-point’ing

Beside the execution time and energy consumption, the simulation requires connectivity parameters of the network link. To this end, we measured the network connectivity in the cellular networks of the providers O2 and T-Mobile. We measured the connectivity traces at different days while riding a train of the public transport from Stuttgart, Germany (cf. Figure 6.5). To retrieve the connectivity traces, a mobile device pings periodically via a cellular connection a server at the University of Stuttgart connected to the Internet through the fixed university network. Based on the connectivity traces measured for the cellular network of O2 and T-Mobile, the offloading service builds the prediction models with the help of Markov chains. Equation 6.12 outlines the prediction model for the cellular network of T-Mobile and Equation 6.13 of O2, differing significantly in the occurrence and duration of disconnections:

\[ P = \begin{pmatrix} 0.8861 & 0.1139 \\ 0.1950 & 0.8051 \end{pmatrix} \quad M_{ij}^{(\infty)} = \begin{pmatrix} 1.5842 & 8.7826 \\ 5.1304 & 2.7119 \end{pmatrix} \quad (6.12) \]

\[ P = \begin{pmatrix} 0.9859 & 0.0141 \\ 0.1081 & 0.8919 \end{pmatrix} \quad M_{ij}^{(\infty)} = \begin{pmatrix} 1.1307 & 70.7500 \\ 9.2500 & 8.6486 \end{pmatrix} \quad (6.13) \]

Informally, the quality of the cellular network is higher for T-Mobile in comparison to O2 (cf. \( M_{01}^{(\infty)} \) of Equation 6.12 and 6.13). Due to the significance of the prediction models for the connectivity, we evaluated the accuracy of the applied Markovian model, dividing the real-world traces of the connectivity into a training set and a test set. Based on the training set and the test set, we built the Markov model on the training set and evaluated it against the test set. In average, the (simple) stochastic model based on Markov chains predict the quality of the cellular network in 65% properly.

To evaluate the performance and efficiency of the predictive distribution called Predictive, the evaluation compares it with a local execution, the basic distribution with a re-execution in case of a failure called Baseline, the preemptable distribution with periodic safe-point’ing called Periodic, and an optimal distribution called Optimal. A local execution executes the application locally on the mobile device without code offloading. Baseline uses a basic strategy at the occurrence of a failure by re-executing the offloaded Java method as soon as a disconnection occurs like in [KT12]. Periodic uses a static schedule (i.e., non-predictive and non-optimized) for the creation and transmission of safe-points, where the offloading client receives from the offloading service a safe-point at an application-specific time interval predefined to 1 s for the evaluation. As a reference, Optimal uses a perfect schedule for safe-point’ing knowing in advance the point in times for a disconnection as well as its duration. Please note that an optimal distribution with a perfect schedule is only achieved theoretically.
6.4. Evaluation

Figure 6.6.: Frequency of execution times for the Chesspresso application on the netbook based on the connectivity traces for T-Mobile.

6.4.2. Results

This subsection describes the results for the evaluation of the predictive distribution, categorized into the evaluated applications.

Chesspresso Application

Now, we present the evaluation results of the execution time and the energy consumption on the netbook and on the laptop, distributing the Java method of the Chesspresso application. Please note that, for a better overview, each following figure only shows the frequencies of the execution time or the energy consumption that are greater than 0. Due to the heterogeneity of the hardware platform from the netbook and the laptop, the execution time for a local execution of the Chesspresso application differs significantly on both hardware platforms, taking 29.542s on the netbook and 8.841s on the laptop. We define the maximum time for the execution of the Chesspresso application slightly higher than a local execution, setting it to 40s on the netbook and to 10s on the laptop (cf. Table 6.1).

For the netbook, Figure 6.6 shows the frequency of execution times for the different approaches, distributing the Java method to the offloading service based on 1278 connectivity traces of the cellular network of T-Mobile. Regarding the frequencies in the figure for the different approaches, the different approaches result in execution times, centered around two peeks at 12.60s and 34.28s. The execution time 12.60s (first peek) corresponds to the duration of a distribution, where no disconnections occur during the remote execution and the offloading service executes completely the Java method.
6. Deadline-aware Code Offloading with Predictive Safe-point’ing

Figure 6.7.: Frequency of energy consumption for the Chesspresso application on the netbook based on the connectivity traces for T-Mobile.

In detail, the frequency of the execution time for the first peek counts for Baseline 676, for Periodic 641, for Predictive 666, and for Optimal 717. The execution time 34.28 s (second peek) corresponds to the duration of a local execution for the Java method on the netbook, where Baseline, Periodic, and Predictive counts 532 and Optimal counts 270. The reduction of the execution time about 21.68 s shows the benefits for the distribution of the Java method, utilizing the faster hardware platform of the desktop on behalf of the slower hardware platform of the netbook. Having a closer look at the execution times between the two peeks, Baseline finishes an execution of the Java method 63 times, Periodic 40 times, Predictive 78 times, and Optimal 291 times between the two peeks. Thus, the execution times for Optimal stay totally between the two peeks, where the execution time for the worst case corresponds to a local execution of the Java method on the netbook (34.28 s). Predictive results in 2 execution times that are higher than the execution time for a local execution, Periodic in 65, and Baseline in 7.

The lowest number of execution times between the two peeks together with the highest number of execution times after the second peek causes the highest execution time for Periodic of 23.27 s on average. Baseline results on average in a slightly smaller execution time of 22.26 s, which Predictive further reduces to 22.09 s. However, Optimal has on average an execution time of 19.20 s and thus, a 13.08% lower execution time than Predictive. Regarding the latest time the execution of the Java method finishes, Baseline violates 5-times the maximum time set for an execution, whereas Periodic stays below the maximum time set and finishes an execution not later than 38.00 s. This shows the positive effect of safe-point’ing in general. Predictive also stays below the maximum time set with a maximum time for an execution of 36.00 s.
Regarding the energy consumption caused by the different approaches for a distribution of the Java method on the netbook (cf. Figure 6.7), a total execution of the Java method on the offloading service consumes 128.53 J (first peek) and on the offloading client 369.12 J (second peek). So, the first peek corresponds to a remote execution, where no disconnections occur. The second peak corresponds to a local execution on the netbook due to the occurrence of failures. Thus, the netbook saves energy of 240.59 J by distributing the Java method to the desktop. In detail, the average energy consumed for Baseline and Predictive are similar, consuming in average 234.43 J and 233.95 J, respectively. Due to the improved schedule for safe-point’ing, the average energy consumed for Predictive is lower than Periodic that causes a higher energy consumption of 13.00 J. The lowest energy consumed in average causes Optimal with 198.76 J, resulting in a 15.04% lower energy consumption compared to Predictive.

Figure 6.8 shows the frequency of execution time for 1185 connectivity traces of the cellular network of O₂. Like for the cellular network of T-Mobile, the frequency of execution times has two peeks at 12.60 s and at 34.28 s, where the execution time of a local execution dominates over the execution time for a remote execution. In detail, Baseline counts 323 at the first peek and 783 at the second peek, Periodic 319 and 783, Predictive 322 and 783, and Optimal 353 and 489. The main reason for the decrease of the first peek together with the increase of the second peek for all approaches is the higher likelihood of a disconnection for O₂. As a result, the failure handler of the approaches starts more often a local execution in case of the occurrence of a failure. As a result, the average time for an execution increases for Baseline to 27.87 s (an increase of 5.61 s), Periodic to 28.32 s (an increase of 5.05 s), Predictive to 27.55 s (an
6. Deadline-aware Code Offloading with Predictive Safe-point’ing

![Graph showing energy consumption for different approaches]

Figure 6.9.: Frequency of energy consumption for the Chesspresso application on the netbook based on the connectivity traces for O2.

increase of 5.46 s), and Optimal to 24.31 s (an increase of 5.11 s). Beside the increase of the average time for an execution, the connectivity traces of the cellular network of O2 leads to the same characteristics as of T-Mobile. Periodic has the highest average time with 30 execution times between the two peeks and 53 execution times after the second peek. In detail, it finishes an execution not later than 38.00 s and thus, does not violate the maximum time for an execution, showing the benefits of safe-point’ing. Due to 61 execution times between the two peeks and 18 execution times after the second peek, Baseline reduces the average time compared to Periodic, however, violating the maximum time for an execution 16 times. Predictive further decreases the maximum time for an execution. Due to 76 execution times between the two peeks and 4 execution times after the second peek, it finishes an execution not later than 35.50 s and thus, staying below the maximum time set. Only Optimal keeps the time for an execution below 34.28 s, counting 343 execution times between the two peeks. As a result, the average time for an execution of Optimal is 11.76% lower compared to Predictive.

Regarding the energy consumption caused by the different approaches for a distribution of the Java method with the cellular network of O2 (cf. Figure 6.9), the consumed energy is proportionally similar to the cellular network of T-Mobile. It is only increased in general by the higher execution times due to the higher likelihood for the occurrence of a failure. In detail, the average energy consumed for Baseline is 273.14 J, for Periodic 279.27 J, for Predictive 271.53 J, and for Optimal 234.79 J, where the average energy consumed for Predictive is lower than Periodic as well as Baseline. However, Optimal consumes in average the lowest energy, resulting in a 13.53% lower energy consumption compared to Predictive.
6.4. Evaluation

Now, we consider the execution times (cf. Figure 6.10 and Figure 6.12) and energy consumption (cf. Figure 6.11 and Figure 6.13) for the different approaches on the laptop. The laptop also distributes the Chesspresso application to the offloading service based on the connectivity traces of the cellular network for T-Mobile and O₂. Compared to the hardware platform of the netbook, the hardware platform of the laptop is more powerful and thus, closer to the performance of the desktop. As a result, the time (and energy) benefit for a distribution of the Java method is smaller, where also a short duration of a disconnection matters significantly.

For the laptop, Figure 6.10 shows the frequency of execution times for the different approaches based on 1428 connectivity traces of the cellular network for T-Mobile. Here, the execution times settle around the two peaks of 6.92s and 8.84s. The first peak at 6.92s corresponds to a remote execution of the Java method, where the offloading service executes it totally without the occurrence of a failure. For the first peak, Baseline counts to 761, Periodic and Predictive to 746, and Optimal to 784. The second peak at 8.84s corresponds to the execution time for a local execution on the offloading client due to the occurrence of failures. For the second peak, Baseline and Periodic counts to 585, Predictive to 599, and Optimal to 567. The reduction of the execution time by 1.92s is the benefit for a distribution of the Java method. Due to the smaller benefit of a distribution for the laptop, the number of executions times between the two peaks is for Baseline and Periodic low (22 and 0, respectively), reacting too late on the occurrence of a failure. Utilizing the predictive scheduler for safe-point‘ing, Predictive results in 67 execution times between the two peaks, only 10 execution times lower than Optimal. Furthermore, Optimal finishes an execution of the Java method not later than the execution time for a local execution (second peak: 8.84s), where
6. Deadline-aware Code Offloading with Predictive Safe-point’ing

Predictive results in 16 execution times that are higher than 8.84 s but lower than the maximum time set to 10 s for the laptop. Both Baseline and Periodic violate several times the maximum time set (56 and 74, respectively), highlighting the positive effect of the utilization of the prediction models. On average, Predictive leads to lower times for an execution (7.78 s) compared to Baseline (7.85 s) as well as Periodic (7.93 s). Compared to Optimal, Predictive leads only to a 0.77% higher average time for an execution, i.e., being very close to optimal.

Figure 6.11 shows the energy consumption for the connectivity traces of the cellular network for T-Mobile on the laptop, corresponding to 250.26 J (first peek) for a remote execution and 406.78 J (second peek) for a local execution. Thus, a distribution of the Java method reduces the energy consumption on the laptop by 156.52 J. Regarding the average energy consumed of the different approaches for T-Mobile, Predictive (317.56 J) consumes in average less energy than Periodic (324.28 J), whereas it consumes in average more energy than Baseline (315.88 J). As the energy consumed by Baseline is not higher than 406.78 J (energy for a local execution), the 16 energy consumption for Predictive that are higher than 406.78 J are responsible for the higher average energy consumed by Predictive (cf. Figure 6.11). Having a closer look at the 16 connectivity traces for T-Mobile that cause the higher energy consumption for Predictive, they correspond to the worst-case scenario of the preemptable distribution. Compared to Optimal (311.69 J), Predictive consumes in average only 1.85% more energy.

The higher likelihood of a disconnection for the 1335 connectivity traces of O₂ increases in general the execution times of the different approaches like for the netbook (cf. Figure 6.12). It shifts the execution times towards the second peek (8.84 s) of a local execution, because the failure handler of the different approaches starts a local execution or continues a remote execution locally in case of the occurrence of a fail-
6.4. Evaluation

The increase of the average time for an execution is roughly 6%, resulting in an average time of 8.36 s for Baseline, 8.38 s for Periodic, 8.27 s for Predictive, and 8.21 s for Optimal. In detail, Baseline counts to 378 for the first peek and 878 for the second peek, Periodic to 375 and 878, Predictive to 375 and 898, and Optimal to 396 and 876. Regarding the number of execution times between the two peeks, Baseline and Periodic reacts too late at the occurrence of a failure (2 and 0, respectively), whereas the utilization of the prediction models improves the reaction to failures and increases the number to 41 for Predictive, being only 22 less than Optimal. Moreover, Baseline and Periodic violate in total 72 and 62 times the maximum time set for an execution, whereas Predictive (and Optimal) never violates the maximum time set (10 s).

The energy consumption of the different approaches on the connectivity for O2 has the same characteristics as for T-Mobile, only increased by roughly 11% owing to the higher likelihood for the occurrence of a failure (cf. Figure 6.13). In detail, the average energy consumed for Baseline is 355.89 J, for Periodic 361.93 J, for Predictive 357.05 J,
6. Deadline-aware Code Offloading with Predictive Safe-point’ing

Figure 6.14.: Frequency of execution time for the face recognition application on the netbook based on the connectivity traces for T-Mobile.

and for Optimal 352.44 J, where Predictive consumes 1.16 J more energy than Baseline due to the worst-case scenarios like for T-Mobile. Compared to Optimal, Predictive consumes only 1.29% more energy.

Summarizing, the basic strategy of failure handling from Baseline violates a maximum time set for a distribution of the Java method from the Chesspresso application. The utilization of the preemptable distribution with safe-points improves the results, however, still violating the maximum time set. Only the improvement with a predictive schedule for safe-point’ing does not violate the maximum time set despite the occurrence of failures due to the utilization of the prediction models. In comparison with an optimal approach that knows in advance the point in time as well as the duration of a failure, the predictive schedule only performs slightly worse.

**Face Recognition Application**

Now, we present the evaluation results of the execution time and the energy consumption on the netbook and on the laptop, distributing the Java method of the face recognition application. Like for the evaluation results of the chess game, each following figure only shows the frequencies that are greater than 0. Moreover, due to the heterogeneity of the hardware platform from the netbook and the laptop, we also set the maximum time to 50 s on the netbook and to 30 s on the laptop related to the execution time for a local execution of 42.89 s and 24.64 s, respectively (cf. Table 6.1).

For the netbook, Figure 6.14 shows the frequency of execution times and Figure 6.15 of the energy consumption for the different approaches. The different approaches distribute the Java method of the face recognition application to the offloading service based on 1228 connectivity traces of the cellular network of T-Mobile. Like for the distribution of the chess game, the two peaks of a remote execution and of a local exe-
6.4. Evaluation

Figure 6.15.: Frequency of energy consumption for the face recognition application on the netbook based on the connectivity traces for T-Mobile.

eution dominate the frequencies of the execution time (22.59 s and 42.89 s) and of the energy consumption (233.19 J and 463.60 J) for the different approaches. The difference between the two peaks highlights the benefits of a distribution without the occurrence of a failure, reducing the execution time by 20.30 s and the energy consumption by 230.41 J. For the execution time (and for the energy consumption), Baseline counts for the first peek and for the second peek to 561 and 581 (561 and 581), Periodic to 503 and 593 (503 and 581), Predictive to 572 and 581 (508 and 581), and Optimal to 632 and 299 (615 and 298). The execution time (energy consumption) between the two peaks sums up for Baseline to 75 (77), for Periodic to 51 (63), for Predictive to 70 (134), and for Optimal to 297 (315). Moreover, Baseline causes 11-times (9-times) a higher execution time (energy consumption) than a local execution, Periodic 81-times (81-times), and Predictive 5-times (5-times), whereas Optimal takes not longer than and consumes not more than a local execution. Baseline as well as Periodic violate in total 9-times and 31-times the maximum time set, whereas Predictive as well as Optimal do not violate it. Due to the similarity of the frequencies for Baseline and Predictive, the average taken time as well as the average consumed energy are similar for both approaches, where the results are slightly lower for Predictive (time: 0.79%; energy: 1.78%). In detail, Baseline results in 32.96 s and 349.28 J and Predictive in 32.70 s and 348.66 J. Compared to the average taken time (29.46 s) and the average consumed energy (307.63 J) for Optimal, Predictive leads to a higher average taken time of 9.91% and a higher average consumed energy of 11.77%.

Figure 6.16 shows the frequency of the execution time and Figure 6.17 of the energy consumption on the netbook from 1335 connectivity traces for the cellular network of O₂. The evaluation results are comparatively similar to the connectivity traces for the cellular network of T-Mobile, only increased by the higher likelihood of the occurrence...
6. Deadline-aware Code Offloading with Predictive Safe-point’ing

Figure 6.16.: Frequency of execution time for the face recognition application on the netbook based on the connectivity traces for O₂.

of a disconnection. In detail, the higher likelihood roughly halves the frequency at the first peek and mainly adds it to the second peek, resulting for the execution time (the energy consumption) in 237 and 813 (237 and 810) for Baseline, 217 and 819 (217 and 810) for Periodic, 246 and 810 (218 and 810) for Predictive, and 289 and 544 (277 and 540) for Optimal. The frequency between the two peeks as well as after the second peek stays similar for the execution time (and the energy consumption), where Baseline sums up to 72 and 13 (76 and 12), Periodic to 38 and 61 (47 and 61), Predictive to 67 and 12 (95 and 12), and Optimal to 302 and 0 (318 and 0). Due to the simple strategy of failure handling in case of a disconnection, Baseline violates 12 times the maximum time set, where Periodic violates it 23 times despite the utilization of safe-point’ing. Only Predictive (and thus, Optimal) does not violate the maximum time set due to the utilization of a predictive schedule for safe-point’ing. On average, the execution time and energy consumption for Baseline (38.09 s and 406.99 J) and Predictive (37.79 s and 406.11 J) are similar, whereas the execution time and energy consumption for Periodic is roughly 1.00 s and 12.50 J higher. Compared to Optimal, the average time and the average energy for an execution of Predictive is 7.94% and 9.52% higher.

For the laptop, Figure 6.18 shows the frequency of the execution time and Figure 6.19 of the energy consumption on the laptop from 1328 connectivity traces for the cellular network of T-Mobile. The time taken as well as the energy consumed for a remote execution (16.04 s and 654.70 J) and for a local execution (24.64 s and 1148.20 J) dominate the frequencies for all approaches like 637 and 585 of the time as well as 647 and 588 of the energy for Baseline, 575 and 585 as well as 575 and 603 for Periodic, 637 and 590 as well as 575 and 586 for Predictive, and 675 and 428 as well as 706 and 419 for Optimal. Thus, the benefit for a distribution of the face recognition application is on
6.4. Evaluation

Figure 6.17.: Frequency of energy consumption for the face recognition application on the netbook based on the connectivity traces for O₂.

Figure 6.18.: Frequency of execution time for the face recognition application on the laptop based on the connectivity traces for T-Mobile.

the laptop 8.60 s and 493.50 J. For the taken time and the consumed energy between the two peeks, Baseline results in 55 and 78, Periodic in 32 and 50, Predictive in 77 and 142, and Optimal in 225 and 203. The taken time and the consumed energy are for Baseline 51 times and 15 times higher than for a local execution, 136 times and 100 times for Periodic, 24 times and 25 times for Predictive, and 0 times and 0 times for Optimal. As a consequence of the frequencies, the average time for an execution is similar for Baseline (20.51 s) and Predictive (20.27 s), whereas the average energy consumed for Baseline (896.51 J) is slightly lower than for Predictive (901.69 J). However, Baseline violates 26 times the maximum time set, where Predictive stays below it for all the connectivity traces. The average time taken and the average energy consumed for an execution is for Periodic higher (21.17 s and 941.89 J), violating 39 times the maximum time set. Compared to Optimal (19.46 s and 834.47 J), Predictive has in average a 4.0% higher execution time and a 7.46% higher energy consumption.

Figure 6.20 shows the frequency of the execution time and Figure 6.21 of the energy consumption.
6. Deadline-aware Code Offloading with Predictive Safe-point’ing

Figure 6.19: Frequency of energy consumption for the face recognition application on the laptop based on the connectivity traces for T-Mobile.

energy consumption on the laptop from 1235 connectivity traces for the cellular network of O₂. Due to the higher likelihood of a disconnection for the cellular network of O₂, it increases the frequencies for both the execution time and the energy consumption towards a local execution (second peek) compared to the cellular network of T-Mobile. The frequencies for the first peek and the second peek are 283 and 841 (288 and 844) for Baseline, 266 and 841 (266 and 853) for Periodic, 283 and 848 (266 and 841) for Predictive, and 307 and 725 (324 and 715) for Optimal. The frequencies between the two peeks as well as after the second peek are similar for O₂ and T-Mobile. Baseline results in 42 and 69 for the execution time as well as 68 and 35 for the energy consumption, Periodic in 22 and 106 as well as 36 and 80, Predictive in 64 and 40 as well as 86 and 42, and Optimal in 203 and 0 as well as 225 and 0. Moreover, Baseline and Periodic violate 46 times and 30 times the maximum time set, whereas Predictive and Optimal stay below it for all the connectivity traces. In average, Baseline takes 22.86 s and consumes 1029.80 J, taking 0.34 s longer and consuming 2.90 J more energy than Predictive. Due to the static strategy of periodic safe-point’ing, Periodic takes the longest average time (23.01 s) and consumes the highest average energy (1048.90 J). Knowing in advance the point in time and the duration of a disconnection, Optimal reduces the average taken time and the average consumed energy by 3.33% and 5.82% compared to Predictive.

Summarizing, like for the Chesspresso application, Baseline and Predictive perform for the face recognition application similar on the netbook as well as on the laptop. Both result in a similar execution time and energy consumption for the connectivity traces of T-Mobile and of O₂. However, the basic strategy of failure handling from Baseline violates the maximum time set for a distribution of the face recognition application, whereas Predictive stays below it for all the connectivity traces. In many cases, Periodic takes the longest time and consumes the highest energy compared to the other
6.5. Summary

The predictive distribution described in Section 6.1 improves the safe-point schedule of the preemptable distribution described in Section 5.1 with an adaptive algorithm based on prediction models. It dynamically adapts the point in times for a creation and transmission of safe-points due to dynamic conditions of the communication net-
work. During a remote execution of an application part, the adaptive algorithm on
the offloading service predicts impending link failures to minimize the number of safe-
points received on an offloading client. In the optimal case, the offloading client only
receives one safe-point from the offloading service just before a disconnection happens
that lasts longer than the remaining time from a time constraint set for the execution.
The underlying system overview described in Section 6.2 with the system model, prob-
lem statement, and system components extends the preemptable distribution with an
additional constraint for the maximum time of an execution. To optimize the efficiency
of the preemptable distribution with safe-point’ing under the constraint, the predictive
distribution utilizes an optimal schedule for safe-point’ing described in Section 6.3. It
selects the optimal time to send a safe-point from the offloading service to the offload-
ing client by using a predictive approach that takes future network connectivity into
account. To evaluate the efficiency of the predictive distribution compared to the pre-
emptable distribution, the evaluation described in Section 6.4 presents the evaluation
setup and the evaluation results of multiple MATLAB simulations calibrated with real-
world measurements. The evaluation of the predictive distribution shows that it sig-
ificantly increases the efficiency – often tending the optimal performance – compared
to other distribution approaches and additionally guarantees a maximum time for an
execution under network failures. In detail, application parts that are computational-
-intensive and have a small execution state like the Chesspresso application pronounce
more the benefits for a predictive distribution. These kind of application parts nat-
urally lend themselves better for code offloading than application parts with a larger
execution state transferred between an offloading client and an offloading service.
Chapter 7
Optimized Code Offloading through Cooperative Caching

The basic distribution presented in detail in Chapter 4 causes computation and communication overhead due to, for instance, executing the offloading framework or sending and receiving execution states. Moreover, an end user has to pay monetary cost for a remote execution of an application part distributed to an offloading service at a commercial cloud. To this end, this chapter presents a caching-aware distribution described in Section 7.1 that extends the offloading framework of the basic distribution with a cache on the remote side. The addition of a cache on the remote side changes the system overview described in Section 7.2 related to the basic distribution. The offloading timeline described in Section 7.3 highlights the function of the cache on the remote side. A remote-side cache serves as a collective storage for output execution states from already executed application parts, avoiding a repeated execution of previously run application parts. Making the offloading framework described in Section 7.4 aware of a cache on the remote side, the offloading client queries the introduced cache for a related output execution state of an application part instead of executing it on an offloading service. In case of a cache hit, the caching service that provides the cache immediately sends the output execution state to the offloading client. Otherwise, in case of a cache miss, the offloading client distributes the application part to an offloading service just like the basic distribution. To evaluate the introduced overhead and gained benefits of the caching-aware distribution compared to the basic distribution, Section 7.5 presents the evaluation of an implemented prototype, including the evaluation setup and the evaluation results. Last, Section 7.6 summarizes the main facts of an optimized code offloading through cooperative caching presented in this chapter.

7.1. Caching-aware Distribution

The offloading client of the basic distribution presented in detail in Subsection 4.5.2 identifies an application part for a distribution, gathers its input execution state, and
determines its execution side. Afterwards, in case of a remote execution on an offloading service, the offloading client sends the required information to the offloading service and waits in idle mode during the remote execution. Finally, it receives the output execution state from the offloading service, installs the contained information on its hardware platform, and continues the execution of the application. Regarding these eight substeps – identify, gather, determine, send, wait, receive, install, and continue – on the offloading client for the basic distribution, each substep introduces either computation or communication overhead to the offloading client. In case of a distribution of an application part, the process of code offloading – including the eight substeps – significantly increases the efficiency of the offloading client. In case of a local execution of an application part, the process of code offloading – now, only including the first three substeps – introduces an overhead to the offloading system. Please note that the basic distribution proposed in Chapter 4 improves these first three substeps, keeping the overhead low. Regarding the remaining five substeps – send, wait, receive, install, and continue – on the offloading client, the offloading client has to perform immutably the last two substeps of installing and continuing. Consequently, we propose the approach of a caching-aware distribution. The caching-aware distribution improves on the offloading client the substeps of sending, waiting, and receiving from the basic distribution with a cache. The cache serves as a collective storage for output execution states from already executed application parts on an offloading client or an offloading service to avoid a repeated execution of previously run application parts.

Before the substep of sending the input execution state to an offloading service, the offloading client just queries the introduced cache. Querying the cache for an output execution state of an application part, the cache starts a lookup in its storage of output execution states from already executed application parts. In case of a cache hit, the cache responds to the offloading client with the related output execution state, avoiding a repeated execution of a previously run application part on the offloading service. As the offloading client immediately receives the output execution state for the application part, it skips the substeps of sending, waiting, and receiving and proceeds with the substeps of installing and continuing. In case of a cache miss, the cache does not possess the related output execution state for an application part. Thus, the offloading client has to proceed with the substeps of sending, waiting, receiving, installing, and continuing. The caching-aware distribution benefits from previous executions of application parts by replacing the time cost of distribution – send, wait, and receive – with the space cost of caching – query, wait, and receive.

In general, there are multiple locations possible for the cache of the caching-aware distribution like directly on the hardware platform of an offloading client, at any place
in the infrastructure (cf. at the Fog [BMZA12]), or at a commercial cloud data center. A cache located on the hardware platform of an offloading client has the lowest delay of querying the cache, but introduces computation and storage overhead for a local processing and storing of the cache. Due to the single usage by an offloading client, only one client fills the cache with output execution states after an execution or a distribution of application parts. Moving the location of the cache to the infrastructure, the delay of communicating with the cache increases dependent on the link quality of the network. However, a cache located in the infrastructure replaces the computation and storage overhead for a cache located on the hardware platform of an offloading client with the communication overhead and monetary cost. The monetary cost arises due to the utilization of resources located in the infrastructure. Now, several offloading clients fill the cache with output execution states, increasing the probability for a cache hit. Going one step further, a cache located at a commercial cloud data center differs from a cache located in the infrastructure in the delay of communicating with the cache and the number of offloading clients filling the cache. However, the communication delay increases slightly and the number of participants considerably.

As the benefits of the caching-aware distribution directly depends on the fill level of the cache, the caching-aware distribution utilizes a cache located at a commercial cloud data center. Thus, the caching-aware distribution benefits from a collective sharing of output execution states cached on the remote side, where multiple offloading clients distribute application parts, filling the remote cache over time. Summarizing, we make the following contributions: (1) A formulation of the problem to make the basic distribution aware of a collective sharing of output execution states based on a cache; (2) an adaptive algorithm that solves the caching-aware problem increasing the efficiency of the basic distribution; (3) an implementation of the adaptive approach for the caching-aware distribution; and (4) an evaluation of the overhead and benefit for the caching-aware distribution based on real-world measurements, where different mobile devices execute different mobile applications.

7.2. System Overview

The caching-aware distribution extends the basic distribution with a cache on the remote side. To this end, Subsection 7.2.1 describes the changes to the system model, Subsection 7.2.2 to the problem statement, and Subsection 7.2.3 to the system components related to the basic distribution (cf. Section 4.2).
7. Optimized Code Offloading through Cooperative Caching

7.2.1. System Model

The system model of the caching-aware distribution extends the basic distribution with a cache on the remote side. Thus, an offloading client on a resource $\Xi(\xi_l)$ distributes application parts to an offloading service on a resource $\Xi(\xi_r)$ and queries a cache on a resource $\Xi(\xi_c)$. Providing a cache to offloading clients and offloading services, the performance characteristic $Q_{\text{chlu}}^{\Xi(\xi_c)}$ for a resource $\Xi(\xi_c)$ is a performance factor, indicating how much time the resource requires for processing a cache query. Moreover, the cost implication $Q_{\text{query}}^{\Xi(\xi_c)}$ for a resource $\Xi(\xi_c)$ is the processing fee, indicating how much monetary cost the resource charges per cache query. Utilizing a cache provided by a resource $\Xi(\xi_c)$, the load metrics $Q_{\text{size}}^{I_{\text{state}}}(A_{\alpha_o})$ and $Q_{\text{chms}}^{\Xi(\xi)}$ indicates the total number of bytes for a cache query of the input execution state $I_{\text{state}}(A_{\alpha_o})$ and for the information about the cache miss. For the communication network, the offloading client communicates with the cache via a bidirectional link $\Lambda(\xi_l;\xi_c)$, where the offloading service and the cache communicate with each other via a bidirectional link $\Lambda(\xi_r;\xi_c)$. Please note that the communication network consists for $\Lambda(\xi_l;\xi_c)$ of wireless and fixed networks, whereas it consists for $\Lambda(\xi_r;\xi_c)$ of a fixed network.

7.2.2. Problem Statement

A remote execution from the basic distribution – identifying, gathering, determining, sending, waiting, receiving, installing, and continuing – replaces a local execution of an application part if it is worthwhile. The caching-aware distribution introduces the substep of "querying" a cache before "sending" information to an offloading service, substituting "sending" in case of a cache hit. To keep the resource efficiency and the user experience high, the caching-aware distribution minimizes the cost function $f_w$ (cf. Section 3.2) subject to the additional constraint $T_{\text{max}}(A_{\alpha_o},\Xi(\xi))$ on the maximum time for an execution of an application part. $T_{\text{max}}(A_{\alpha_o},\Xi(\xi))$ is the maximum time tolerated for an execution of the application part $A_{\alpha_o}$ on $\Xi(\xi)$, ensuring a minimal responsiveness of an application. For instance, a distribution in case of a limited quality of the communication network keeps the energy consumed on the offloading client low but increases the overall time for an execution of an application part. Formally, the caching-aware distribution changes the problem stated in Equation 3.1 as follows:

$$\min_{\xi} f_w(A_{\alpha_o},\Xi(\xi))$$

s.t. $T(A_{\alpha_o},\Xi(\xi)) \leq T_{\text{max}}(A_{\alpha_o},\Xi(\xi))$  \hspace{1cm} (7.1)
As an end user sets the constraint $T^{\text{max}}(A_\alpha, \Xi(\xi))$ of the maximum time for the execution, we set the user-defined weight $w_e$ for the execution time to 0. Thus, the cost function $f_w$ is the sum of the energy consumption multiplied by the weight $w_e$ and the monetary cost multiplied by the weight $w_c$, simplifying the equation as follows:

$$\min_{\xi} \quad w_e \cdot E(A_\alpha, \Xi(\xi)) + w_c \cdot C(A_\alpha, \Xi(\xi))$$
$$\text{s.t.} \quad T(A_\alpha, \Xi(\xi)) \leq T^{\text{max}}(A_\alpha, \Xi(\xi))$$ (7.2)

Having a closer look on the execution time $T(A_\alpha, \Xi(\xi))$, each substep of a remote execution with caching – identifying, gathering, determining, sending/querying, waiting, receiving, installing, and continuing – takes time on the offloading client. In detail, a local execution of the application part on the offloading client takes time for identifying, gathering, determining, and continuing. A cache hit from the cache takes time for the eight substeps of a cache hit, but receives a small message about the cache miss instead of the output execution state. Thus, it additionally takes time for a remote execution on the offloading service after the receive of the message about the cache miss. As a result, the execution time $T(A_\alpha, \Xi(\xi))$ is either the time for a local execution on the resource $\Xi(\xi_l)$, a cache hit from the cache on the resource $\Xi(\xi_c)$, a remote execution on the resource $\Xi(\xi_r)$, or a cache miss from the cache on the resource $\Xi(\xi_c)$. Please note that Section 3.2 defines the time $T_{\text{local}}(A_\alpha, \Xi(\xi_l))$ for an execution on a resource $\Xi(\xi_l)$ and $T_{\text{remote}}(A_\alpha, \Xi(\xi_r))$ for a distribution to a resource $\Xi(\xi_r)$, consisting of the execution time $T_{\text{send}}(A_\alpha, \Xi(\xi_r))$, $T_{\text{wait}}(A_\alpha, \Xi(\xi_r))$, and $T_{\text{recv}}(A_\alpha, \Xi(\xi_r))$. For a cache hit, $T_{\text{chhit}}(A_\alpha, \Xi(\xi_c))$ sums up the execution time $T_{\text{query}}(A_\alpha, \Xi(\xi_c))$ for querying the cache, $T_{\text{chlu}}(A_\alpha, \Xi(\xi_c))$ for waiting until the cache processed the cache query, and $T_{\text{recv}}(A_\alpha, \Xi(\xi_c))$ for receiving the output execution state from the cache. For a cache miss, $T_{\text{chms}}(A_\alpha, \Xi(\xi))$ sums up the execution time $T_{\text{query}}(A_\alpha, \Xi(\xi_c))$ for querying the cache, $T_{\text{chlu}}(A_\alpha, \Xi(\xi_c))$ for processing the cache query at the cache, $T_{\text{recv}}(A_\alpha, \Xi(\xi_c))$ for receiving the information about the cache miss, and $T_{\text{remote}}(A_\alpha, \Xi(\xi_r))$ for executing the application part $A_\alpha$ remotely on the resource $\Xi(\xi_r)$.

$$T_{\text{query}}(A_\alpha, \Xi(\xi_c)) = \frac{Q_{\text{size}} L_{\text{late}}(A_\alpha)}{B_{\text{up}}} + L_{\text{up}}$$
$$T_{\text{chhit}}(A_\alpha, \Xi(\xi_c)) = Q_{\text{chhit}}$$
$$T_{\text{chms}}(A_\alpha, \Xi(\xi_c)) = \frac{Q_{\text{chms}} L_{\text{down}}}{B_{\text{down}}} + L_{\text{down}}$$
7. Optimized Code Offloading through Cooperative Caching

Like the time taken for each substep, each substep also consumes energy on the offloading client. In detail, the energy consumption $E(A_{\alpha_o}, \Xi(\xi))$ is the energy consumed for a local execution on the resource $\Xi(\xi)$, a cache hit from the cache on the resource $\Xi(\xi_c)$, a remote execution on the resource $\Xi(\xi_r)$, or a cache miss from the cache on the resource $\Xi(\xi_r)$. Again, please note that Section 3.2 defines the energy consumption $E_{\text{local}}(A_{\alpha_o}, \Xi(\xi))$ consumed for an execution on a resource $\Xi(\xi)$ and $E_{\text{remote}}(A_{\alpha_o}, \Xi(\xi_r))$ for a distribution to a resource $\Xi(\xi_r)$, consisting of the energy consumption $E_{\text{send}}(A_{\alpha_o}, \Xi(\xi))$, $E_{\text{wait}}(A_{\alpha_o}, \Xi(\xi_r))$, and $E_{\text{recv}}(A_{\alpha_o}, \Xi(\xi_r))$. For a cache hit, $E_{\text{chhit}}(A_{\alpha_o}, \Xi(\xi))$ sums up the energy consumption $E_{\text{query}}(A_{\alpha_o}, \Xi(\xi_c))$ for querying the cache, $E_{\text{chlu}}(A_{\alpha_o}, \Xi(\xi_c))$ for waiting until the cache processed the query, and $E_{\text{recv}}(A_{\alpha_o}, \Xi(\xi))$ for receiving from the cache the output execution state. For a cache miss, $E_{\text{chms}}(A_{\alpha_o}, \Xi(\xi))$ sums up the energy consumption $E_{\text{query}}(A_{\alpha_o}, \Xi(\xi_c))$ for querying the cache, $E_{\text{chlu}}(A_{\alpha_o}, \Xi(\xi_c))$ for waiting until the cache processed the query, $E_{\text{chms}}(A_{\alpha_o}, \Xi(\xi_c))$ for receiving the output execution state, and $E_{\text{remote}}(A_{\alpha_o}, \Xi(\xi_r))$ for executing $A_{\alpha_o}$ remotely on the resource $\Xi(\xi_r)$.

$$
E_{\text{query}}(A_{\alpha_o}, \Xi(\xi)) = T_{\text{query}}(A_{\alpha_o}, \Xi(\xi_c)) \cdot E_{\text{send}}^\Xi(\xi)
$$
$$
E_{\text{chlu}}(A_{\alpha_o}, \Xi(\xi_c)) = T_{\text{chlu}}(A_{\alpha_o}, \Xi(\xi_c)) \cdot E_{\text{wait}}^\Xi(\xi)
$$
$$
E_{\text{chms}}(A_{\alpha_o}, \Xi(\xi_c)) = T_{\text{chms}}^\Xi(\xi) \cdot E_{\text{recv}}(A_{\alpha_o}, \Xi(\xi))
$$

Having a closer look on the monetary cost $C(A_{\alpha_o}, \Xi(\xi))$, a cache hit from the cache on the resource $\Xi(\xi_c)$ (querying, waiting, and receiving), a remote execution on the resource $\Xi(\xi_r)$ (sending, waiting, and receiving), and a cache miss from the cache on the resource $\Xi(\xi_r)$ (querying, waiting, receiving, sending, waiting, and receiving) raise monetary cost for the offloading client. Again, please note that Section 3.2 defines the monetary cost $C_{\text{remote}}(A_{\alpha_o}, \Xi(\xi_r))$ raised for a distribution to a resource $\Xi(\xi_r)$, consisting of the monetary cost $C_{\text{recv}}(A_{\alpha_o}, \Xi(\xi_r))$, $C_{\text{wait}}(A_{\alpha_o}, \Xi(\xi_r))$, and $C_{\text{send}}(A_{\alpha_o}, \Xi(\xi_r))$. For a cache hit, $C_{\text{chhit}}(A_{\alpha_o}, \Xi(\xi))$ sums up the monetary cost $C_{\text{query}}(A_{\alpha_o}, \Xi(\xi_c))$ for querying the cache, $C_{\text{chlu}}(A_{\alpha_o}, \Xi(\xi_c))$ for processing the query on the cache, and $C_{\text{send}}(A_{\alpha_o}, \Xi(\xi))$ for receiving the output execution state from the cache. For a cache miss, $C_{\text{chms}}(A_{\alpha_o}, \Xi(\xi))$ sums up the monetary cost $C_{\text{query}}(A_{\alpha_o}, \Xi(\xi_c))$ for querying the cache, $C_{\text{chlu}}(A_{\alpha_o}, \Xi(\xi_c))$ for processing the query on the cache, $C_{\text{chms}}(A_{\alpha_o}, \Xi(\xi_c))$ for receiving the information about the cache miss from the cache, and $C_{\text{remote}}(A_{\alpha_o}, \Xi(\xi_r))$ for executing $A_{\alpha_o}$ remotely on the resource $\Xi(\xi_r)$.

$$
C_{\text{query}}(A_{\alpha_o}, \Xi(\xi_c)) = Q_{\text{size}}(A_{\alpha_o}) \cdot C_{\text{recv}}^{\Lambda(\xi_l; \xi_c)}
$$
$$
C_{\text{chlu}}(A_{\alpha_o}, \Xi(\xi_c)) = Q_{\text{query}}^{\Xi(\xi)} \cdot C_{\text{query}}
$$
$$
C_{\text{chms}}(A_{\alpha_o}, \Xi(\xi_c)) = Q_{\text{chms}}^{\Xi(\xi)} \cdot C_{\text{send}}^{\Lambda(\xi_l; \xi_c)}
$$

172
7.2. System Overview

Figure 7.1.: The system components for the caching-aware distribution, where an offloading client offloads code via a communication network to an offloading service. A caching service provides a cache of output execution states.

Summarizing, the caching-aware distribution may increase the efficiency of the offloading client compared to the basic distribution by querying a cache for an application part instead of executing it on an offloading service. If the cache possesses a corresponding value in its storage (cache hit), it responds to the offloading client with the output execution state. Optimally, each query of an offloading client to the cache corresponds to a cache hit that significantly reduces the execution time, energy consumption, and monetary cost for an application part. In case of a cache miss, additional time, energy, and money arise for caching and distributing. Thus, the substeps required for caching introduces some overhead compared to the basic distribution.

7.2.3. System Components

Figure 7.1 shows the integration of a caching service to the system components of the basic distribution, where the caching service functions as a cache on the remote side. It is a classic server machine, hosted in a (commercial) cloud data center, providing Caching as a Service (CaaS) to offloading clients and offloading services with a pay-as-you-go cost model. Due to the pay-as-you-go cost model, each cache query or cache insert causes monetary cost for an offloading client or an offloading service that queries
7. Optimized Code Offloading through Cooperative Caching

or inserts values (cf. Google’s Memcache [Pla16]). To handle cache queries or cache inserts, the caching service stores key-value pairs in a database, where a key identifies globally uniquely its corresponding value. Offloading clients and offloading services query the caching service for key-value pairs and/or insert key-value pairs into it.

7.3. Offloading Timeline

Figure 7.2 shows a timeline for the caching-aware distribution that extends the basic distribution described in Section 4.4 with a cache on the remote side (caching service). First, an offloading client sends a cache request to the caching service, requesting the monetary cost charged for a utilization of the caching service. On a receive of a cache request, the caching service creates a cache response that contains the monetary cost for its utilization and sends the cache response to the offloading client. Like for the basic distribution, the offloading client identifies an application part that is feasible for a distribution based on the offload-specific annotation offloadable. Invoking an annotated application part, the offloading client gathers the input execution state of
7.3. Offloading Timeline

the Java method and creates an offload request containing the input execution state. Afterwards, the offloading client determines the execution side for the Java method.

To make the basic distribution aware of caching, the caching-aware distribution additionally determines whether to query the caching service before either executing the Java method locally or distributing it to the offloading service. In case of querying the caching service, the offloading client creates three hash values for a Java method: (1) A hash value from the complete name of the application like `app_name-v1.3`, (2) a hash value from the entire signature of the Java method like `d.s.Klass.call(Ljava.lang.String;J[F)S`, and (3) a hash value of the offload request. For the computation of the hash values, the offloading client utilizes the efficient function DJB (Daniel Julius Bernstein)\(^1\) for hash computation (cf. Henke et al. [HSZ08]). These three hash values provide a proper mapping between the call of a Java method within an application and its right output execution state, constituting an Identifier (ID) for the call of a Java method within an application. To this end, a cache request only consists of the three hash values that the offloading client sends to the caching service querying for a corresponding output execution state.

Each time the caching service receives a cache request from an offloading client, it starts a cache lookup in its local database based on the received ID (`key`). The local database is a (huge) hash map that stores key-value pairs requiring \(O(n)\) for the storage space and in average \(O(1)\) (in worst case \(O(n)\)) for searching, inserting, or deleting values [CSRL09]. If the database contains a corresponding output execution state (`value`) for the received key (cache hit), the caching service immediately sends the offload response to the offloading client. Otherwise, it sends a small message to the offloading client indicating the non-existence of a key-value pair (cache miss).

In case of a cache hit, the offloading client receives the offload response, installs the contained information, and continues the execution of the application. In case of a cache miss, the offloading client receives the information of a cache miss and starts either an execution of the Java method locally or a distribution of the Java method to the offloading service as the basic distribution (Figure 7.2).

To fill the cache of the caching service with key-value pairs, offloading clients and offloading services send after a local execution of a Java method its ID together with the offload response to the caching service (cache insert). Then, the caching service updates its local database with the key-value pairs proactively, inserting the appropriate entries into the hash map. Regarding the memory limit of a cache storage, a caching service has to start a replacement strategy in case of running out of space, where the caching service utilizes the common replacement strategy of Least-Recently-Used (LRU) [JS94].

\[^1\] Professor Daniel J. Bernstein invented the algorithm for the hash function.
7. Optimized Code Offloading through Cooperative Caching

7.4. Offloading Framework

The caching-aware distribution extends the offloading framework of the basic distribution described in Section 4.5 by modifying the offloading client (cf. Subsection 7.4.1) and the offloading service (cf. Subsection 7.4.2). Beside the modifications to the offloading client and the offloading service, it adds the caching service (cf. Subsection 7.4.3).

7.4.1. Offloading Client

On the offloading client, the caching-aware distribution only adjusts the offload controller to send cache inserts and cache requests (cf. Subsection 4.5.2). To send a cache insert after a local execution of an application part that is offloadable, the execution controller on the offloading client invokes the offload controller due to the execution of the offload-specific Java bytecode instruction \texttt{offload\_end}. Afterwards, the offload controller generates a cache insert for the application part and sends it via the service connector to the caching service. To send a cache request before a local execution or the basic distribution of an application part that is offloadable, the offload controller on the offloading client minimizes the cost function from Equation 7.1 at its invocation from the execution controller due to the execution of the offload-specific Java bytecode instruction \texttt{offload}. Please note that the execution time $T(A_{\alpha o}, \Xi(\xi))$, the energy consumption $E(A_{\alpha o}, \Xi(\xi))$, and the monetary cost $C(A_{\alpha o}, \Xi(\xi))$ depend in general on dynamic parameters that the offload controller only knows at run-time. For instance, an end user playing a chess game determines the parameters of the Java method that is offloadable by moving a chess piece during the play. Thus, the offload controller has to minimize online the cost function at run-time based on the current parameters.

Algorithm 7.1 visualizes the decision making of the offload controller on the offloading client minimizing the cost function at run-time. To this end, it first considers for a Java method $A_{\alpha o}$ the tradeoff between a local execution on the hardware platform of the offloading client or receiving an offload response from the caching service due to a cache request (cf. Line 4). The offload controller executes the Java method locally in two cases: (1) If the sum of the energy consumption $W_{cht}^E$ and the monetary cost $W_{cht}^C$ for a cache hit is equal or higher than the energy consumption $W_{ld}^E$ for a local execution or (2) the execution time $W_{cht}^T$ for a cache hit is equal or higher than the maximum time $T_{cht}^{\max}(A_{\alpha o}, \Xi(\xi))$ for its execution. For instance, in case of a bad quality of the communication network (e.g., due to a high latency of the wireless link), a local execution is more beneficial than querying the caching service for an offload response.

In case the \textit{if-statement} in Line 4 of Algorithm 7.1 is \textbf{true}, the offloading client sends
Algorithm 7.1: Algorithm of the decision making for the caching-aware distribution on the offload controller of the offloading client.

a cache request to the caching service. The caching service responses to the offloading client with the cache response containing either the offload response (cache hit) or the information about the cache miss. In case of a cache hit, the offload controller on the offloading client installs the contained information and continues the execution of the application (cf. Line 6). In case of a cache miss, the offload controller considers a further tradeoff between a local execution on the offloading client or a remote execution on the offloading service (cf. Line 11). Like for the first tradeoff, the offload controller executes the Java method locally in two cases: (1) If the sum of the energy consumption $W_{rmt}^{E}$ and the monetary cost $W_{rmt}^{C}$ for a remote execution is equal or higher than the energy consumption $W_{ld}^{E}$ for a local execution or (2) the execution time $W_{rmt}^{T}$ for a remote execution is equal or higher than the maximum time $T_{max}^{A}(A_\alpha, \Xi(\xi))$ for its execution.

In case the if-statement in Line 11 is true, the offloading client sends an offload request to the offloading service. After the remote execution of the offload request on the offloading service, the offloading service responses to the offloading client with the offload response. On receiving the offload response, the offloading client installs the contained information and continues the execution of the application (cf. Line 12).
7. Optimized Code Offloading through Cooperative Caching

7.4.2. Offloading Service

For the offloading service, the caching-aware distribution only adjusts the offload controller and adds a service connector to send cache inserts (cf. Subsection 4.5.3). To send a cache insert after an execution of an application part that is offloadable, the execution controller invokes the offload controller due to the execution of the offload-specific Java bytecode instruction \texttt{offload\_end}. Afterwards, the offload controller generates a cache insert for the application part and sends it via the service connector to the caching service. Please note that the service connector corresponds to a service connector on an offloading client, providing the same functionality.

7.4.3. Caching Service

Figure 7.3 shows the runtime environment of the caching service, providing the cache-side functionality for the caching-aware distribution. It consists of a cache interface, a cache connector, a cache controller, and a cache database.

\textbf{Cache Interface} The cache controller of the caching service invokes the cache interface to retrieve the monetary cost for querying the caching service of an output execution state and inserting an input execution state to the caching service. To this
end, the cache interface provides an interface – more precisely text entry boxes, where a provider of the cache defines the monetary cost.

**Cache Connector** The cache connector of the caching service receives cache requests, cache inserts, and cache queries from offloading clients and offloading services. Moreover, it sends cache responses to offloading clients and offloading services. Receiving a cache request, an offloading client or an offloading service requests the monetary cost for querying and inserting of the caching service. To this end, the cache connector forwards the cache request to the cache controller. Receiving a cache insert, an offloading client or an offloading service updates the caching service with an output execution state after an execution of an application part. Thus, the cache connector forwards the cache insert to the cache controller. Receiving a cache query, an offloading client or an offloading service queries the caching service for an output execution state before an execution of an application part. To this end, the cache connector forwards the cache query to the cache controller. Receiving a cache response, the cache controller responses to a cache request, cache insert, or a cache query. Thus, the cache connector forwards it to the corresponding offloading client or offloading service.

**Cache Controller** The cache connector of the caching service invokes the cache controller due to the receive of a cache request, a cache insert, or a cache query. In case of a cache request, the cache controller creates a cache response – that contains the monetary cost of the caching service requested from an offloading client or an offloading service – and forwards the cache response to the cache connector. In case of a cache insert, the cache controller updates the cache database if it does not possess the key-value pair from the cache insert. In case of a cache query, the cache controller starts a lookup on the cache database if it possesses a corresponding value (output execution state) for the queried key. Afterwards, the cache controller creates a cache response – that contains either a corresponding value of the output execution state (cache hit) or the information about the cache miss – and forwards it to the cache connector.

**Cache Database** The cache database is a hash map that provides an efficient insert and lookup of key-value pairs for the caching service.

### 7.5. Evaluation

To evaluate the introduced overhead and gained benefits of the caching-aware distribution, we extended the implementation of the OpenJDK prototype for the basic
distribution (cf. Subsection 4.6.2). The extensions to the OpenJDK prototype comprise changes to the implementation of the JVM to provide the functionality required for the caching-aware distribution on the offloading client and the offloading service (cf. Section 7.4). Moreover, we implemented the caching service in a Java application, executed on an unaltered OpenJDK JRE. For the evaluation, mobile devices execute Java applications on an unaltered OpenJDK JRE (local execution), on an offload-aware OpenJDK JRE (basic distribution), and on a caching-aware OpenJDK JRE (caching-aware distribution). To this end, Subsection 7.5.1 describes the evaluation setup before Subsection 7.5.2 presents the evaluation results.

7.5.1. Setup

The evaluation setup for the caching-aware distribution extends the evaluation setup for the basic distribution on the OpenJDK described in Subsection 4.7.1 as follows:

The desktop computer that executes the offloading service also executes the Java application for a caching service on its hardware platform, possessing enough processing power for the execution of both services (cf. Section C.4).

The application evaluated for the caching-aware distribution are the chess game (cf. Section B.3) and the text-to-voice application (cf. Subsection B.5). The Java method that is offloadable from the chess game occupies only a small amount of memory on the cache of the caching service due to its small size of the output execution state. Regarding the huge number of possible configurations of the chess board\(^2\), however, the hit ratio of a cache for chess decreases fast with the number of rounds played for chess. Today's chess engines also look up favorable chess moves from a chess opening book in place of actually calculating the next best move on hardware. The Java method that is offloadable from the text-to-voice application occupies a large amount of memory on the cache of the caching service due to its big size of the output execution state. Regarding the general frequency of a word in a text (cf. Zipf's law [Pow98]), however, the hit ratio of a cache for text-to-voice is significantly higher than for chess. For instance, Fagan and Gençay analyzed in [FG10] the Brown University Standard Corpus of Present-Day American English (text collection)\(^3\) that consists of 1,014,312 words in total. Remarkably, half of the text collection only requires 135 different words.

For the monetary cost charged by the offloading service or the caching service, we

---

\(^2\)In the year 1950, Claude E. Shannon estimated the number of possible positions of chess pieces in the general order of roughly \(10^{43}\). [Sha50]

\(^3\)http://clu.uni.no/icame/manuals/BROWN/INDEX.HTM
consider the prices charged by Google\textsuperscript{4} or Amazon\textsuperscript{5} for the utilization of instances from their cloud platform. Both providers of cloud instances do not charge for data sent to the data center as well as data received from the data center. Together with the availability of a flat-rate data plans for end users, we set $C_{\text{recv}}^{\Lambda(\xi_l;\xi_c)}$ – and thus $C_{\text{qury}}(A_{\alpha_o},\Xi(\xi_c))$ and $C_{\text{recv}}(A_{\alpha_o},\Xi(\xi_c))$ – as well as $C_{\text{send}}^{\Lambda(\xi_l;\xi_c)}$ – and thus $C_{\text{send}}^{\text{chms}}(A_{\alpha_o},\Xi(\xi_c))$ and $C_{\text{send}}(A_{\alpha_o},\Xi(\xi_r))$ – to 0 (cf. Subsection 7.2.2). Typically, a cloud instance from Google or Amazon executes either the offloading service or the caching service. As the desktop computer executes both services, the monetary cost for $C_{\text{wait}}(A_{\alpha_o},\Xi(\xi_r))$ or $C_{\text{chlu}}(A_{\alpha_o},\Xi(\xi_c))$ equals the prices for an $m3.\text{medium}$ machine type from Amazon as well as an $n1-\text{standard-1}$ machine type from Google. The utilization of an $m3.\text{medium}$ machine type from Amazon causes a monetary cost of 0.070 $ per hour and an $n1-\text{standard-1}$ machine type from Google 0.063 $ per hour. As a provider of an offloading service or a caching service want to earn money with its service running on a cloud instance, we assume the sum of the price for a cloud instance from Amazon and Google, resulting in a monetary cost of $3.69 \cdot 10^{-5}$ ($= \frac{0.070+0.063}{60+60}$) $ per second for $C_{\text{wait}}(A_{\alpha_o},\Xi(\xi_r))$ and $C_{\text{chlu}}(A_{\alpha_o},\Xi(\xi_c))$. For instance, utilizing a Google $n1-\text{standard-1}$ instance, a provider amortizes the leasing cost of an hour (3600s) for the Google instance after 1705.26s of consecutive execution of application parts distributed to its service. As a provider of a caching service wants its cache to be filled by many participants, it does not charge monetary cost for cache inserts.

The evaluation compares the caching-aware distribution – named \textit{Cached} – in two configurations – named \textit{CachedEmpty} and \textit{CachedFull} – differing in the fill level of the cache on the caching service with the two approaches \textit{Local} and \textit{Baseline}. The approach \textit{Local} totally executes an application on an unaltered OpenJDK JRE, not distributing or caching a Java method that is offloadable. The approach \textit{Baseline} utilizes the basic distribution (cf. Section 4.1) distributing but not caching a Java method that is offloadable. The configuration \textit{CachedEmpty} of \textit{Cached} represents the worst case of the caching-aware distribution. The fill level of the cache from the caching service is empty and thus, the caching service responds to an offloading client that queries the caching service every time with a cache miss. The configuration \textit{CachedFull} of \textit{Cached} represents the optimal case of the caching-aware distribution. The cache from the caching service possesses each corresponding value for a key queried from an offloading client and thus, responses to an offloading client that queries the caching service every time with a cache hit. Based on the two configurations of \textit{CachedEmpty} and \textit{CachedFull}, the evaluation highlights for the caching-aware distribution its overhead and benefit.

\textsuperscript{4}https://cloud.google.com/compute
\textsuperscript{5}https://aws.amazon.com/ec2/pricing/?nc1=h_ls
7. Optimized Code Offloading through Cooperative Caching

Beside the evaluation of the overhead and benefit for the caching-aware distribution, the evaluation further gives an insight into the actual performance of the caching-aware distribution (Cached) with the help of a mobile scenario. To this end, each mobile device executes alternately on its hardware platform either the chess game or the text-to-voice application with different parameters like the configuration of the chess board or the transformed sentence. Both mobile devices utilize the same cache for the chess game and the text-to-voice application, where the fill level of the cache is empty at the start of the mobile scenario. To keep the results of Cached comparable with Baseline, only two mobile devices fill the cache of the caching service, because the hit ratio of the cache increases with the number of participating devices. The total space of the cache amounts to 1 GB that is large enough to store all of the values filled from the two mobile devices during the mobile scenario.

7.5.2. Results

Now, the following subsection presents the evaluation results, first describing the evaluation results for the chess game and afterwards the evaluation results for the text-to-voice application. Last, it describes the evaluation results for the mobile scenario.

Chess Game

For the netbook, Figure 7.4 shows the execution time, energy consumption, and monetary cost of the different approaches after 10 rounds of chess moves for each opening move. Subsection 4.7.2 describes the evaluation results for a local execution (134.70 s and 345.03 s; 1450.38 J and 3715.22 J; 0 $) and the basic distribution (24.44 s and 45.31 s; 241.10 J and 433.51 J; 0.0527260 $ and 0.1299504 $) of the chess game on the related OpenJDK JRE (cf. dark-red and red bars in Figure 7.4a, Figure 7.4b, and Figure 7.4c). Regarding the execution time, energy consumption, and monetary cost of CachedEmpty (cf. cyan bars in Figure 7.4a, Figure 7.4b, and Figure 7.4c), the caching-aware distribution keeps the (computation and communication) overhead introduced by caching low. Compared to the evaluation results of Baseline, the differences are minimal resulting in similar results for Baseline and CachedEmpty. For instance, the difference of the execution time is in the order of ten milliseconds owing to the additional queries to the caching service. The benefits of a cache that possesses all values queried in advance are significantly (cf. blue bars in Figure 7.4a, Figure 7.4b, and Figure 7.4c), because the time taken, the energy consumed, or the money raised for a cache query compared to a remote execution is very small. In detail, CachedFull takes at least 10.156 s and at most 10.255 s, consumes at least 109.34 J and at most
7.5. Evaluation

(a) Execution time on the netbook for the chess game.

(b) Energy consumption on the netbook for the chess game.

(c) Monetary cost on the netbook for the chess game.

Figure 7.4.: Execution time, energy consumption, and monetary cost on the netbook for the evaluation of the chess game with the different opening moves.
7. Optimized Code Offloading through Cooperative Caching

110.41 J, and raises at least 0.0000242 $ and at most 0.0000402 $. Compared to Baseline, Cached\textsubscript{Full} reduces the execution time, energy consumption, and monetary cost by up to 77.45%, 74.62%, and 99.98%, respectively. Compared to Local, it further reduces both execution time and energy consumption by up to 97.04%.

For the laptop, Figure 7.5 shows the execution time, energy consumption, and monetary cost of the different approaches after 10 rounds of chess moves for each opening move. Subsection 4.7.2 also describes the evaluation results for a local execution (69.019 s and 178.608 s; 3187.30 J and 8248.12 J; 0 $) and the basic distribution (23.912 s and 44.889 s; 890.93 J and 1519.67 J; 0.0499715 $ and 0.1296594 $) of the chess game on the related OpenJDK JRE (cf. dark-red and red bars in Figure 7.5a, Figure 7.5b, and Figure 7.5c). Like for the netbook, the caching-aware distribution keeps the introduced (computation and communication) overhead low (cf. cyan bars in Figure 7.5a, Figure 7.5b, and Figure 7.5c), where the maximum overhead of Cached\textsubscript{Empty} for the execution time is in average 0.237 s and for the energy consumption 10.964 J compared to Baseline. Regarding the benefits produced from Cached\textsubscript{Full} (cf. blue bars in Figure 7.5a, Figure 7.5b, and Figure 7.5c), it reduces the execution time (minimum: 9.637 s; maximum: 9.968 s), energy consumption (minimum: 444.89 J; maximum: 460.18 J), and monetary cost (minimum: 0.0000239 $; maximum: 0.0000479 $) by up to 78.46%, 70.63%, and 99.97% compared to Baseline. Compared to Local, it further reduces both execution time and energy consumption by up to 94.59%.

Summarizing, the caching-aware distribution further increases the benefits of the basic distribution that already improves a local execution of applications. As it only sends the ID to a caching service to query the cache, the total number of bytes transferred is low, especially compared to sending the offload request to the offloading service for a remote execution (order of bytes vs. multiple kilobytes or even megabytes). Thus, the caching-aware distribution outperforms the basic distribution in the optimal case (Cached\textsubscript{Full}), also keeping the introduced overhead low in the worst case (Cached\textsubscript{Empty}).

Text-to-Voice Application

For the netbook, Figure 7.6 shows the execution time, energy consumption, and monetary cost of the different approaches transforming 15 or 20 words from ThinkAir or MAUI to voice. Subsection 4.7.2 also describes the evaluation results for a local execution (65.532 s and 85.024 s; 705.64 J and 915.52 J; 0 $) and the basic distribution (32.519 s and 41.472 s; 332.97 J and 423.29 J; 0.0386653 $ and 0.0555976 $) of the text-to-voice application on the related OpenJDK JRE (cf. dark-red and red bars in Figure 7.6a, Figure 7.6b, and Figure 7.6c). The difference of the results between
7.5. Evaluation

(a) Execution time on the laptop for the chess game.

(b) Energy consumption on the laptop for the chess game.

(c) Monetary cost on the laptop for the chess game.

Figure 7.5.: Execution time, energy consumption, and monetary cost on the laptop for the evaluation of the chess game with the different opening moves.
7. Optimized Code Offloading through Cooperative Caching

Figure 7.6.: Execution time, energy consumption, and monetary cost on the netbook for the evaluation of the text-to-voice application.
7.5. Evaluation

Baseline and CachedEmpty highlights the (computation and communication) overhead introduced by caching, being for the execution time, energy consumption, and monetary cost very low (cf. cyan bars in Figure 7.6a, Figure 7.6b, and Figure 7.6c). Thus, the basic distribution (Baseline) and the caching-aware distribution (CachedEmpty) results in similar execution time (in the order of 100 milliseconds), energy consumption (in the order of 2 joules), and monetary cost (in the order of 0.1 cent). Regarding the benefits of a cache that possesses all values queried in advance (cf. blue bars in Figure 7.6a, Figure 7.6b, and Figure 7.6c), CachedFull takes the shortest time (minimum: 21.686s; maximum: 26.872s), consumes the least energy (minimum: 233.48J; maximum: 289.32J), and raises the lowest monetary cost (minimum: 0.0000596 $; maximum: 0.0001426 $) compared to the other approaches. CachedFull reduces the execution time, energy consumption, and monetary cost compared to Baseline by up to 35.39%, 31.85%, and 99.86%, respectively. Compared to Local, it further reduces both execution time and energy consumption by up to 68.39% and 68.40%.

For the laptop, Figure 7.7 shows the execution time, energy consumption, and monetary cost of the different approaches transforming the words to voice. Subsection 4.7.2 also describes the evaluation results for a local execution (26.083s and 33.731s; 280.86 J and 363.21 J; 0 $) and the naive basic distribution (29.304s and 38.508s; 298.35 J and 392.98 J; 0.03803 $ and 0.05479 $) of the text-to-voice application on the related OpenJDK JRE (cf. dark-red and red bars in Figure 7.7a, Figure 7.7b, and Figure 7.7c). Please note that the basic distribution would not distribute the Java method that is offloadable from the text-to-voice application just like CachedEmpty or CachedFull. However, both CachedEmpty and CachedFull can still send a query to the caching service. As a result, Baseline corresponds on the laptop to the naive basic distribution that still distributes the Java method that is offloadable. Regarding the (computation and communication) overhead introduced by caching, the caching-aware distribution also keeps the overhead on the laptop low (cf. cyan bars in Figure 7.7a, Figure 7.7b, and Figure 7.7c), resulting in an average increase of the execution time and energy consumption by up to 0.033s and 0.35J. Despite the fact that the Java method that is offloadable is a bad candidate for a distribution, CachedFull reduces the execution time (minimum: 18.156s; maximum: 23.333s) and energy consumption (minimum: 195.48 J; maximum: 251.18 J) compared to Local and thus, to Baseline (cf. blue bars in Figure 7.7a, Figure 7.7b, and Figure 7.7c). Raising monetary cost between 0.0000417 $ and 0.0001527 $, CachedFull lowers the execution time and energy consumption by up to 39.46% and 36.33% compared to Baseline. Compared to Local, it has a lower execution time and energy consumption of up to 31.62% and 31.62%.

Summarizing, the caching-aware distribution on the netbook further increases the
7. Optimized Code Offloading through Cooperative Caching

(a) Execution time on the laptop for the text-to-voice application.

(b) Energy consumption on the laptop for the text-to-voice application.

(c) Monetary cost on the laptop for the text-to-voice application.

Figure 7.7.: Execution time, energy consumption, and monetary cost on the laptop for the evaluation of the text-to-voice application.
7.5. Evaluation

(a) Execution time, energy consumption, and monetary cost on the netbook.

(b) Execution time, energy consumption, and monetary cost on the laptop.

Figure 7.8: Execution time, energy consumption, and monetary cost (a) on the netbook and (b) on the laptop for the evaluation of the mobile scenario.

benefits of the basic distribution that already improves a local execution of applications. Due to the fact that, on the one hand, the laptop is a more powerful device with a higher energy consumption and, on the other hand, the text-to-voice application has a big size of the output execution state, only the additional utilization of a caching service \((Cached\text{\_}{\text{Full}})\) reduces the execution time and energy consumption on the laptop lower than a local execution. Thus, on both mobile devices, the caching-aware distribution outperforms the basic distribution in the optimal case \((Cached\text{\_}{\text{Full}})\), also keeping the introduced overhead in the worst case low \((Cached\text{\_}{\text{Empty}})\).

Mobile Scenario

For the mobile scenario, each mobile device executes in parallel five times either the chess game or the text-to-voice application as follows. The randomly selected sequence on the netbook is a chess game with \(1\ \text{d}4\), a text-to-voice application with \textbf{ThinkAir: Idx=0 Cnt=15}, a text-to-voice application with \textbf{ThinkAir: Idx=0 Cnt=15}, a chess game with \(1\ \text{h}4\), and a chess game with \(1\ \text{c}4\). On the laptop, the randomly selected sequence is a text-to-voice application with \textbf{MAUI: Idx=15 Cnt=20}, a text-to-voice application with \textbf{ThinkAir: Idx=0 Cnt=15}, a chess game with \(1\ \text{a}4\), a chess game with \(1\ \text{h}4\), and a chess game with \(1\ \text{d}4\).

Figure 7.8 shows the execution time, energy consumption, and monetary cost on the netbook (cf. Figure 7.8a) and on the laptop (cf. Figure 7.8b) for the different approaches executing the sequence of applications for the mobile scenario. Regarding the execution time of \textit{Local}, \textit{Baseline}, and \textit{Cached} on both devices (cf. white-dotted bars in Figure 7.8), \textit{Local} takes the longest time for the execution, namely on the netbook 633.521s and on the laptop 340.424s. Compared to \textit{Local}, \textit{Baseline} reduces
7. Optimized Code Offloading through Cooperative Caching

the execution time by 76.46% on the netbook and by 53.72% on the laptop, resulting in 149.102s and 157.562s, respectively. At the start of the mobile scenario, the cache on the caching service is empty, where both mobile devices fill it up during the evaluation of the mobile scenario. As a result, the execution time of Baseline and Cached are similar, where Cached further reduces the execution time of Baseline by 6.41% on the netbook and 23.89% on the laptop as well as of Local by 97.95% on the netbook and 98.24% on the laptop. It takes on the netbook 139.550s and on the laptop 119.913s. The same applies to the energy consumption of the three approaches (cf. white-squared bars in Figure 7.8), where Local consumes the most energy with 6821.63 J on the netbook and 13623.17 J on the laptop. Baseline reduces the energy consumption of Local by 78.19% on the netbook and 70.94% on the laptop (1488.11 J and 3958.74 J, respectively), where Cached reduces it by 79.56% on the netbook and 78.60% on the laptop (1394.30 J and 2914.73 J). Regarding the monetary cost of Local, Baseline, and Cached on both devices (cf. white-waved bars in Figure 7.8), Local raises no monetary cost, executing the sequence of applications totally on the mobile devices. To reduce the execution time and energy consumption of Local, Baseline raises monetary cost of 0.2805151 $ on the netbook and 0.2984818 $ on the laptop due to the remote execution of the Java methods on the Distribution Service. Due to the cache hits during the mobile scenario, Cached reduces the monetary cost raised by 65.64% on the netbook and 69.81% on the laptop, resulting in 0.0963923 $ and 0.0901089 $.

Summarizing, the netbook and the laptop benefits significantly from the basic distribution of the Java methods that are offloadable in the mobile scenario. Extending the basic distribution with a cache on the remote side, however, the execution time and energy consumption reduces slightly due to an increased hit ratio of the cache during the mobile scenario. Moreover, the cache hits from the caching-aware distribution reduces significantly the monetary cost raised during the mobile scenario.

7.6. Summary

To provide an optimized code offloading through cooperative caching, the caching-aware distribution described in Section 7.1 extends the basic distribution with a cache on the remote side. The cache stores output execution states (value) of application parts previously executed on offloading clients or offloading services. As stated by Satyanarayanan in [Sat96], “caching plays a key role in mobile computing”, the caching-aware distribution adds a caching service to the system overview described in Section 7.2, adapting the system model, problem statement, and system components of the basic distribution. Regarding the offloading timeline described Section 7.3 the ad-
ditional utilization of a caching service for the basic distribution enables an offloading client to query the cache on the remote side for a corresponding output execution state of an application part that is offloadable. To this end, the offloading client just sends a cache query either before the local execution of the application part or the basic distribution of it. In case of a cache hit, the cache queries, on the one hand, avoids a repeated execution of application parts previously executed, and, on the other hand, reduces significantly the execution time required to get an output execution state. In case of a cache miss, the overhead introduced for cache queries is small, consisting only of hash values that identify the call of an application part. The offloading framework described in Section 7.4 provides the functionality required for the caching-aware distribution on the offloading client, on the offloading service, and on the caching service. To evaluate the overhead introduced and benefit gained by the caching-aware distribution compared to the basic distribution, the evaluation described in Section 7.5 analyzes different mobile devices with different applications based on the implemented prototype on the OpenJDK JRE. The evaluation results show that the caching-aware distribution increases the efficiency of the basic distribution. It reduces the execution time, energy consumption, and monetary cost for an application part that is offloadable compared to the basic distribution as well as to a local execution.
The basic distribution presented in detail in Chapter 4 offloads computation between a mobile device and a remote resource (two-tier architecture). It increases the efficiency on a mobile device and the execution speed of a mobile application. A common environment in today’s Mobile Cloud Computing (MCC), however, comprises highly distributed heterogeneous resources that can participate in code offloading. Such environments enlarge the number of available resources for a remote execution of application parts that are offloadable. To this end, this chapter presents a bubbling distribution in environments with multiple tiers described in Section 8.1. The bubbling distribution makes the basic distribution aware of an efficient and applicable code offloading to multiple resources with different performance characteristics and cost implications (multi-tier architecture). Thus, the system overview described in Section 8.2 with the system model, problem statement, and system components changes now from a two-tier architecture (basic distribution) to a multi-tier architecture (bubbling distribution). For a multi-tier architecture with multiple resources, it is very costly for a resource-poor mobile device to gather a global view of all available resources in all available tiers. For this reason, the bubbling distribution utilizes the concept of code bubbling. Code bubbling moves application parts that are offloadable dynamically and adaptively towards more powerful and more distant tiers, described in the offloading timeline in Section 8.3. Each tier makes autonomous decisions either to execute an application part in the tier or to forward it further to the next tier. To support such a recursive escalation of application parts along autonomous tiers, the bubbling distribution extends the offloading framework described in Section 8.4. The offloading framework provides a corresponding Application Programming Interface for the Java programming language to support code bubbling. Moreover, it also extends the offloading client and offloading service to support code bubbling and introduces a tier service to manage tiers. To evaluate the efficiency and applicability of a distribution with code bubbling compared to the basic distribution, Section 8.5 presents the evaluation of a
8. Code Offloading in Environments with Multiple Tiers

prototype, including the evaluation setup and the evaluation results. Last, Section 8.6 summarizes the main facts of a code offloading in environments with multiple tiers through code bubbling presented in this chapter.

8.1. Bubbling Distribution

The basic distribution presented in Section 4.1 of applications parts comprises an offloading client (first tier), an offloading service (second tier), and a (wireless) communication network that connects both tiers. In such a two-tier environment, the basic distribution can increase the energy efficiency of devices and execution speed of applications by executing resource-intensive application parts on more powerful resources (offloading service) in the infrastructure. To ensure a minimum execution time, energy consumption, and monetary cost, an offloading client regards the trade-off between the cost for a local execution and the cost for a remote execution of an application part (cf. Section 4.2). The cost for a local execution comprises the energy and time for an execution on the offloading client, whereas the cost for a remote execution comprises the energy, time, and monetary cost for an execution on an offloading service.

In the literature, researchers propose different approaches for offloading computation considering mostly two-tier environments or (seldom) three-tier environments. Compared to a two-tier environment, a three-tier environment, for instance, considers additionally a server instance at the edge of a network being close to the offloading client (cf. Fog Computing [BMZA12]). We assume that in the future, environments with multiple tiers will become relevant, where multiple classes of highly distributed heterogeneous resources are available for code offloading. Figure 8.1 shows such an environment with multiple tiers as an exemplary scenario. It consists of a smart watch (Tier 1) connected via Bluetooth to a smart phone (Tier 2). The smart phone in turn is connected via Wi-Fi to a smart car (Tier 3) that is connected to a server at the edge of the network (Tier 4) via a 3G cellular network. The server at the edge is finally connected to a server in a cloud data center (Tier 5) via a fixed network. In such a multi-tier environment, the heterogeneous devices differ significantly in terms of energy, compute, and communication resources, increasing typically from very limited (Tier 1) to virtually unlimited (Tier n).

The bubbling distribution targets such environments with multiple tiers that consists of highly distributed heterogeneous resources with different performance characteristics and cost implications. It utilizes a distributed architecture of autonomous tiers to fully exploit the capabilities of environments with multiple tiers. Each tier manages participating resources within its tier independently and decides autonomously whether
8.1. Bubbling Distribution

Figure 8.1.: An exemplary environment of multiple tiers. It consists of a smart watch (Tier 1) connected via Bluetooth to a smart phone (Tier 2). The smart phone in turn is connected via Wi-Fi to a smart car (Tier 3) that is connected to an edge server (Tier 4) via a 3G cellular network. The edge server is finally connected to a cloud server (Tier 5) via a fixed network.

to execute an application part within the tier or to forward it to the next upper tier. Due to the recursive escalation of application parts along autonomous tiers, an offloading client only interacts with its neighboring tier. At run-time, the offloading client forwards to its neighboring tier the request of offloading an application part and continues asynchronously the local execution of this application part (pessimistic approach). This decoupled and non-blocking distribution of application parts keeps the overhead on the offloading client low, not requiring a global view on the environment with multiple tiers. Moreover, the decentralized architecture based on autonomous tiers handles the increasingly connected environment in a simple, scalable, and flexible way. We call this concept of a recursive distribution of application parts within autonomous tiers code bubbling, since code rises towards higher tiers like bubbles towards the surface. Code bubbling keeps the distribution of application parts for an offloading client efficient, because it neither requires the interaction nor the exchange of state information with all tiers or resources in the environment. Moreover, the distribution decision on an offloading client is rather simple compared to the question on which resource to execute which application part, requiring a global view onto multiple tiers.

Summarizing, we make the following key contributions: (1) A framework for a distribution based on code bubbling in environments with multiple tiers that supports highly distributed heterogeneous resources with different performance characteristics and cost implications; (2) the concept of code bubbling that dynamically and adaptively distributes application parts in environments with multiple tiers based on a recursive escalation of code along autonomous tiers, where participating resources are unknown.
8. Code Offloading in Environments with Multiple Tiers

8.2. System Overview

The bubbling distribution is applicable in environments with multiple tiers, where each tier has zero or more resources available for code offloading. To this end, the system model for the bubbling distribution extends the general system model for multiple resources described in detail in Section 3.1 with tiers. It structures the multiple classes of highly distributed heterogeneous resources in several tiers $\Gamma(\gamma)$ with $\gamma = 0, 1, \ldots, n_\gamma - 1$. Each tier $\Gamma(\gamma)$ comprises a number of resources $\Xi(\gamma, \xi)$ with $\xi = 0, 1, \ldots, n_{\gamma, \xi} - 1$, where the performance characteristics and cost implications of a resource are the same as in the general system model. Thus, compared to the basic distribution, the problem statement of the bubbling distribution changes as follows:

$$\min_{\gamma, \xi} f_w(A_{\alpha_o}, \Xi(\gamma, \xi)) = \min_{\gamma, \xi} \left( w_t \cdot T(A_{\alpha_o}, \Xi(\gamma, \xi)) + w_e \cdot E(A_{\alpha_o}, \Xi(\gamma, \xi)) + w_c \cdot C(A_{\alpha_o}, \Xi(\gamma, \xi)) \right)$$

with

$$f_w(A_{\alpha_o}, \Xi(\gamma, \xi)) = w_t \cdot T(A_{\alpha_o}, \Xi(\gamma, \xi)) + w_e \cdot E(A_{\alpha_o}, \Xi(\gamma, \xi)) + w_c \cdot C(A_{\alpha_o}, \Xi(\gamma, \xi))$$

The execution time $T(A_{\alpha_o}, \Xi(\gamma, \xi))$, the energy consumption $E(A_{\alpha_o}, \Xi(\gamma, \xi))$, and the monetary cost $C(A_{\alpha_o}, \Xi(\gamma, \xi))$ for a resource $\Xi(\gamma, \xi)$ are calculated like the execution time $T(A_{\alpha_o}, \Xi(\xi))$, the energy consumption $E(A_{\alpha_o}, \Xi(\xi))$, and the monetary cost $C(A_{\alpha_o}, \Xi(\xi))$ for a resource $\Xi(\xi)$ (cf. Section 3.2).

To efficiently provide the bubbling distribution for environments with multiple tiers, the architecture of the bubbling distribution uses a multi-tiered hierarchy (cf. Figure 8.2), consisting of an offloading client, a tier service on each tier, an offloading service on each resource within a tier, and a communication network.
8.2. System Overview

Figure 8.2.: The system components for the bubbling distribution, where an offloading client (OC) offloads computation via a communication network to tier services (TSs) and accordingly to offloading services (OSs).

**Offloading Client:** The offloading client on a resource $\Xi(\gamma, \xi)$ controls the local execution of an application (cf. Subsection 3.3.1). Each time an execution of an application reaches an application part $A_{\alpha_o}$ that is offloadable, the offloading client gathers the input execution state $I_{\text{state}}(A_{\alpha_o})$ and the portable code $P_{\text{pcode}}(A_{\alpha})$ of the application part $A_{\alpha_o}$. Beside the application-related information of $I_{\text{state}}(A_{\alpha_o})$ and $P_{\text{pcode}}(A_{\alpha})$, the offloading client further gathers resource-related and user-related information. The resource-related information are the energy factors of sending bytes $E_{\Xi(\gamma, \xi)}^{\text{send}}$, waiting in idle mode $E_{\Xi(\gamma, \xi)}^{\text{wait}}$, and receiving bytes $E_{\Xi(\gamma, \xi)}^{\text{recv}}$ on the resource $\Xi(\gamma, \xi)$. The user-related information are the user-defined weights of $w_t$ for the execution time, $w_e$ for the energy consumption, and $w_c$ for the monetary cost. Based on the application-related, resource-related, and user-related information, the offloading client creates an offload request. The offload request contains all of the information required, on the one hand, to minimize the cost function $f_w$ on a tier service and, on the other hand, to execute the application part remotely on an offloading service.

**Tier Service:** A tier service manages centrally participating resources within its tier. For instance, in Figure 8.1, a smart car provides for its passengers a Wi-Fi Access Point (AP) and runs a tier service that possesses a direct connection to each (participating) resource within its tier. On the one hand, a tier service monitors the non-varying performance characteristic (cf. $P_{\text{ppwr}}^{\Xi(\gamma, \xi_m)}$) and cost implications (cf. $C_{\text{exec}}^{\Xi(\gamma, \xi_m)}$),
8. Code Offloading in Environments with Multiple Tiers

\[ C_{\text{send}}(\gamma_i, \xi_k; \gamma_j, \xi_m), \] and \[ C_{\text{recv}}(\gamma_i, \xi_k; \gamma_j, \xi_m) \] as well as the time-varying performance characteristics (cf. \( B_{\text{up}}(\gamma_i, \xi_k; \gamma_j, \xi_m), B_{\text{down}}(\gamma_i, \xi_k; \gamma_j, \xi_m), L_{\text{up}}(\gamma_i, \xi_k; \gamma_j, \xi_m), \) and \( L_{\text{down}}(\gamma_i, \xi_k; \gamma_j, \xi_m) \)) of participating resources (cf. Section 3.1). On the other hand, a tier service functions as a gateway to other tiers and plans the execution of offload requests for participating resources within its tier. Receiving an offload request, the tier service regards the application-related, resource-related, and user-related information, determining dynamically the best available resource within its tier.

**Offloading Service:** The offloading service on participating resources executes an offload request on its hardware platform (cf. Subsection 3.3.2), providing a lightweight execution engine for portable code. It transforms an input execution state \( I_{\text{state}}(A_{\alpha_o}) \) into an output execution state \( O_{\text{state}}(A_{\alpha_o}) \) based on the execution of portable code \( P_{\text{pcode}}(A_{\alpha_o}) \) on its hardware platform. As an offloading client distributes an application part to unknown remote resources, an offloading service guarantees the “real” execution of an application part based on a further instrumentation of Java bytecode instructions.

**Communication Network:** The communication network connects each offloading service with its tier service, each tier service with its neighboring tier service(s), and an offloading client with its neighboring tier service(s). Thus, the communication network consists of multiple wireless network(s) and/or fixed network(s).

8.3. Offloading Timeline

A resource that wants to distribute application parts to other resources runs an offloading client. If it also functions as an Access Point (AP) for other resources (cf. the smart phone in Figure 8.1 that a Bluetooth link connects to the smart watch), the distributing resource also runs a tier service managing offloading services from connected resources. Each time the execution of an application \( A \) reaches an application part \( A_{\alpha_o} \) that is offloadable, the execution of the application part annotated with \textbf{offloadable} invokes the offloading client that creates a corresponding offload request for \( A_{\alpha_o} \). Based on the cost function \( f_w \) (cf. Equation 8.1), the offloading client calculates the cost for a local execution of \( A_{\alpha_o} \) and adds it to the offload request. To keep the communication overhead and the waiting time low, the offloading client only submits a downgraded version of the offload request to the next tier service and continues the local execution of \( A_{\alpha_o} \). The downgraded offload request only contains the input size \( I_{\text{size}}(A_{\alpha_o}) \), the output size \( O_{\text{size}}(A_{\alpha_o}) \), and the code complexity \( P_{\text{exe}}(I_{\text{state}}(A_{\alpha_o}), P_{\text{pcode}}(A_{\alpha_o})) \) instead of the actual values of \( I_{\text{state}}(A_{\alpha_o}) \) and \( P_{\text{pcode}}(A_{\alpha_o}) \). The asynchronous submission of a
8.3. Offloading Timeline

downgraded offload request decouples the decision of a distribution from the offloading client into the infrastructure.

Managing offloading services from participating resources within a tier, each time a tier service receives a downgraded offload request, it determines the best available offloading service within its tier. To this end, it first calculates the value of the cost function for the available offloading services within its tier. As it currently does not know the actual bandwidth and latency of the link between the offloading client and its tier, it assumes an infinite bandwidth and a zero latency. Later on, the offloading client replaces these optimistic values for the bandwidth and latency with the proper bandwidth and latency. After the calculation of the value of the cost function, a tier service compares it with the value of the cost function for a local execution contained in the downgraded offload request. Only in case of a smaller value, the tier service sends to the offloading client a resource offer containing the performance characteristics and cost implications for an execution on the offloading service within its tier. Independently from the calculated value of the cost function, the tier service forwards the downgraded offload request to its next upper tier service. It finds its next upper tier service with the help of a service discovery, where the bubbling distribution does not rely on a specific service discovery protocol. Thus, an efficient and optimal discovery is out of scope of this dissertation. Forwarding the downgraded offload request to the next tier service, the offload request bubbles up the tiers, visiting recursively multiple tier services until it reaches the highest tier (cf. the cloud resources in Figure 8.1).

Due to the code bubbling of the bubbling distribution, an offloading client receives from multiple tier services at different points in time a resource offer. Receiving a resource offer, an offloading client measures simultaneously the bidirectional latency of the link to the corresponding tier service. Possessing the actual latency of the link, an offloading client incorporates the measured latency and a profiled bandwidth of the link into the value of the cost function offered from a tier service. Due to the local continuation of the application execution, an offloading client also recalculates the value of the cost function for finishing the execution of the application locally. Comparing the recalculated values, an offloading client decides whether to continue the local execution or to distribute the application part $A_{\alpha_o}$ to the offloading service in the tier. In case of a distribution of $A_{\alpha_o}$, an offloading client sends the input execution state $I_{\text{state}}(A_{\alpha_o})$ and the portable code $P_{\text{pcode}}(A_{\alpha_o})$ to the corresponding tier service, waiting for the output execution state $O_{\text{state}}(A_{\alpha_o})$ from the remote execution of $A_{\alpha_o}$.

Receiving the actual $I_{\text{state}}(A_{\alpha_o})$ and $P_{\text{pcode}}(A_{\alpha_o})$ of an offload request, a tier service allocates the offloading service offered from the resource offer and forwards the offload request to the offloading service. Moreover, it starts a monitoring of the execution on
the offloading service, to react on varying conditions in its tier like a crash failure with a redistribution of the offload request to another resource.

Receiving an offload request, an offloading service installs the input execution state \( I_{state}(A_{\alpha_0}) \) and starts the execution of the portable code \( P_{pcode}(A_{\alpha_0}) \). At the end of the execution, it sends the output execution state \( O_{state}(A_{\alpha_0}) \) to its tier service that subsequently sends \( O_{state}(A_{\alpha_0}) \) to the offloading client.

Receiving an output execution state \( O_{state}(A_{\alpha_0}) \), an offloading client interrupts the local execution of the portable code \( P_{pcode}(A_{\alpha_0}) \) and installs the output execution state \( O_{state}(A_{\alpha_0}) \). After the installation of \( O_{state}(A_{\alpha_0}) \), the offloading client jumps to the end of the local execution of \( P_{pcode}(A_{\alpha_0}) \) and continues the execution of the application.

### 8.4. Offloading Framework

The offloading framework for the bubbling distribution adjusts the offloading framework for the basic distribution described in Section 4.5. To provide the functionality required for the bubbling distribution, Subsection 8.4.1 describes the Application Programming Interface (API) for the Java programming language extended for the bubbling distribution. Afterwards, Subsection 8.4.2 and Subsection 8.4.3 presents the changes to the offloading client and to the offloading service, respectively. Last, Subsection 8.4.4 outlines the tier service for the bubbling distribution.

#### 8.4.1. Application Programming Interface

The bubbling distribution utilizes an extended Application Programming Interface (API) for the Java programming language (cf. Subsection 4.3.4) to abstract from the actual hardware architectures of smart phones, laptops, or server machines (e.g., x86 vs. ARM and 32-bit vs. 64-bit). Due to the extension of the Java technology for a distribution with code bubbling, the extended Java platform provides a platform-independent execution of Java bytecode instructions on different hardware architectures.

To keep the complexity for end users as well as application developers low, the extended Java API of the bubbling distribution hides the distributed and parallel nature of environments with multiple tiers. It relies on least interventions from an end user and an application developer. An end user only specifies the user-related information for the weights of execution time, energy consumption, and monetary cost (cf. Equation 8.1) via the user interface of the offloading client (cf. Subsection 4.5.2). An application developer simple extends a provided Java core class at development-time to enable the distribution of a Java method. The Java core class libraries like `java.lang.*` are essential
8.4. Offloading Framework

```java
package java.lang;
import java.io.Serializable;

public abstract class OffloadExecution implements Serializable {
    private static final long serialVersionUID = 42L;
    @Offloadable
    public abstract OffloadIO execute(OffloadIO inputs);
}
```

Listing 8.1: The source code of the abstract Java class `OffloadExecution` from the Java core package `java.lang`. The class `OffloadExecution` has an abstract Java method `execute` with the annotation `Offloadable` that utilizes the Java class `OffloadIO` as input and output.

libraries for using a Java Virtual Machine (JVM) of a Java Runtime Environment (JRE) (cf. Subsection 4.3.4). The additional core classes `Offloadable`, `OffloadExecution`, and `OffloadIO` in the core class library `java.lang` enable the bubbling distribution of application parts from an offloading client to offloading services. In detail, an application developer only extends the abstract Java class `OffloadExecution` that possesses the abstract Java method `execute` (cf. Listing 8.1). This method possesses the annotation `Offloadable` (cf. Section 4.4), marking a Java method as a suitable candidate for a remote execution. For instance, a suitable candidate only relies on the processing resource of a hardware platform not accessing a local sensor like Global Positioning System (GPS) for localization. Due to the annotation of Java methods that are offloadable, the extended Java compiler instruments the Java bytecode instructions of the Java method with `offload` and `offload_end` at development-time (cf. Subsection 4.5.1).

At run-time, the execution of the Java bytecode instructions `offload` or `offload_end` invokes the execution controller (cf. Subsection 4.5.2), creating an offload request for the Java method. Thus, an application developer implements an offload request by simply implementing the method `execute` of the Java core class `OffloadExecution`. The parameter and the return value of the Java method `execute` correspond both to the Java core class `OffloadIO` (cf. Listing 8.1). This core class serves as an efficient storage of the data for Java primitives, arrays, and objects of the parameters or the return value for a Java method (cf. Subsection 4.3.4).

For an efficient serialization and deserialization of the Java data types stored in the Java core class `OffloadIO`, it implements the standard Java `Serializable` interface of the Java core class libraries (cf. Subsection 4.3.4).

The execution of an offload request on an offloading service only utilizes data from the input execution state – e.g., the parameters of the Java method or the members of its declaring Java class. As a result, an application developer only references data
from the declaring Java class of the Java method `execute` during its execution. Based on this `Input \rightarrow Execution \rightarrow Output` processing of the extended Java API, an execution does not alter the execution state of an application outside of an offload request beside the resulting output. This guarantees the equality of a monolithic execution of an application on a local resource with a distributed execution of an application on multiple resources. Moreover, the `Input \rightarrow Execution \rightarrow Output` processing offers a resource-friendly execution on an offloading service, not needing to trace or monitor changes to the execution state during an execution.

### 8.4.2. Offloading Client

For a dynamic distribution of application parts, an offloading client has to obtain the application-related, resource-related, and network-related information.

**Application-related Information:** The application-related information required for an offload request comprises from an application part $A_\alpha$ the input execution state $I_{state}(A_\alpha)$ and its size $I_{size}(A_\alpha)$, the Java bytecode instructions $P_{pcode}(A_\alpha)$ and its code complexity $P_{exe}(I_{state}(A_\alpha), P_{pcode}(A_\alpha))$, and the size of the output execution state $O_{state}(A_\alpha)$ and its size $O_{size}(A_\alpha)$ (cf. Section 8.3). At the call of a Java method $A_\alpha$ that is offloadable, its input execution state $I_{state}(A_\alpha)$ and its Java bytecode instructions $P_{pcode}(A_\alpha)$ are immediately available from the state generator (cf. Subsection 4.5.2). Due to the implementation of the standard Java serialization of the Java core class `OffloadIO`, the size of the input execution state $I_{size}(A_\alpha)$ is also directly available by serializing the input execution state $I_{state}(A_\alpha)$. To obtain the code complexity $P_{exe}(I_{state}(A_\alpha), P_{pcode}(A_\alpha))$ of the Java bytecode instructions $P_{pcode}(A_\alpha)$ as well as the size of the output execution state $O_{state}(A_\alpha)$, the offload controller predicts the code complexity $P_{exe}(I_{state}(A_\alpha), P_{pcode}(A_\alpha))$ or the size $O_{size}(A_\alpha)$ with the help of the profile models built by the app profiler (cf. Subsection 4.5.2).

**Resource-related Information:** The resource-related information required for an offload request comprises on an offloading client on the resource $\Xi(\gamma_i, \xi_k)$ its execution speed of Java bytecode instructions $P_{ppwr}^{\Xi(\gamma_i, \xi_k)}$ and its energy factors $E_{exec}^{\Xi(\gamma_i, \xi_k)}$, $E_{send}^{\Xi(\gamma_i, \xi_k)}$, $E_{recv}^{\Xi(\gamma_i, \xi_k)}$, and $E_{wait}^{\Xi(\gamma_i, \xi_k)}$. Moreover, the resource-related information comprises the monetary cost $C_{exec}^{\Xi(\gamma_j, \xi_m)}$ for an offloading service on the resource $\Xi(\gamma_j, \xi_m)$. To obtain the performance characteristic $P_{ppwr}^{\Xi(\gamma_i, \xi_k)}$, an offloading client as well as an offloading service support an execution mode for the JVM that benchmarks the underlying capabilities of a resource including the architectural differences of participating resources (cf. Subsec-
8.4. Offloading Framework

This specific execution mode determines the execution speed of simple Java bytecode instructions like `add` or `mod` for Java primitive data types, array data types, and object data types (cf. Subsection 4.3.4). The outcome of this specific execution mode is a profile of a resource based on Java bytecode instructions, where this performance model determines the execution speed of Java bytecode instructions on the resource. To obtain the performance characteristics $E_{\text{exec}}(\gamma_i, \xi_k)$, $E_{\text{send}}(\gamma_i, \xi_k)$, and $E_{\text{recv}}(\gamma_i, \xi_k)$, an offloading client provides the device interface (cf. Subsection 4.5.2), where a device manufacturer defines these resource-specific factors of the energy consumption offline. To obtain the cost implication $C_{\text{exec}}(\gamma_j, \xi_m)$, an offloading service provides the provider interface (cf. Subsection 4.5.3), where a resource provider defines this resource-specific factor of the monetary cost either offline or online.

**Network-related Information:** The network-related information required for an offload request comprises for a link between two resources $\Xi(\gamma_i, \xi_k)$ (offloading client) and $\Xi(\gamma_j, \xi_m)$ (offloading service) the up and down latency $L_{\text{up}}(\gamma_i, \xi_k, \gamma_j, \xi_m)$ and $L_{\text{down}}(\gamma_i, \xi_k, \gamma_j, \xi_m)$, the up and down bandwidth $B_{\text{up}}(\gamma_i, \xi_k, \gamma_j, \xi_m)$ and $B_{\text{down}}(\gamma_i, \xi_k, \gamma_j, \xi_m)$, and the monetary cost $C_{\text{send}}(\gamma_i, \xi_k, \gamma_j, \xi_m)$ and $C_{\text{recv}}(\gamma_i, \xi_k, \gamma_j, \xi_m)$. To obtain the performance characteristic of the up and down latency and use sparingly link resources, the network monitor on an offloading client measures the up and down latency associated with the resource offers from available resources within a tier (cf. Section 8.3). Moreover, it measures the up and down bandwidth for each connection type (e.g., GSM, UMTS, LTE, or WiFi) in the background while an end user utilizes its resource like a smart phone during the day. Based on the bandwidth measured for each connection type, it calculates a weighted average based on the occurrence of the measured bandwidth. This avoids a periodic lavish measurement of the up and down bandwidth with message probing to multiple tiers (cf. Subsection 4.5.2). Combining the actual latency with the history-based bandwidth, the offload controller determines a profiled duration for the transmission of an input execution state and an output execution state. During the sending or receiving of the data for an input execution state or an output execution state, the network monitor monitors the actual bandwidth and reacts, for instance, in case of too low bandwidth by invoking the offload controller. Subsequently, the offload controller reacts on the too low quality of the link for a distribution with, for instance, a re-distribution to another resource or a local completion of the execution. To obtain the cost implications $C_{\text{send}}(\gamma_i, \xi_k, \gamma_j, \xi_m)$ and $C_{\text{recv}}(\gamma_i, \xi_k, \gamma_j, \xi_m)$, an offloading client provides the network interface (cf. Subsection 4.5.2), where a network provider defines the link-specific factor of the monetary cost either offline or online.
8. Code Offloading in Environments with Multiple Tiers

8.4.3. Offloading Service

The functionality provided by the offloading service is nearly the same as for the basic distribution described in detail in Subsection 4.5.3. As an offloading client does not know a priori – and maybe does not trust – the resource that executes the offloading service, the bubbling distribution extends the offloading service with a mechanism that provides the prove for an offloading client if the offloading service really executed an application part. To guarantee the “real” execution of an application part on an unknown resource, the bubbling distribution secures a remote execution based on the extended API for the Java programming language (cf. Subsection 8.4.1). To this end, the bubbling distribution utilizes a shared secret at the run-time level between an offloading client and offloading services. It utilizes a shared secret at the run-time level, because an attacker possibly knows the Java bytecode instructions of a Java method (application level) and can intercept the machine code produced by an offloading service during an execution of the Java method (system level). The shared secret is a large set of mathematical functions $f_i(x)$ with $i = 0, 1, \ldots$ only known to the run-time environments of an offloading client and an offloading service. During an execution of a Java method on an unknown remote resource, its offloading service calculates $f_i(x)$ in parallel. At the end of a remote execution, an offloading service hashes the result of $f_i(x)$, the values chosen for $i$ and $x$, the input execution state $I_{state}(A_{ao})$, the portable code $P_{pcode}(A_{ao})$, and the output execution state $O_{state}(A_{ao})$ based on the cryptographic hash function SHA-3 (Secure Hash Algorithm 3). After the hashing of the corresponding values, an offloading service sends the hash value, the values of $i$ and $x$, and the output execution state to the offloading client. The offloading client verifies the correctness of the output execution state by recalculating the result of $f_i(x)$ requiring $i$ and $x$, hashing the six values, and comparing the equality of the two hash values. Due to the inclusion of the result from $f_i(x)$ and the values of $i$ and $x$, an attacker cannot intercept the hash value and replace it with an own generated hash value due to the shared secret of $f_i(x)$. Due to the inclusion of $I_{state}(A_{ao})$, $P_{pcode}(A_{ao})$, and $O_{state}(A_{ao})$, an attacker cannot produce a valid hash value based on the “wrong” execution of a resource-friendly Java method, sparing its resources. As an attacker can trace the machine codes executed by an offloading service for a Java method and the mathematical function, an offloading service executes additionally from time to time unused machine codes to circumvent the comparison with a code trace from an execution on an unaltered JRE. Technically, the offload-aware Java compiler further inserts at development-time before branching instructions the offload-specific Java bytecode.

\footnote{https://www.nist.gov/node/555116?pub_id=919061}
8.4. Offloading Framework

Figure 8.3.: The runtime environment on the tier service for the bubbling distribution.

instruction offload_branch. At run-time, the execution of offload_branch invokes either the next step of the calculation for the mathematical function or unused machine codes. Summarizing, a remote resource has to actually execute the Java method to obtain the correct hash value. It cannot just fool an offloading client with an arbitrary output execution state, charging monetary cost for an execution that never happened.

8.4.4. Tier Service

Figure 8.3 shows the runtime environment of the tier service, providing the tier-side functionality for the bubbling distribution. It consists of a network monitor, a network interface, a service monitor, a service connector, a tier connector, and a tier controller. The network interface, service monitor, and service connector provides the same functionality as for the basic distribution described in detail in Subsection 4.5.3.

Network Monitor: The network monitor for the bubbling distribution is the same as the network monitor of the basic distribution described in detail in Subsection 4.5.3, offering to a tier service the capability to measure the link bandwidth and link latency.

Tier Connector: The tier controller of the tier service invokes the tier connector to handle the communication between the tier service and its neighboring tier service(s).
Moreover, it also invokes the tier connector to handle the communication between the tier service and an offloading client being also a tier. Receiving an offload request from its next lower tier service, the tier service forwards the offload request to the tier controller and to its next upper neighboring tier service. Receiving an input execution state and the Java bytecode instructions for an application part from an offloading client, the tier connector forwards both to the tier controller. Last, receiving from the tier controller a resource offer or an output execution state, the tier connector sends both to the corresponding offloading client.

**Tier Controller:** The tier controller of the tier service manages the participating offloading services within its tier, functioning as a gateway to other tiers. To this end, it plans the execution of received offload requests by minimizing Equation 8.1 with the application-related, resource-related, and user-related information contained in an offload request. To determine dynamically the best available offloading service within its tier, the tier controller requires from participating offloading services further parameters. First, it requires from an offloading service the non-varying parameters of the processing power and monetary cost retrieved via the service monitor. Second, it requires from the utilized links to offloading services the non-varying parameters of the monetary cost retrieved via the network interface as well as the time-varying parameters of the up and down bandwidth and latency retrieved via the network monitor. Possessing the required parameters to minimize Equation 8.1, the tier controller sends a resource offer for the best available offloading service within its tier via the tier connector to the service connector on the offloading client that sent the offload request.

Receiving an input execution state and the Java bytecode instructions for an application part from the tier connector – because the tier service sent a resource offer to an offloading client, the tier controller forwards both via the service connector to the corresponding offloading service. During the execution of an application part on an offloading service, the tier controller monitors via the service monitor the execution, to react, for instance, on crash failures of the offloading service.

Receiving from the service connector an output execution state from an offloading service after the execution of an application part, the tier controller forwards the output execution state to the tier connector.

### 8.5. Evaluation

This section evaluates the overhead and benefit for the operational system of the bubbling distribution in real-world environments compared to related approaches from the
8.5. Evaluation

(a) A smart phone (\(S\)) and the fog (\(F\)) communicate via a Wi-Fi link (\(w\)) with each other.

(b) A smart phone (\(S\)) and the cloud (\(C\)) communicate either via a Wi-Fi link (\(w\)) or a cellular link (\(c\)) with each other.

(c) A smart wearable (\(D\)) and a smart phone (\(S\)) communicate via a Bluetooth link (\(b\)) with each other, the smart phone (\(S\)) and a laptop (\(L\)) via a Wi-Fi link (\(w\)), the laptop (\(L\)) and the fog (\(F\)) via a tethered link (\(t\)), and the fog (\(F\)) and the cloud (\(C\)) via a tethered link (\(t\)).

Figure 8.4.: Overview of the different environments evaluated for the bubbling distribution, where (a) and (b) shows the environments with two tiers and (c) the environment with multiple tiers.

literature. In detail, the evaluation considers two different environments, namely an environment with two tiers and an environment with multiple tiers. To this end, we extended the implementation of the AOSP prototype for the basic distribution (cf. Subsection 4.6.3) to provide the functionality required on the offloading client for the bubbling distribution. Moreover, we extended the implementation of the OpenJDK prototype for the basic distribution (cf. Subsection 4.6.2) to provide the functionality required on the offloading service for the bubbling distribution. Please note that through the execution of an adapted OpenJDK JRE instead of a complete Android OS on the hardware platform of an offloading service, the bubbling distribution keeps the overhead introduced on participating resources very low. The extension to the AOSP prototype as well as to the OpenJDK prototype comprises changes to the implementation of the JVM and the Java core class libraries. Last, we implemented the functionality required for a tier service (cf. Subsection 8.4.4) in a Java application, executed on an unaltered OpenJDK JRE. The following subsections describe the evaluation setup (cf. Subsection 8.5.1) and the evaluation results (cf. Subsection 8.5.2).

8.5.1. Setup

The evaluation setup for the bubbling distribution extends the evaluation setup for the basic distribution on the AOSP described in Subsection 4.7.1 as follows:

Figure 8.4 shows the environments with two tiers (cf. Figure 8.4a and Figure 8.4b) and the environment with multiple tiers (cf. Figure 8.4c) evaluated for the bubbling distribution. The environments with two tiers consist of a smart phone (\(S\)) that dis-
attributes code to a remote tier, where the remote tier is either a fog distribution tier ($F$) or a cloud distribution tier ($C$). Both tiers differ in the performance of the participating resources. The smart phone and the remote tier communicate either via a Wi-Fi connection ($w$) or a cellular connection ($c$) with each other, differing in the latency as well as the bandwidth of the link. In total, the environments with two tiers result in $S_w F$, $S_w C$, and $S_c C$. The environment with multiple tiers consists of a smart phone ($S$), a device distribution tier ($D$), a local distribution tier ($L$), a fog distribution tier ($F$), and a cloud distribution tier ($C$) (cf. Figure 8.4c). The smart phone communicates via a Bluetooth connection ($b$) with the device distribution tier and via a Wi-Fi connection ($w$) with the local distribution tier. The local distribution tier communicates via a tethered connection ($t$) with the fog distribution tier that again communicates via a tethered connection ($t$) with the cloud distribution tier. Thus, the environment with multiple tiers results in $D_b S_w L_t F_t C$.

The device distribution tier ($D$) consists of the netbook Dell Inspiron Mini 10v (cf. Section C.2). It runs a tier service and an offloading service on its resources, possessing a benchmark value of $1.305$, and raises no monetary cost. The smart phone ($S$) is the Samsung Galaxy Nexus (cf. Section C.1). It runs an offloading client on its resources with a benchmark value of $3.106$. The local distribution tier ($L$) consists of the laptop Lenovo ThinkPad T61 (cf. Section C.3). It runs a tier service and an offloading service on its resources, possessing a benchmark value of $0.989$, and also raises no monetary cost. The fog distribution tier ($F$) consists of the desktop HP Compaq 8200 Elite (cf. Section C.4). It runs a tier service and three offloading services on its resources, possessing a benchmark value of $0.434$, and raises half the prices of the cloud distribution tier ($C$). Last, the cloud distribution tier ($C$) consists of the server instance t2.micro from the AWS EC2 (cf. Section C.5). It runs a tier service and an offloading service on its resources, possessing a benchmark value of $0.335$, and raises the prices$^2$ from the AWS EC2.

The smart phone ($S$) establishes the Bluetooth link ($b$) to the netbook ($D$) based on the Bluetooth version 2.1, having an average latency and bandwidth of $7.751\text{ms}$ and $235\text{kB/s}$. The Linksys WRT54GL wireless router (cf. Section C.7) establishes a wireless Access Point (AP) based on IEEE 802.11g connecting the smart phone ($S$) either with the laptop ($L$), the desktop computer ($F$), or the cloud ($C$) with an average latency and bandwidth of $3.681\text{ms}$ and $752\text{kB/s}$, $3.712\text{ms}$ and $2.7\text{MB/s}$, or $8.562\text{ms}$ and $2.6\text{MB/s}$, respectively. Moreover, the smart phone ($S$) communicates via a 3.5G mobile communication network with the cloud ($C$) having an average latency and bandwidth of $43.964\text{ms}$ and $200\text{kB/s}$. The LevelOne GSW-0809 Gigabit Ethernet

---

$^2$Prices of the AWS for the utilization of instances from its EC2: [aws.amazon.com/de/ec2/pricing](http://aws.amazon.com/de/ec2/pricing)
8.5. Evaluation

Switch (cf. Section C.8) connects the laptop (L) and the desktop computer (F) with each other, where the average latency and bandwidth of a link between the smartphone (S) and the desktop computer (F) is 4.936 ms and 2.1 MB/s. Last, a 10 Gigabit Ethernet link\(^3\) connects the desktop computer (F) and the cloud (C) with each other, where the average latency and bandwidth of a link between the smartphone (S) and the cloud (C) is 7.614 ms and 2.0 MB/s.

The evaluation compares the bubbling distribution with two approaches from the literature: An approach for a distribution regarding two tiers – named Two-tier Distribution – and an approach for a distribution regarding multiple tiers – named Multi-tier Distribution. The two-tier distribution distributes application parts from the smartphone to resources in the vicinity (fog distribution tier) or in a far-away cloud (cloud distribution tier) like MAUI [CBC+10]. To this end, it utilizes an offload-enabled Android OS for code offloading in environments with two tiers. The multi-tier distribution distributes code to resources organized into multiple tiers like Cloudlets [SBCD09]. To this end, it utilizes an offload-enabled Android OS for code offloading in environments with multiple tiers. We also compared the two-tier distribution, the multi-tier distribution, and the bubbling distribution with a local execution of the application. A local execution serves as a baseline, where the smartphone executes everything on its local resources in airplane mode. To this end, it utilizes an unaltered Android OS.

8.5.2. Results

The following subsection presents the evaluation results. First, it describes the overhead and benefit for a distribution in the environments with two tiers and for a distribution in the environment with multiple tiers. Afterwards, it presents the overhead introduced for securing a remote execution on an a priori unknown resource.

Environments with two tiers

The evaluation result of a local execution, the two-tier distribution, and the bubbling distribution are categorized into the overhead and benefit of a distribution.

Overhead of a Distribution: Figure 8.5 shows the execution time, Figure 8.6 the energy consumption, and Figure 8.7 the monetary cost of the local execution, the two-

\(^3\)The University of Stuttgart is part of the network “Baden-Württembergs extended LAN (BelWü)” that connects scientific organizations in Baden-Württemberg with low-latency and high-bandwidth links (https://www.belwue.de/netz/topologie/aktuell.html). The BelWü network peers with DE-CIX Frankfurt (https://www.de-cix.net/de/locations/germany/frankfurt), where the data center of the server instance from the Amazon Web Service is located in Frankfurt.
8. Code Offloading in Environments with Multiple Tiers

![Bar Chart]

Figure 8.5.: Execution time of a local execution, the two-tier distribution, and the bubbling distribution evaluated in the environments with two tiers.

tier distribution, and the bubbling distribution evaluated in the environments SwF, SwC, ScC, and ScC. For the first four group of bars in Figure 8.5 and Figure 8.6, the offloading service on the remote side does not possess a beneficial resource for a remote execution, indicating the overhead introduced by each approach. Please note that Table 8.1 outlines each value for the execution time, energy consumption, and monetary cost of the approaches evaluated in the environments with two tiers.

To make the decision of distributing a Java method that is offloadable, the two-tier distribution probes periodically the link to and monitors the state of the remote tier like in MAUI from Cuervo et al. [CBC+10]. Due to the missing beneficial resource in the remote tier, the two-tier distribution decides each time to execute the offloadable Java method locally on the smart phone. Compared to a local execution of the mobile application on the smart phone, the periodic link probing and state monitoring increase the execution time as well as the energy consumption of the two-tier distribution. For a latency of the Wi-Fi link between the smart phone and the fog distribution tier of 6.669 ms (SwF), the two-tier distribution takes 6.380 s longer and consumes 17.017 J more energy than a local execution. In case of a distant tier with a bad quality of the cellular link between the smart phone and the cloud distribution tier (cf. ScC in Figure 8.5 and Figure 8.6), the overhead of the two-tier distribution increases heavily with a latency of the cellular link of 294.197 ms. In detail, it takes 10.729 s longer and consumes 56.636 J more energy compared to a local execution.

In contrast to the periodic link probing and state monitoring, the bubbling distribution only submits the downgraded offload requests into the infrastructure and continues the local execution on the smart phone. As the gathering of a downgraded
8.5. Evaluation

Figure 8.6.: Energy consumption of a local execution, the two-tier distribution, and the bubbling distribution evaluated in the environments with two tiers.

offload request interrupts a pure local execution, the evaluation results of the bubbling distribution are worse than for a local execution. For the environment $SwF$, a local execution and the bubbling distribution have in average a similar power consumption of 2.5 W. However, the bubbling distribution finishes the local execution of the mobile application on the smart phone later at 72.139 s (cf. Figure 8.5) and thus, consumes more energy (cf. Figure 8.6). Compared to the two-tier distribution, the bubbling distribution performs better with a shorter execution time of 3.094 s and a lower energy consumption of 15.493 J. Distributing the Java methods that are offloadable to a distant cloud with a bad quality of the link (cf. $ScC$ in Figure 8.5 and Figure 8.6), the execution time and the energy consumption for the bubbling distribution only minor increase by 1.460 s and 10.495 J, respectively. As the high latency to the cloud distribution tier does not influence the bubbling distribution much, it performs better than the two-tier distribution. In detail, it reduces the execution time by 5.983 s and the energy consumption by 44.671 J.

Summarizing, the two-tier distribution as well as the bubbling distribution introduce some overhead in the environments with two tiers, increasing the execution time and the energy consumption compared to a local execution. Due to the distribution without a beneficial resource on the remote side, both approaches cause no monetary cost at all. In detail, the overhead introduced by the two-tier distribution is higher than for the bubbling distribution, because the two-tier distribution probes periodically the link to and monitors the state of the remote tier. The decoupled submission of downgraded offload requests keeps the overhead introduced by the bubbling distribution low, especially in case of a bad quality of the link between the smart phone and the remote tier.
8. Code Offloading in Environments with Multiple Tiers

**Figure 8.7.** Monetary cost of a local execution, the two-tier distribution, the multi-tier distribution, and the bubbling distribution evaluated in the environments with two tiers and the environment with multiple tiers, respectively.

**Benefit of a Distribution:** Figure 8.5 shows the execution time, Figure 8.6 the energy consumption, and Figure 8.7 the monetary cost of a local execution, the two-tier distribution, and the bubbling distribution evaluated in the environments $SwF$, $SwC$, $ScC$, and $ScC$. For the last four group of bars in Figure 8.5 and Figure 8.6 and the first four group of bars in Figure 8.7, the offloading service on the remote side possesses a beneficial resource for a remote execution, indicating the benefit gained.

The environment $SwF$ corresponds to a perfect environment for a distribution of the Java methods that are offloadable, providing a sufficient link with a latency of 4.679 ms and a bandwidth of 1.9 MB/s between the smart phone and the remote tier with the beneficial resource. As a result, the two-tier distribution distributes all three Java methods that are offloadable, where the execution time of the best chess move engine decreases to 4.217 s, of the human face recognition engine to 7.456 s, and of the text-to-voice engine to 1.589 s. In total, it takes 14.900 s and consumes 44.518 J with a monetary cost of 3.653 E-6 $, outperforming significantly a local execution on the smart phone. Changing the environment to a distant tier with a cellular link (cf. $ScC$ in Figure 8.5 and Figure 8.6), the bandwidth decreases to 185 kB/s. Thus, the two-tier distribution does not distribute the human face recognition engine due to its huge size of the input execution state. The local execution of the human face recognition engine on the smart phone increases the execution time and the energy consumption of the two-tier distribution, however, still resulting in a shorter execution time of 33.860 s.
8.5. Evaluation

Figure 8.8.: The power consumption of the bubbling distribution evaluated in the environment $ScC$. The bubbling distribution recognizes the too bad bandwidth of the link between the smart phone and the remote tier after the transmission of 250kB to the cloud distribution tier.

and a lower energy consumption of 84.882 J compared to a total local execution of the application on the smart phone. Increasing simultaneously the latency of the cellular link to 253.772 ms (cf. $ScC$ in Figure 8.5 and Figure 8.6), the two-tier distribution only distributes the best chess move engine to the remote tier. Despite the local execution of the two Java methods that are offloadable, it still performs better than a total local execution of the application on the smart phone. In detail, the two-tier distribution results in an execution time of 58.831 s and an energy consumption of 155.566 J with a monetary cost of 8.088 E-6 $.

For the environment $SwF$, the bubbling distribution results in a similar execution time, energy consumption, and monetary cost as the two-tier distribution. Due to the huge size of the input execution state, the bubbling distribution also does not distribute the human face recognition engine for the environment $ScC$. It recognizes the too bad bandwidth of the link between the smart phone and the remote tier by sending the first 250kB to the cloud distribution tier (cf. the peak at 14s in Figure 8.8). As the bubbling distribution executes in parallel the human face recognition engine on the smart phone, however, the evaluation results of the bubbling distribution are better than of the two-tier distribution. The bubbling distribution reduces the execution time by 8.333 s and the energy consumption by 2.282 J with a similar monetary cost. Increasing simultaneously the latency of the cellular link to the cloud distribution tier (cf. $ScC$ in Figure 8.5 and Figure 8.6), the bubbling distribution also does not distribute the text-to-voice engine like the two-tier distribution. It executes the
text-to-voice engine locally based on the bandwidth and latency measured during the
evaluation, not sending any bytes for the input execution state to the remote tier unlike
for the human face recognition engine. Thus, the evaluation results of the bubbling
distribution evaluated in $S_C$ are better than of the two-tier distribution, reducing the
execution time by 25.711s and the energy consumption by 53.097J. Compared to a
local execution, the bubbling distribution reduces the execution time by 35.733s and
the energy consumption by 70.690J with a monetary cost of $12.582 E-6$.

Summarizing, the benefits gained of the two-tier distribution and the bubbling dis-
bution for a distribution are high for the environments with two tiers possessing a
sufficient link to the remote tier with a beneficial resource. Due to the distribution
with a beneficial resource, both approaches reduce the execution time and the energy
consumption compared to a local execution, causing monetary cost for the remote exe-
cution on an offloading service. In detail, the benefit gained of the two-tier distribution
is lower than of the bubbling distribution, because the two-tier distribution probes pe-
riodically the link to and monitors the state of the remote tier. Especially in case of a
link with a high latency and a low bandwidth, the decoupled submission of downgraded
offload requests keeps the benefit of the bubbling distribution high.

Environment with Multiple Tiers

The evaluation result of a local execution, the multi-tier distribution, and the bubbling
distribution are categorized into the overhead and benefit of a distribution.

Overhead of a Distribution: Figure 8.5 shows the execution time, Figure 8.6 the en-
ergy consumption, and Figure 8.7 the monetary cost of a local execution, the multi-tier
distribution, and the bubbling distribution evaluated in the environments $D_{bSwLtFtC}$,
$D_{bSwLtFrC}$, and $D_{bSwLtFtC}$. For the first three group of bars in Figure 8.5 and
Figure 8.6, the offloading services at the multiple tiers do not possess a beneficial re-
source for a remote execution, indicating the introduced overhead. Please note that
Table 8.1 outlines each value for the execution time, energy consumption, and monetary
cost of the approaches evaluated in the environment with multiple tiers.

The multi-tier distribution also makes its decision for a distribution in the environ-
ment with multiple tiers by periodically probing the link to and monitoring the state
of the tiers. Due to the missing beneficial resources in each tier, the multi-tier dis-
bution executes all of the Java methods that are offloadable on the smart phone.
Thus, the periodic probing and monitoring together with the local execution of the
mobile application increases the execution time and the energy consumption of the
8.5. Evaluation

Figure 8.9.: Execution time of a local execution, the multi-tier distribution, and the bubbling distribution evaluated in the environment with multiple tiers.

The multi-tier distribution compared to a local execution without code offloading. In detail, it takes 15.932 s longer and consumes 28.141 J more energy (cf. DbSwLtFtC in Figure 8.5 and Figure 8.6). Increasing the latency of the tethered link between the fog distribution tier and the cloud distribution tier to 268.189 ms, the evaluation results for the multi-tier distribution are getting worse. In detail, it takes 22.362 s longer and consumes 44.986 J more energy compared to a local execution without code offloading (cf. DbSwLtFtC in Figure 8.5 and Figure 8.6). Increasing simultaneously the latency of the tethered link between the local distribution tier and the fog distribution tier to 154.686 ms and between the fog distribution tier and the cloud distribution tier to 402.399 ms, it takes 32.937 s longer and consumes 58.891 J more energy compared to a local execution without code offloading (cf. DbSwLtFtC in Figure 8.5 and Figure 8.6).

Due to the gathering and submitting of the downgraded offload requests, the bubbling distribution results in worse results compared to a local execution without code offloading. In detail, it takes 4.835 s longer and consumes 1.360 J more energy for the environment DbSwLtFtC (cf. Figure 8.5 and Figure 8.6). However, both increases of the latency of the links do not affect the results of the bubbling distribution, keeping the execution time and the energy consumption constant (cf. DbSwLtFtC and DbSwLtFtC in Figure 8.5 and Figure 8.6). Compared to the execution time and the energy consumption of the multi-tier distribution, the bubbling distribution performs better for the environment with multiple tiers with and without a bad quality of a link. In detail, it reduces the execution time by 11.097 s, 17.403 s, and 27.711 s as well as the
8. Code Offloading in Environments with Multiple Tiers

Figure 8.10.: Energy consumption of a local execution, multi-tier distribution, and bubbling distribution evaluated in the environment with multiple tiers.

energy consumption by 29.501 J, 41.962 J, and 56.216 J (cf. Figure 8.5 and Figure 8.6).

Summarizing, the multi-tier distribution as well as the bubbling distribution introduce some overhead in the environment with multiple tiers, increasing the execution time and the energy consumption compared to a local execution without code offloading. Due to the distribution without a beneficial resource in the multiple tiers, both approaches cause no monetary cost at all. However, the overhead introduced for the multi-tier distribution is higher than for the bubbling distribution. The decoupled submission of downgraded offload requests also keeps the overhead introduced in the environment with multiple tiers for the bubbling distribution low.

Benefit of a Distribution: Figure 8.5 shows the execution time, Figure 8.6 the energy consumption, and Figure 8.7 the monetary cost of a local execution, the multi-tier distribution, and the bubbling distribution evaluated in the environments DbSwLtFtC, DbSwLtFtC, and DbSwLtFtC. For the last three group of bars in the figures, the offloading services at the multiple tiers possess beneficial resources for a remote execution, indicating the benefit gained by the approaches.

The environment DbSwLtFtC corresponds to an optimal environment for a distribution. Thus, the multi-tier distribution distributes all three Java methods that are offloadable to the cloud distribution tier. The distribution of the three Java methods that are offloadable decreases the execution time by 47.049 s and the energy consump-
8.5. Evaluation

tion by 115.694 J with a monetary cost of 8.587 E-6 $ compared to a local execution without code offloading (cf. Figure 8.5, Figure 8.6, and Figure 8.7). Increasing the latency of the tethered link between the fog distribution tier and the cloud distribution tier, the evaluation results are getting worse but still performing better than a local execution without code offloading (cf. \texttt{DbSwLtFtC} in Figure 8.5, Figure 8.6, and Figure 8.7). In detail, the multi-tier distribution distributes the best chess move engine to the cloud distribution tier, executes the human face recognition engine locally on the smart phone, and distributes the text-to-voice engine to the fog distribution tier. It results in a reduction of the execution time by 20.869 s and the energy consumption by 73.187 J compared to a local execution without code offloading. However, increasing both the latency of the tethered link between the local distribution tier and the fog distribution tier as well as between the fog distribution tier and the cloud distribution tier, the multi-tier distribution executes all three Java methods that are distributable locally on the smart phone (cf. \texttt{DbSwLtFtC} in Figure 8.5, Figure 8.6, and Figure 8.7). As a result, the evaluation results of the multi-tier distribution are worse than a local execution without code offloading with an increase of the execution time by 43.266 s and the energy consumption by 73.937 J.

For the environment \texttt{DbSwLtFtC}, the bubbling distribution distributes all three Java methods that are distributable to the fog distribution tier, reducing the execution time by 8.418 s and the energy consumption by 10.971 J compared to the multi-tier distribution. Increasing the latency of the tethered link between the local distribution tier and the fog distribution tier as well as between the fog distribution tier and the cloud distribution tier, the evaluation results for the bubbling distribution remain constant (cf. \texttt{DbSwLtFtC} and \texttt{DbSwLtFtC} in Figure 8.5, Figure 8.6, and Figure 8.7). Due to the high link latency, the bubbling distribution distributes the text-to-voice engine to the local distribution tier. In total, it performs better than the multi-tier distribution and a local execution without code offloading for the environment with multiple tiers.

Summarizing, the benefits of the multi-tier distribution and the bubbling distribution for a distribution are high for the environment with multiple tiers possessing suitable links to the multiple tiers with beneficial resources. Due to the distribution with beneficial resources, both approaches reduce the execution time and the energy consumption compared to a local execution without code offloading, causing monetary cost for the remote execution on the offloading services. In detail, the benefit gained of the bubbling distribution is higher than of the multi-tier distribution due to the decoupled submission of downgraded offload requests instead of the periodic probing of the links to and monitoring of the state from the multiple tiers. Especially in case of links with a high latency, the bubbling distribution keeps the related benefit high.
Table 8.2: Overview of the execution time, the energy consumption, and the monetary cost for a local execution (Lcl), the two-tier distribution (TtD), and the bubbling distribution (BlD) evaluated in the environment with two tiers with and without beneficial resources.

<table>
<thead>
<tr>
<th>Without Beneficial Resources:</th>
<th>With Beneficial Resources:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Energy (in Joule)</strong></td>
<td><strong>Energy (in Joule)</strong></td>
</tr>
<tr>
<td>173.159</td>
<td>173.159</td>
</tr>
<tr>
<td>201.300</td>
<td>57.465</td>
</tr>
<tr>
<td>171.799</td>
<td>46.494</td>
</tr>
<tr>
<td>218.145</td>
<td>99.972</td>
</tr>
<tr>
<td>176.183</td>
<td>48.514</td>
</tr>
<tr>
<td>232.050</td>
<td>247.096</td>
</tr>
<tr>
<td>175.834</td>
<td>57.014</td>
</tr>
<tr>
<td></td>
<td>187.065</td>
</tr>
<tr>
<td></td>
<td>229.795</td>
</tr>
<tr>
<td><strong>Time (in Seconds)</strong></td>
<td><strong>Time (in Seconds)</strong></td>
</tr>
<tr>
<td>68.853</td>
<td>68.853</td>
</tr>
<tr>
<td>84.785</td>
<td>21.804</td>
</tr>
<tr>
<td>73.688</td>
<td>13.386</td>
</tr>
<tr>
<td>91.215</td>
<td>47.984</td>
</tr>
<tr>
<td>73.812</td>
<td>12.651</td>
</tr>
<tr>
<td>101.790</td>
<td>112.119</td>
</tr>
<tr>
<td>74.079</td>
<td>18.706</td>
</tr>
<tr>
<td></td>
<td>173.159</td>
</tr>
<tr>
<td><strong>Cost (in 10^-6 $)</strong></td>
<td><strong>Cost (in 10^-6 $)</strong></td>
</tr>
<tr>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>0.000</td>
<td>8.587</td>
</tr>
<tr>
<td>0.000</td>
<td>3.559</td>
</tr>
<tr>
<td>0.000</td>
<td>14.736</td>
</tr>
<tr>
<td>0.000</td>
<td>3.617</td>
</tr>
<tr>
<td>0.000</td>
<td>2.882</td>
</tr>
<tr>
<td></td>
<td>173.159</td>
</tr>
<tr>
<td><strong>I°l</strong></td>
<td><strong>I°l</strong></td>
</tr>
<tr>
<td></td>
<td>173.159</td>
</tr>
<tr>
<td></td>
<td>173.159</td>
</tr>
<tr>
<td></td>
<td>173.159</td>
</tr>
<tr>
<td></td>
<td>173.159</td>
</tr>
</tbody>
</table>

Table 8.3: Overview of the execution time, the energy consumption, and the monetary cost for a local execution (Lcl), the multi-tier distribution (MtD), and the bubbling distribution (BlD) evaluated in the environment with multiple tiers with and without beneficial resources.
8.5. Evaluation

<table>
<thead>
<tr>
<th>Application Part</th>
<th>Distribution Tier</th>
<th>Offloading Service Time</th>
<th>Time Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Basic</td>
<td>Bubbling</td>
<td></td>
</tr>
<tr>
<td>Best Chess Move Engine</td>
<td>D</td>
<td>18.386s</td>
<td>18.446s</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>5.014s</td>
<td>5.028s</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>1.800s</td>
<td>1.801s</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>2.461s</td>
<td>2.463s</td>
</tr>
<tr>
<td>Human Face Recognition Engine</td>
<td>D</td>
<td>2.579s</td>
<td>2.627s</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>0.582s</td>
<td>0.591s</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>0.248s</td>
<td>0.249s</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>0.616s</td>
<td>0.620s</td>
</tr>
<tr>
<td>Text-to-Voice Engine</td>
<td>D</td>
<td>5.228s</td>
<td>5.659s</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>1.216s</td>
<td>1.276s</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>0.534s</td>
<td>0.538s</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>0.821s</td>
<td>0.834s</td>
</tr>
</tbody>
</table>

Table 8.3.: Overview of execution time on an offloading service for the basic distribution and the bubbling distribution for each resource-intensive application part. The distribution tiers that executes the application parts are the device distribution tier (D), the local distribution tier (L), the fog distribution tier (F), and the cloud distribution tier (C).

8.5.3. Securing Overhead

Table 8.3 shows the time and transmission overhead introduced for securing a remote execution of application parts on an a priori unknown resource. To this end, participating resources in the device distribution tier (D), the local distribution tier (L), the fog distribution tier (F), and the cloud distribution tier (C) execute each resource-intensive application part – namely the best chess move engine, the human face recognition engine, and the text-to-voice engine – 10 times on an unaltered offloading service from the basic distribution (cf. Subsection 4.5.3) and 10 times on an offloading service for the bubbling distribution (cf. Subsection 8.4.3).

For the best chess move engine, the time overhead introduced for the execution of the additional Java bytecode instruction `offload_branch` ranges between 0.06% and 0.35% for the participating resources in D, L, F, and C (cf. Table 8.3). In detail, the execution time for the best chess move engine in F differs only by 1ms and in C by 2ms, being very similar. The execution time in the device distribution tier (D) differs by 60ms, also resulting in a very low time overhead. Regarding the time overhead, the tier with the most powerful resources (F) induces the lowest overhead and the tier with the poorest resources (D) has the highest overhead. The same characteristic applies for the human face recognition engine and the text-to-voice engine. The time overhead for
8. Code Offloading in Environments with Multiple Tiers

D, L, F, and C ranges between 0.40% and 1.86% for the human face recognition engine and between 0.75% (F) and 8.24% (D) for the text-to-voice engine (cf. Table 8.3). However, the absolute values for the additional time from the text-to-voice engine ranges between 0.004 ms (F) and 0.431 ms (D), still being acceptable with regard to the local execution time of 18.624 ms on the smart phone (cf. Subsection 4.7.2).

Beside the time overhead for the additional execution of the Java bytecode instruction `offload_branch`, the offloading service for the bubbling distribution also induces a transmission overhead. In detail, it is independent from the actual resource at the distribution tier (cf. D, L, F, or C) that executes a Java method as well as from the distributed Java method itself. Regarding the additional transmission, an offloading service for the bubbling distribution transfers additionally to an offloading client the calculated hash value and the integer values of i and x compared to an offloading service for the basic distribution. In total, the additional number of bytes transferred is 64 byte for the hash value plus 8 byte for the two integer values, resulting in total in a very low transmission overhead of 72 byte.

8.6. Summary

To provide a code offloading in environments with multiple tiers through code bubbling, the *bubbling distribution* described in Section 8.1 extends the basic distribution. The bubbling distribution offloads computation not only between a mobile device (first tier) and a remote resource (second tier) but also between multiple tiers with highly distributed heterogeneous resources. To this end, the bubbling distribution utilizes a *system overview* described in Section 8.2 of multiple tiers, adjusting accordingly the system model, problem statement, and system components of the basic distribution. For the bubbling distribution, an offloading client can distribute an application part that is offloadable among heterogeneous resources on many tiers (e.g., wearable device ↔ smart phone ↔ edge servers ↔ cloud servers, etc.). Thus, the decision making for code offloading in environments with multiple tiers is more complex. Keeping code offloading in environments with multiple tiers also applicable, the bubbling distribution introduces the concept of code bubbling – *offloading timeline* described in Section 7.3. Code bubbling enables each tier to make autonomous decisions for code offloading without the need to gather a global view onto all resources of all tiers. As a result, the usage of code bubbling potentially scales to an arbitrary number of tiers since the decision making for code offloading only involves neighboring tiers. The *offloading framework* described in Section 8.4 provides the functionality required for the bubbling distribution, extending the API of the Java programming language. Moreover,
8.6. Summary

It extends the offloading client and the offloading service with the functionality for the bubbling distribution, and adds a tier service for managing participating resources within a tier. To evaluate the overhead introduced and benefit gained by the bubbling distribution compared to the basic distribution, the *evaluation* described in Section 8.5 of a prototype based on the Android OS and the OpenJDK JRE analyzes multiple approaches in different environments. The evaluation results show that a distribution with code bubbling keeps the introduced overhead low and gained benefit high due to the submission of downgraded offload requests and the local continuation of the application execution. Especially, in case of a network link with a bad quality, the bubbling distribution significantly performs better than related approaches from the literature.
Chapter 9

Related Work

This chapter presents the work related to the contributions previously described in this dissertation. The related work comprises computation offloading in the landscape of Mobile Cloud Computing (MCC) that covers both code offloading and cyber foraging. Both augment a (resource-poor) mobile device with (powerful) remote resources, where cyber foraging locates a remote resource in the vicinity like an underutilized desktop computer and code offloading locates it at a distant data center at the cloud. To this end, Section 9.1 discusses the work related to an efficient code offloading with annotations described in Chapter 4. It provides the basic distribution of application parts between a mobile device and a remote resource, keeping the efficiency of a distribution high and at the same time the burden on application developers and end users low. During a remote execution of application parts distributed to a remote resource, crash failures like a disconnection might occur. To this end, Section 9.2 discusses the work related to a robust code offloading through safe-point’ing described in Chapter 5. It provides the preemptable distribution of application parts improving the basic distribution to locally reuse intermediate states – so-called safe-points – from the remote side, not abandoning partial computations achieved remotely so far. Section 9.3 discusses the work related to a deadline-aware code offloading with predictive safe-point’ing described in Chapter 6 that provides the predictive distribution. It improves the preemptable distribution with an adaptive algorithm based on prediction models to dynamically adapt the point in times for safe-point’ing due to dynamic conditions of the communication network. Next, Section 9.4 discusses the work related to an optimized code offloading through cooperative caching described in Chapter 7 that provides the caching-aware distribution of application parts. It increases the efficiency of the basic distribution with a cache on the remote side that stores the result of application parts previously executed, avoiding a repeated execution. As highly distributed heterogeneous resources surround nowadays a mobile device in the landscape of MCC, Section 9.5 discusses the work related to a code offloading in environments with multiple tiers described in Chapter 8. It provides the bubbling distribution of application parts to remote resources in multiple tiers, shifting the decision making for code of-
9. Related Work

floading from a (resource-poor) mobile device into the infrastructure, not requiring to
gather a global view onto all resources of all tiers. Last, Section 9.6 summarizes the
main facts for the related work of this dissertation presented in this chapter.

9.1. Efficient Code Offloading with Annotations

In the literature, several approaches show the capabilities of a distribution of applica-
tion parts from a (resource-constrained) mobile device like a smart phone to a (power-
ful) remote resource in the infrastructure like a server instance at the cloud.

Cuervo et al. [CBC+10] propose an offloading approach called MAUI that improves
the energy efficiency of a mobile device. MAUI distributes application parts at the
granularity of methods from a mobile device to a remote server located in a data cen-
ter at the cloud. MAUI identifies application parts that are offloadable based on an
annotation from an application developer annotated at development-time. To this end,
they formulate an Integer Linear Program (ILP) that minimizes the energy consump-
tion of a mobile application on the mobile device subjected to a maximum time for
an execution of an application. The ILP considers the network parameters like the
bandwidth and latency of the link between the mobile device and the server.

CloneCloud from Chun et al. [CIM+11] also improves the energy efficiency of a
mobile device by distributing application parts from a mobile device to a remote server
at the cloud. CloneCloud distributes application parts at the granularity of threads,
where a so-called device clone for each mobile device runs at the cloud. A main goal
for the design of CloneCloud is a transparent distribution of application parts for an
application developer by automatically transforming and partitioning the code of an
application to enable a distributed execution. To automatically distribute threads from
a mobile device to its clone, it partitions applications based on a static analysis of the
application at development-time together with a profiling of the application at run-
time. Thereupon, CloneCloud determines the execution side for an application part
by formulating an optimization problem that minimizes the energy consumption of the
mobile device subjected to a maximal time for an execution of an application.

Giurgiu et al. [GRA12] propose an approach that distributes application parts at the
granularity of modules. To this end, the approach partitions applications into modules
based on the technology of remote OSGi (Open Services Gateway initiative) [RAR07].
After the partitioning of an application, the approach distributes dynamically the part-
tioned modules between a mobile device and a remote server based on profiling the
mobile device as well as the application.

Gitzenis et al. [GB09] propose an approach that considers the local management
for the power consumption of a mobile device for a distribution of application parts by adjusting, for instance, the processor speed. The approach focuses on the trade-off between the execution time of an application and the energy saving on the mobile device, modeling the decision problem of a distribution as a controlled Markov chain. The solution of the problem is a policy that determines the execution side as well as the power consumption of the mobile device.

ThinkAir from Kosta et al. [KAH+12] also distributes application parts from a mobile device to a remote server at the cloud reducing the execution time and energy consumption on the mobile device for an execution of an application. Beside the execution time and energy consumption, ThinkAir explicitly includes the monetary cost for a remote execution at the cloud. To this end, it dynamically scales the computational power at the cloud to achieve an optimal performance, allocating on-demand server instances at the cloud and mapping the application parts to the allocated server instances. Furthermore, it allows a parallel execution of application parts on multiple server instances at the cloud.

The middleware proposed from Ferber et al. [FRTH12] augments the execution of compute-intensive applications on a resource-constrained mobile device with cloud resources. It provisions on-demand the utilization of server instances at the cloud and thus, considers the costs arose for a remote execution by accounting end users.

Shi et al. [SHP+14] present the distribution system COSMOS that minimizes the execution time, energy consumption, and monetary cost of an application on a mobile device. COSMOS also allocates resources at the cloud to schedule efficiently the remote execution of distributed application parts. It offers the distribution of computation as a service mapping dynamically application parts from mobile devices to compute resources in a commercial data center at the cloud. As a result, its optimization problem for a distribution additionally minimizes the leasing costs of cloud resources while handling variable network connectivity.

Tasklet from Paluska et al. [PPS+12, SEV+16] provides a lightweight abstraction for best-effort computation at the granularity of threads to migrate computation, for instance, from local resources of a mobile device to resources in the cloud. In detail, a Tasklet is a fine-grained and self-contained computation unit created dynamically at run-time that includes everything for its processing like instructions, data, and state. The underlying representation of instructions and data are chunks, building a graph of linked chunks as a Tasklet. A Tasklet container consists of a virtual machine to execute Tasklets, running on any suitable hardware connected to the Internet.

Summarizing, multiple approaches proposed in the literature show the benefits of computation offloading increasing significantly the energy efficiency on a (resource-
9. Related Work

Constrained mobile device and the execution speed for a (resource-intensive) application. However, MAUI from Cuervo et al. [BC+10] and CloneCloud from Chun et al. [CM+11] focuses on the improvement of the energy efficiency of a mobile device and the execution speed of an application. They do not regard monetary cost for a remote execution of application parts distributed to a server instance, hosted, for instance, in a commercial data center at the cloud. In detail, MAUI is similar to the basic distribution presented in Chapter 4, where an application developer annotates methods that are offloadable. However, MAUI does not instrument bytecodes for the efficient identification of methods that are offloadable. ThinkAir from Kosta et al. [KAH+12] and COSMOS from Shi et al. [SHP+14] focuses on improvements on the remote side like resource allocation and execution parallelism, executing on a server instance at the cloud a virtual machine of a complete system from a smartphone. Tasklet from Paluska et al. [PPS+12] focuses on the issue of best-effort computation, utilizing at the system core an own virtual machine for thread migration implemented in a new programming language. In contrast to related approaches proposed in the literature, the basic distribution presented in Chapter 4 minimizes the execution time, energy consumption, and monetary cost for a computation offloading of application parts between a mobile device and a remote resource. It utilizes an annotation-based instrumentation of Java bytecode instructions that keeps both the overhead introduced to a mobile device and the burden on application developers and end users low.

9.2. Robust Code Offloading through Safe-point’ing

The approaches related to the basic distribution of application parts described in Section 9.1 show the benefits for a distribution of application parts between a mobile device and a remote server. The execution of application parts on a remote server on behalf of a mobile device reduces significantly the energy consumption and execution time on the mobile device. Due to the utilization of a remote resource, the remote execution causes monetary cost for an end user. Although these approaches increase the efficiency on a mobile device, they assume a permanent link between a mobile device and a remote resource as well as a permanent availability of a remote resource. They do not consider the adverse effects from the occurrence of failures like a link disconnection, especially as the occurrence of failures can result in a higher execution time and/or energy consumption than a local execution on a mobile device.

Kwon et al. [KT12] propose a basic mechanism handling the occurrence of failures. The approach creates a single safe-point before the distribution of application parts that are offloadable, annotated by an application developer. In case of an occurrence
of a failure during a remote execution of an application part, it starts a complete re-
execution of the application part on the mobile device based on the single safe-point.

The ENDA system from Li et al. [LBLX13] also considers the occurrence of failures.
ENDA distributes application parts that are offloadable from a mobile device either
to small servers called cloudlets via a Wi-Fi connection or to server instances at the
cloud via a 3G mobile communication network. Based on historic traces of a mobile
device, it selects Wi-Fi access points along the predicted route that connect the mobile
device to the cloudlets. In case of the unavailability of a cloudlet, it uses the 3G mobile
communication to distribute an application part to a server instances at the cloud.
Furthermore, it simply waits for a re-connection to a cloudlet or a server instance if a
failure occurs during a remote execution of an application part.

The COMET system from Gordon et al. [GJM+12] also focuses on the robustness
for a distribution of an application part – more precisely, a thread – between a mobile
device and a remote server. To handle the occurrence of failures from the network or
the server, COMET utilizes the techniques of Distributed Shared Memory (DSM) and
Virtual Machine Synchronization (VMS), already utilized for the offloading of threads.
Based on DSM and VMS, COMET synchronizes threads distributed between a mobile
device and a remote sever, where this fine-grained heap and stack synchronization
between a (resource-constrained) mobile device and a remote server induces a high
overhead of communication. For instance, COMET has to synchronize the complete
initial state at the start of an application, resulting in a typical size of 750-810kB.

Summarizing, multiple approaches proposed in the literature show techniques for
computation offloading that handles the occurrence of failures during a distribution of
an application part. However, the approach from Kwon et al. [KT12] starts in case of a
failure a complete re-execution on a mobile device of an application part. It abandons
the intermediate state achieved on the remote side so far resulting in a higher execution
time and energy consumption in case of a failure. ENDA from Li et al. [LBLX13] waits
in case of a failure for a re-connection to the remote resource that executes the applica-
tion part, also resulting in a higher execution time and energy consumption, especially,
for instance, for long lasting disconnections. COMET from Gordon et al. [GJM+12]
utilizes a fine-grained synchronization of threads distributed between a mobile device
and a remote resource, introducing a large communication overhead. In contrast to the
related approaches proposed in the literature, the preemptable distribution presented
in Chapter 5 utilizes incremental safe-point’ing, where a safe-point corresponds to the
intermediate state on the remote side. It re-uses locally the intermediate states of the
remote side to continue the remote execution locally, leading to a much smaller amount
of data that a remote resource transfers to a mobile device.
9. Related Work

9.3. Deadline-aware Code Offloading with Predictive Safe-point’ing

The approaches related to the preemptable distribution of application parts described in Section 9.2 handle the occurrence of failures during a remote execution of application parts distributed between a mobile device and a remote resource. In case of an occurrence of a failure, the approaches either re-executes a (resource-intensive) application part on a (resource-constrained) mobile device (cf. Kwon et al. [KT12]) or waits for a re-connection to the remote resource (cf. Li et al. [LBLX13]). Handling an occurrence of failures by re-executing application parts or waiting for a re-connection reduces the efficiency on a mobile device of a distribution in case of a failure. Furthermore, both strategies of re-executing and re-connecting might not be optimal with regard to energy efficiency and execution speed, giving no guarantees about an execution deadline for an application. The preemptable distribution with safe-point’ing presented in Chapter 5 re-uses locally partial results (intermediate states) from the remote side in case of a failure to continue a remote execution on the mobile device. To this end, the preemptable distribution creates safe-points – that captures the temporal execution state of an application part executed on a remote resource – and transmits these safe-points to the mobile device during a remote execution of an application part. After an occurrence of a failure, a mobile device continues the remote execution locally based on the safe-point received most recently. Summarizing, the preemptable distribution with safe-point’ing proposed from Berg et al. [BDR14a] uses a non-predictive schedule to determine the points in time to create and transmits safe-points. The strategy that determines the points in time for safe-point’ing does not always perform optimally under dynamic conditions of the network. In contrast to the related approaches proposed in the literature, the predictive distribution presented in Chapter 6 utilizes an adaptive schedule based on prediction models. It dynamically adapts the point in times for a creation and transmission of safe-points due to dynamic conditions of the communication network, increasing the efficiency of code offloading.

9.4. Optimized Code Offloading through Cooperative Caching

The approaches related to the basic distribution of application parts described in Section 9.1 show the benefits related to the basic distribution augmenting a (resource-constrained) mobile device with a remote resource. The execution of application parts
9.4. Optimized Code Offloading through Cooperative Caching

distributed to a remote resource causes monetary cost due to the utilization, for instance, of cloud resources (basic distribution). To this end, the caching-aware distribution improves the basic distribution with a cache. Today, many computing systems utilize different types of a cache for data and function as, for instance, nodes in a distributed system that utilize a cache for Domain Name System (DNS) to speed-up the lookup of a name resolution. In general, the basic idea behind the concept of caching is that a cache stores information of current requests, speeding up future requests in case of a cache hit. In case of a cache miss, a cache needs to obtain the information elsewhere, which is comparatively slower.

WhereStore from Stuedi et al. [SMT10] is a data cache for mobile devices that replicates cloud data on the limited memory of a mobile device based on the device’s history of location. WhereStore distributes the data between mobile devices and the cloud based on filtered replication together with the device’s history of location. As a result, it decreases the access times for the data and improves availability of the data, especially in case of a disconnection between a mobile device and the cloud. The main idea of this location-aware caching between mobile devices and the cloud is that a mobile device typically accesses only specific data at certain places (e.g., restaurant recommendations at downtown).

Michie [Mic68] proposes a run-time optimization for a local execution related to function calls, avoiding the overhead for a redundant execution of a function locally called Memoization. To this end, a local cache for functions stores the arguments and the corresponding result from a previous call of a function. In case of the local cache for functions has a corresponding result (cache hit), it returns immediately a call of a function with the corresponding result. This avoids the local execution of the function’s body once again, reducing the execution time of a function.

ADEOM from Jiang et al. [JHLX14] consider a cache on the remote side for a distribution of application parts between a mobile device and a remote server. The cache stores an input execution state of an application part required for its execution on a remote resource. Through the integration of a cache, it reduces the time for a transmission of an input execution state for an application part distributed between a mobile device and a remote server. In case of a repeated execution on a remote resource of an application part previously distributed, a mobile device does not have to sent the input execution state already stored in the cache on the remote side. Thus, ADEOM decreases the memory size of migrated data and accelerates the transmission time.

Summarizing, multiple approaches proposed in the literature show the benefits related to the utilization of caching. WhereStore from Stuedi et al. [SMT10] provides a location-aware caching for information related to the location of a mobile device,
9. Related Work

replicating cloud data on the mobile device. “Memoization” from Michie [Mic68] is a local cache for functions, avoiding a redundant execution of a function in case of a cache hit. ADEOM from Jiang et al. [JHLX14] provides for computation offloading a cache on the remote side storing input execution states for application parts, avoiding a repeated transmission of information required for an execution on a remote resource. In contrast to the related approaches proposed in the literature, the caching-aware distribution presented in Chapter 7 utilizes for computation offloading a cache on the remote side. The cache stores output execution states to (further) speed up the time and energy required for a remote execution of an application part that is offloadable.

9.5. Code Offloading in Environments with Multiple Tiers

The approaches related to the basic distribution of application parts described in Section 9.1 show the benefits related to the basic distribution of application parts between a mobile device (first tier) and a remote resource (second tier). Although these approaches increase the energy efficiency on a mobile device and the execution speed for an application in environments with two tiers, they largely neglect a distribution of application parts in environments with multiple tiers, typically found in today’s landscape of MCC. Compared to an environment with two tiers, highly distributed heterogeneous resources from multiple tiers surround a mobile device, differing in performance characteristics and cost implications.

Scavenger from Kristensen [Kri10] distributes application parts in mobile, heterogeneous environments. Scavenger enables a resource-constrained mobile device to execute tasks – more precisely RPCs – from Python applications on underutilized, unknown devices in the vicinity called surrogates like a desktop computer. Scavenger selects the best available surrogate for a task with the help of its “strength”, determined from a CPU benchmark. To retrieve the execution time for a task on a surrogate, it utilizes a history-based profiling of tasks and of (task, surrogate)-pairs, where an application developer defines the task complexity.

Cheng et al. [CLWG15] propose a three-layer architecture for a distribution of application parts. A wearable device (first layer) distributes application parts to mobile devices like smart phones (middle layer) or to the remote cloud (third layer). To this end, they formulate an optimization problem that maximizes the number of distributed parts subjected to a given delay of the execution time. Cheng et al. provide a proof that the formulated optimization problem is NP-hard and approximate the optimal
9.6. Summary

In the literature, multiple approaches propose techniques for computation offloading related to the contributions in this dissertation. In detail, the work related to an efficient code offloading with annotations described in Section 9.1 also proposes computation offloading between a mobile device and a remote resource. However, the related work does not provide an efficient distribution that minimizes the execution time, energy consumption, and monetary cost for application parts that are offloadable with least interventions from application developers and end users based on an annotation-based instrumentation of Java bytecode instructions (cf. Chapter 4). The work related to a robust code offloading through safe-point’ing described in Section 9.2 also proposes for computation offloading approaches to handle (network or node) failures during a remote execution of application parts on a remote resource. However, the related work does not provide a robust distribution with safe-point’ing that locally
9. Related Work

re-uses intermediate states from the remote side to continue a remote execution on a mobile device, not abandoning partial results achieved on a remote resource so far (cf. Chapter 5). The work related to a deadline-aware code offloading with predictive safe-point'ing described in Section 9.3 also proposes scheduling strategies for a local re-use of intermediate states from the remote side (safe-points). However, the related work does not provide an adaptive approach based on prediction models for the scheduling strategy to dynamically adapt the point in times for a creation and transmission of safe-points due to dynamic conditions of the communication network (cf. Chapter 6). The work related to an optimized code offloading through cooperative caching described in Section 9.4 also proposes for computation offloading an integration of a cache to speed up the time required, for instance, for a transmission of an input execution state. However, the related work does not provide a cooperative cache on the remote side storing output execution states from mobile devices and remote resources to speed up the total time and energy required for an execution of an application part distributed to a remote resource (cf. Chapter 7). The work related to a code offloading in environments with multiple tiers described in Section 9.5 also proposes computation offloading in environments with two tiers like a mobile device and a remote resource and seldom in environments with three tiers like a mobile device, a fog resource, and a cloud resource. However, the related work does not provide computation offloading in unknown environments with multiple tires, not requiring a global view (cf. Chapter 8).
Chapter 10

Conclusion

By offloading computation from a (resource-constrained) mobile device to a (powerful) remote resource, a mobile device benefits significantly through a higher energy efficiency and a faster execution speed. In the area of code offloading, this dissertation covers different topics like a basic distribution for an efficient offloading of computation, a preemptable distribution for an efficient failure handling, or a bubbling distribution for efficient computation offloading in future environments with multiple tiers. This chapter concludes this dissertation, first presenting a summary described in Section 10.1 on the topics covered in this dissertation together with concluding remarks on key results. Afterwards, it gives an outlook described in Section 10.2 on research questions remained open, guiding future work in the domain of code offloading.

10.1. Summary

Mobile devices like smart phones, tablets, or laptops have become an integral part in daily life, where an end user, for instance, works on a project, plays a mobile game, or surfs the web on its way. To this end, an end user executes on its battery-operated mobile device different applications like an office suite, a high-end video game, or a web browser, to accomplish a desired task. Due to the mobility provided to end users by a battery-operated mobile device, however, a mobile device faces fast its main limiting factor, the energy consumption. A heavy utilization of a mobile device via, for instance, an execution of a resource-intensive application like a high-end video game drains its limited battery in few hours. Beside the limited battery capacity of a mobile device, the issue of mobility also influences further hardware resources like its processing unit, making a mobile device a resource-constrained computing environment – especially compared with the virtual unlimited performance of cloud platforms.

To handle the resource limitation of mobile devices, code offloading augments a (resource-constrained) mobile computing environment like a smart phone via a wireless network like Wi-Fi with a (powerful) remote resource like a server instance at a (commercial) data center of a cloud provider. Thus, code offloading combines mobile
computing with cloud computing to mobile cloud computing, bringing the cloud closer to end users together with its advantages like virtual unlimited resources (e.g., storage, processing, . . .), flexibility (e.g., on-demand up or down scale), or cost efficiency (e.g., pay-as-you-go manner). The augmentation involves the offloading of resource-intensive computations from a mobile computing environment to a remote resource, where the remote execution increases the energy efficiency of a mobile computing environment and the execution speed of an application. Due to the utilization of remote resources like a server instance at the cloud, an end user has to pay a monetary cost. Please note that in today’s landscape of mobile cloud computing, highly distributed heterogeneous resources like smart wearables, laptops, desktops, smart cars, or resources at the fog and the cloud surround a mobile device like a smart phone, differing in performance characteristics and cost implications.

To offload computation from a mobile computing environment (client) to a remote resource (service), a distribution process on the client comprises the eight steps of identifying, gathering, determining, sending, waiting, receiving, installing, and continuing. First, a client has to identify application parts that are offloadable, where a remote execution on a service is more beneficial than a local execution on the client. An application part that is offloadable only utilizes a processing unit – without accessing further resources like a local sensor. After identifying an application part for a remote execution, a client gathers the input information required for a remote execution like method parameters or global variables. Moreover, it also has to gather further parameters about the current situation like network quality or performance of a service, to determine the execution side of an application part. To this end, the client regards the cost for a local execution and the cost for a remote execution. For an application part, the cost for a local execution includes the execution time and energy consumption for a local execution, where the cost for a remote execution includes the execution time, energy consumption, and monetary cost for a remote execution. In case of a local execution, a client just continues the execution of an application locally, whereas it sends the input information required for a remote execution to a remote service in case of a remote execution. During a remote execution of an application part, a client waits in idle mode for the end to receive the output information of the execution on the remote service. After receiving an output information resulted from a remote execution, a client installs the output information locally and continues the execution of an application just like in case of a local execution of the application part.

Regarding the distribution process described above, an offloading framework – that enables a client to offload computation to a remote resource – has to keep the overhead introduced by the steps for a computation offloading on a client and on a service
low, providing a high energy efficiency and a fast execution speed. Moreover, it also has to keep the interventions required by an application developer or an end user low, providing an ease of use computation offloading. To this end, this dissertation presented the basic distribution for an efficient code offloading with least interventions, where a (resource-poor) mobile computing environment offloads code to a (powerful) remote resource. It utilizes an annotation-based instrumentation of Java bytecode instructions to efficiently identify feasible parts from an application for a remote execution. As a result, an application developer only annotates feasible parts at development-time, where this little application-specific knowledge is enough that the offloading framework does the rest of the distribution process at run-time on its own. The same applies to an end user which only specifies weights for the execution time, energy consumption, and monetary cost. The evaluation results of different prototypes implemented on the Jikes Research Virtual Machine, the OpenJDK, and the Android Open-Source Project show the efficiency and performance of the basic distribution, outperforming significantly a local execution on a mobile computing environment.

Regarding the distribution process described above, an offloading framework has to handle the occurrence of failures like node or network failures. For instance, a long-lasting disconnection between a client and a remote service near the end of a remote execution prevents a receive of the output information on the client. Due to the occurrence of failures, a client starts mechanisms for error handling like a local re-execution. To this end, this dissertation presented the preemptable distribution for a robust code offloading through safe-point’ing to increase the energy efficiency and responsiveness also under (node or/and network) failures. The preemptable distribution allows an interruption of computation offloading after a failure without losing the intermediate result calculated remotely so far until the occurrence of a failure. This avoids a complete re-execution of an application part locally or an (undefined long) waiting for a re-connection to a remote service. The interruption of computation offloading relies on safe-points that contain all information required to continue a remote execution locally. Thus, a remote service creates and transmits such safe-points to a client during a remote execution whenever it is beneficial in terms of energy. Although the creation and transmission of safe-points introduce some overhead, the evaluation results show that this overhead is small and quickly pays off in scenarios with link failures. As a result, the preemptable distribution leads to lower execution time and energy consumption.

Regarding the preemptable distribution described above, an offloading framework has to determine optimal points in time for the creation and transmission of safe-points. For instance, fewer safe-points are sufficient for a communication network with less frequent failures reducing the communication overhead, whereas a communica-
10. Conclusion

tion network with more frequent failures require more safe-points to be prepared for impending network failures. To this end, this dissertation presented the predictive distribution for a deadline-aware code offloading with predictive safe-point’ing. It optimizes the schedule for the creation and transmission of safe-points to minimize the introduced overhead and guarantee a maximum time of an execution for a minimal responsiveness. The predictive distribution utilizes an adaptive algorithm for scheduling safe-points that adapts dynamically the point in time of creating and transmitting safe-points due to dynamic conditions of the communication network. To determine the optimal points in time for safe-point’ing, it predicts the quality of the network link and solves a given constrained optimization problem (minimum overhead for safe-point’ing under a constraint for the execution deadline). The evaluation results show that the predictive distribution increases the energy efficiency and execution speed compared to the basic distribution and the preemptable distribution by additionally guaranteeing a maximum time for an execution under failures (improved responsiveness).

Regarding the distribution process described above, an offloading framework has to speed up a remote execution (e.g., the three steps of sending, waiting, and receiving) from an application part, to keep the energy efficiency and execution speed high. In computer science, many distributed systems utilize a cache for data and function to speed up future requests by storing information of current requests in a cache. To this end, this dissertation presented the caching-aware distribution for an optimized code offloading through cooperative caching. A cache on the remote side stores output information from local executions on mobile computing environments or remote executions on services. In case of a distribution of an application part that is offloadable, a mobile computing environment queries a cache on the remote side, avoiding, on the one hand, a repeated execution of application parts already executed previously, and reducing, on the other hand, the time required for a remote execution in case of a cache hit. As a result, a mobile computing environment benefits not only from shorter execution times for application parts but also from lower monetary cost, not utilizing a remote service for an execution in case of a cache hit. The evaluation results show that the caching-aware distribution increases the energy efficiency and execution speed as well as decreases the monetary cost compared to the basic distribution, keeping the overhead introduced for caching low.

Regarding the distribution process described above, an offloading framework nowadays has to offload computation not only between a mobile computing environment (first tier) and a remote service (second tier) but also between a mobile computing environment (first tier) and multiple highly distributed heterogeneous resources (multiple tiers) with different performance characteristics and cost implications. For instance,
a mobile computing environment can offload application parts among heterogeneous
compute resources on many tiers (e.g., smart wearables ↔ laptops ↔ desktops ↔ edge
servers ↔ cloud servers, etc.), making a decision for code offloading more complex.
To this end, this dissertation presented the bubbling distribution for a code offloading
in environments with multiple tiers. The concept of code bubbling enables each tier
to make autonomous decisions for code offloading without the need to gather a global
view onto all resources of all tiers. Using the concept of code bubbling, code offloading
potentially scales to an arbitrary number of tiers since decisions for code offloading
only involve neighboring tiers. The evaluation results show that the bubbling distribu-
tion enables an efficient code offloading for environments with multiple tiers in today’s
landscape of mobile cloud computing, keeping the introduced overhead low as well as
the energy efficiency and execution speed high.

10.2. Outlook

Apart from a direct improvement to the basic distribution, the preemptable distri-
bution, the predictive distribution, the caching-aware distribution, or the bubbling
distribution presented in this dissertation, additional information, optimization, and
functionality improves each distribution as follows.

For the basic distribution, an application developer annotates feasible parts from
an application as “offloadable” for an execution on a remote service, relying on a little
application-specific knowledge from an application developer. An application part that
is offloadable only requires a processing unit, not accessing further resources like a local
sensor. Moreover, a suitable candidate for a remote execution is offloadable and has a
long running computation with a small size of input information required for a remote
execution and of output information received on a client. Thus, an improvement to the
basic distribution can be an approach that automatically identifies and proposes suit-
able application parts at development-time based on criteria for a suitable candidate.
This supports an application developer in its decision to annotate suitable application
parts with “offloadable”.

For the preemptable distribution and the predictive distribution, a remote service at
the cloud creates and transmits safe-points during an execution of an application part,
enabling an interruption of a remote execution and a local continuation on a client
based on the safe-point received most recently. Despite the low overhead introduced
on a client and a remote service, the main overhead for a client constitutes the com-
munication cost of receiving safe-points while waiting in idle mode during a remote
execution. Thus, an improvement to the preemptable distribution and the predictive
10. Conclusion

distribution can be a buffer of safe-points at, for instance, the last communication hop (access point) to a client. Only in case of the occurrence of a failure the buffer sends the safe-points to the client increasing the energy efficiency on the client.

For the caching-aware distribution, a cache on the remote side stores output information from application parts previously executed on clients or remote services. Thus, a client queries the cache for corresponding output information for application parts that are offloadable. The significant increase of the energy efficiency and execution speed on a client depends mainly on the hit ratio at the cache on the remote side. Thus, an improvement to the caching-aware distribution can be a hierarchical composition of multiple caches at a client, an access point, the fog, and the cloud, where an important entry of a cache flows up the hierarchy improving the overall hit ratio of caching and thus, further increasing the energy efficiency and execution speed.

For the bubbling distribution, a client gathers the input information required for a remote execution creating an offload request. Thus, it has to interrupt the local execution of a corresponding application part. Due to the interruption of a local execution to gather and create an offload request, the execution time (and energy consumption) is worse compared to a pure local execution. Thus, an improvement to the bubbling distribution can be an optimization to the local creation of an offload request that first only copies the input information from an application part, continues the local execution, and then starts the steps required for code bubbling.

For all distributions, a remote service, for instance, at an underutilized desktop, a server instance at the fog or the cloud executes an application part from a client. Thus, an end user has to trust a corresponding (personal, fog, or cloud) provider of a remote service. The remote processing of information like an execution of application parts or a caching of output information raises the question of privacy and security implications to an end user. For instance, an execution of an application part requires personal information or a caching of output information from a specific client reveals its execution flow. Thus, an improvement to all distributions can be the consideration of privacy and security implications on the remote side that prevents an end user from misuse of, for instance, its personal information.
Appendix
A. Java Bytecode Instructions

Table A.1, Table A.2, and Table A.3 outline the Java bytecode instructions:

<table>
<thead>
<tr>
<th>Constants:</th>
<th>Loads:</th>
<th>Stores:</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 (0x00)</td>
<td>iload</td>
<td>istore</td>
</tr>
<tr>
<td>01 (0x01)</td>
<td>iload</td>
<td>istore</td>
</tr>
<tr>
<td>02 (0x02)</td>
<td>iload</td>
<td>istore</td>
</tr>
<tr>
<td>03 (0x03)</td>
<td>iload</td>
<td>istore</td>
</tr>
<tr>
<td>04 (0x04)</td>
<td>iload</td>
<td>istore</td>
</tr>
<tr>
<td>05 (0x05)</td>
<td>iload</td>
<td>istore</td>
</tr>
<tr>
<td>06 (0x06)</td>
<td>iload</td>
<td>istore</td>
</tr>
<tr>
<td>07 (0x07)</td>
<td>iload</td>
<td>istore</td>
</tr>
<tr>
<td>08 (0x08)</td>
<td>iload</td>
<td>istore</td>
</tr>
<tr>
<td>09 (0x09)</td>
<td>iload</td>
<td>istore</td>
</tr>
<tr>
<td>10 (0x0a)</td>
<td>iload</td>
<td>istore</td>
</tr>
<tr>
<td>11 (0x0b)</td>
<td>iload</td>
<td>istore</td>
</tr>
<tr>
<td>12 (0x0c)</td>
<td>iload</td>
<td>istore</td>
</tr>
<tr>
<td>13 (0x0d)</td>
<td>iload</td>
<td>istore</td>
</tr>
<tr>
<td>14 (0x0e)</td>
<td>iload</td>
<td>istore</td>
</tr>
<tr>
<td>15 (0x0f)</td>
<td>iload</td>
<td>istore</td>
</tr>
<tr>
<td>16 (0x10)</td>
<td>iload</td>
<td>istore</td>
</tr>
<tr>
<td>17 (0x11)</td>
<td>iload</td>
<td>istore</td>
</tr>
<tr>
<td>18 (0x12)</td>
<td>iload</td>
<td>istore</td>
</tr>
<tr>
<td>19 (0x13)</td>
<td>iload</td>
<td>istore</td>
</tr>
<tr>
<td>20 (0x14)</td>
<td>iload</td>
<td>istore</td>
</tr>
<tr>
<td>21 (0x15)</td>
<td>iload</td>
<td>iload_0</td>
</tr>
<tr>
<td>22 (0x16)</td>
<td>iload</td>
<td>iload_1</td>
</tr>
<tr>
<td>23 (0x17)</td>
<td>iload</td>
<td>iload_2</td>
</tr>
<tr>
<td>24 (0x18)</td>
<td>iload</td>
<td>iload_3</td>
</tr>
<tr>
<td>25 (0x19)</td>
<td>iload</td>
<td>iload_0</td>
</tr>
<tr>
<td>26 (0x1a)</td>
<td>iload</td>
<td>iload_1</td>
</tr>
<tr>
<td>27 (0x1b)</td>
<td>iload</td>
<td>iload_2</td>
</tr>
<tr>
<td>28 (0x1c)</td>
<td>iload</td>
<td>iload_3</td>
</tr>
<tr>
<td>29 (0x1d)</td>
<td>iload</td>
<td>iload_0</td>
</tr>
<tr>
<td>30 (0x1e)</td>
<td>iload</td>
<td>iload_1</td>
</tr>
<tr>
<td>31 (0x1f)</td>
<td>iload</td>
<td>iload_2</td>
</tr>
<tr>
<td>32 (0x20)</td>
<td>iload</td>
<td>iload_3</td>
</tr>
<tr>
<td>33 (0x21)</td>
<td>iload</td>
<td>iload_0</td>
</tr>
<tr>
<td>34 (0x22)</td>
<td>iload</td>
<td>iload_1</td>
</tr>
<tr>
<td>35 (0x23)</td>
<td>iload</td>
<td>iload_2</td>
</tr>
<tr>
<td>36 (0x24)</td>
<td>iload</td>
<td>iload_3</td>
</tr>
<tr>
<td>37 (0x25)</td>
<td>iload</td>
<td>iload_0</td>
</tr>
<tr>
<td>38 (0x26)</td>
<td>iload</td>
<td>iload_1</td>
</tr>
<tr>
<td>39 (0x27)</td>
<td>iload</td>
<td>iload_2</td>
</tr>
<tr>
<td>40 (0x28)</td>
<td>iload</td>
<td>iload_3</td>
</tr>
<tr>
<td>41 (0x29)</td>
<td>iload</td>
<td>iload_0</td>
</tr>
<tr>
<td>42 (0x2a)</td>
<td>iload</td>
<td>iload_1</td>
</tr>
<tr>
<td>43 (0x2b)</td>
<td>iload</td>
<td>iload_2</td>
</tr>
<tr>
<td>44 (0x2c)</td>
<td>iload</td>
<td>iload_3</td>
</tr>
<tr>
<td>45 (0x2d)</td>
<td>iload</td>
<td>iload_0</td>
</tr>
<tr>
<td>46 (0x2e)</td>
<td>iload</td>
<td>iload_1</td>
</tr>
<tr>
<td>47 (0x2f)</td>
<td>iload</td>
<td>iload_2</td>
</tr>
<tr>
<td>48 (0x30)</td>
<td>iload</td>
<td>iload_3</td>
</tr>
<tr>
<td>49 (0x31)</td>
<td>iload</td>
<td>iload_0</td>
</tr>
<tr>
<td>50 (0x32)</td>
<td>iload</td>
<td>iload_1</td>
</tr>
<tr>
<td>51 (0x33)</td>
<td>iload</td>
<td>iload_2</td>
</tr>
<tr>
<td>52 (0x34)</td>
<td>iload</td>
<td>iload_3</td>
</tr>
<tr>
<td>53 (0x35)</td>
<td>iload</td>
<td>iload_0</td>
</tr>
</tbody>
</table>

Table A.1.: Part I of the Java bytecode instructions from 00 to 86. [LYBB15]
## A. Java Bytecode Instructions

### Math:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>96</td>
<td>(0x60)</td>
<td>iadd</td>
</tr>
<tr>
<td>97</td>
<td>(0x61)</td>
<td>ladd</td>
</tr>
<tr>
<td>98</td>
<td>(0x62)</td>
<td>fadd</td>
</tr>
<tr>
<td>99</td>
<td>(0x63)</td>
<td>dadd</td>
</tr>
<tr>
<td>100</td>
<td>(0x64)</td>
<td>isub</td>
</tr>
<tr>
<td>101</td>
<td>(0x65)</td>
<td>lsub</td>
</tr>
<tr>
<td>102</td>
<td>(0x66)</td>
<td>fsub</td>
</tr>
<tr>
<td>103</td>
<td>(0x67)</td>
<td>dsub</td>
</tr>
<tr>
<td>104</td>
<td>(0x68)</td>
<td>imul</td>
</tr>
<tr>
<td>105</td>
<td>(0x69)</td>
<td>lmul</td>
</tr>
<tr>
<td>106</td>
<td>(0x6a)</td>
<td>fmul</td>
</tr>
<tr>
<td>107</td>
<td>(0x6b)</td>
<td>dmul</td>
</tr>
<tr>
<td>108</td>
<td>(0x6c)</td>
<td>idiv</td>
</tr>
<tr>
<td>109</td>
<td>(0x6d)</td>
<td>ldiv</td>
</tr>
<tr>
<td>110</td>
<td>(0x6e)</td>
<td>fdiv</td>
</tr>
<tr>
<td>111</td>
<td>(0x6f)</td>
<td>ddiv</td>
</tr>
<tr>
<td>112</td>
<td>(0x70)</td>
<td>irem</td>
</tr>
<tr>
<td>113</td>
<td>(0x71)</td>
<td>lrem</td>
</tr>
<tr>
<td>114</td>
<td>(0x72)</td>
<td>frem</td>
</tr>
<tr>
<td>115</td>
<td>(0x73)</td>
<td>drem</td>
</tr>
<tr>
<td>116</td>
<td>(0x74)</td>
<td>ineg</td>
</tr>
<tr>
<td>117</td>
<td>(0x75)</td>
<td>lneg</td>
</tr>
<tr>
<td>118</td>
<td>(0x76)</td>
<td>fneg</td>
</tr>
<tr>
<td>119</td>
<td>(0x77)</td>
<td>dneg</td>
</tr>
<tr>
<td>120</td>
<td>(0x78)</td>
<td>ishl</td>
</tr>
<tr>
<td>121</td>
<td>(0x79)</td>
<td>lshl</td>
</tr>
<tr>
<td>122</td>
<td>(0x7a)</td>
<td>ishr</td>
</tr>
<tr>
<td>123</td>
<td>(0x7b)</td>
<td>lshr</td>
</tr>
<tr>
<td>124</td>
<td>(0x7c)</td>
<td>iushr</td>
</tr>
<tr>
<td>125</td>
<td>(0x7d)</td>
<td>lushr</td>
</tr>
<tr>
<td>126</td>
<td>(0x7e)</td>
<td>iand</td>
</tr>
<tr>
<td>127</td>
<td>(0x7f)</td>
<td>land</td>
</tr>
<tr>
<td>128</td>
<td>(0x80)</td>
<td>ior</td>
</tr>
<tr>
<td>129</td>
<td>(0x81)</td>
<td>lor</td>
</tr>
<tr>
<td>130</td>
<td>(0x82)</td>
<td>ixor</td>
</tr>
<tr>
<td>131</td>
<td>(0x83)</td>
<td>lxor</td>
</tr>
<tr>
<td>132</td>
<td>(0x84)</td>
<td>iinc</td>
</tr>
</tbody>
</table>

### Stack:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>87</td>
<td>(0x57)</td>
<td>pop</td>
</tr>
<tr>
<td>88</td>
<td>(0x58)</td>
<td>pop2</td>
</tr>
<tr>
<td>89</td>
<td>(0x59)</td>
<td>dup</td>
</tr>
<tr>
<td>90</td>
<td>(0x5a)</td>
<td>dup_x1</td>
</tr>
<tr>
<td>91</td>
<td>(0x5b)</td>
<td>dup_x2</td>
</tr>
<tr>
<td>92</td>
<td>(0x5c)</td>
<td>dup2</td>
</tr>
<tr>
<td>93</td>
<td>(0x5d)</td>
<td>dup2_x1</td>
</tr>
<tr>
<td>94</td>
<td>(0x5e)</td>
<td>dup2_x2</td>
</tr>
<tr>
<td>95</td>
<td>(0x5f)</td>
<td>swap</td>
</tr>
</tbody>
</table>

### Conversions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>133</td>
<td>(0x85)</td>
<td>i2l</td>
</tr>
<tr>
<td>134</td>
<td>(0x86)</td>
<td>i2f</td>
</tr>
<tr>
<td>135</td>
<td>(0x87)</td>
<td>i2d</td>
</tr>
<tr>
<td>136</td>
<td>(0x88)</td>
<td>l2i</td>
</tr>
<tr>
<td>137</td>
<td>(0x89)</td>
<td>l2f</td>
</tr>
<tr>
<td>138</td>
<td>(0x8a)</td>
<td>l2d</td>
</tr>
<tr>
<td>139</td>
<td>(0x8b)</td>
<td>f2i</td>
</tr>
<tr>
<td>140</td>
<td>(0x8c)</td>
<td>f2l</td>
</tr>
<tr>
<td>141</td>
<td>(0x8d)</td>
<td>f2d</td>
</tr>
<tr>
<td>142</td>
<td>(0x8e)</td>
<td>d2i</td>
</tr>
<tr>
<td>143</td>
<td>(0x8f)</td>
<td>d2l</td>
</tr>
<tr>
<td>144</td>
<td>(0x90)</td>
<td>d2f</td>
</tr>
<tr>
<td>145</td>
<td>(0x91)</td>
<td>i2b</td>
</tr>
<tr>
<td>146</td>
<td>(0x92)</td>
<td>i2c</td>
</tr>
<tr>
<td>147</td>
<td>(0x93)</td>
<td>i2s</td>
</tr>
</tbody>
</table>

Table A.2.: Part II of the Java bytecode instructions from 87 to 147. [LYBB15]
<table>
<thead>
<tr>
<th>Comparisons:</th>
<th>References:</th>
</tr>
</thead>
<tbody>
<tr>
<td>148 (0x94) lcm</td>
<td>178 (0xb2) getstatic</td>
</tr>
<tr>
<td>149 (0x95) fcmpl</td>
<td>179 (0xb3) putstatic</td>
</tr>
<tr>
<td>150 (0x96) fcmpl</td>
<td>180 (0xb4) getfield</td>
</tr>
<tr>
<td>151 (0x97) dcmpl</td>
<td>181 (0xb5) putfield</td>
</tr>
<tr>
<td>152 (0x98) dcmpl</td>
<td>182 (0xb6) invokevirtual</td>
</tr>
<tr>
<td>153 (0x99) ifeq</td>
<td>183 (0xb7) invokespecial</td>
</tr>
<tr>
<td>154 (0x9a) ifne</td>
<td>184 (0xb8) invokestatic</td>
</tr>
<tr>
<td>155 (0x9b) iflt</td>
<td>185 (0xb9) invokeinterface</td>
</tr>
<tr>
<td>156 (0x9c) ifge</td>
<td>186 (0xba) invokedynamic</td>
</tr>
<tr>
<td>157 (0x9d) ifgt</td>
<td>187 (0xbb) new</td>
</tr>
<tr>
<td>158 (0x9e) ifle</td>
<td>188 (0xbc) newarray</td>
</tr>
<tr>
<td>159 (0x9f) if_icmpeq</td>
<td>189 (0xbd) anewarray</td>
</tr>
<tr>
<td>160 (0xa0) if_icmpne</td>
<td>190 (0xbe) arraylength</td>
</tr>
<tr>
<td>161 (0xa1) if_icmplt</td>
<td>191 (0xbf) athrow</td>
</tr>
<tr>
<td>162 (0xa2) if_icmpgt</td>
<td>192 (0xc0) checkcast</td>
</tr>
<tr>
<td>163 (0xa3) if_icmpge</td>
<td>193 (0xc1) instanceof</td>
</tr>
<tr>
<td>164 (0xa4) if_icmple</td>
<td>194 (0xc2) monitorenter</td>
</tr>
<tr>
<td>165 (0xa5) if_acmpeq</td>
<td>195 (0xc3) monitorexit</td>
</tr>
<tr>
<td>166 (0xa6) if_acmpne</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Control:</th>
<th>Extended:</th>
</tr>
</thead>
<tbody>
<tr>
<td>167 (0xa7) goto</td>
<td>196 (0xc4) wide</td>
</tr>
<tr>
<td>168 (0xa8) jsr</td>
<td>197 (0xc5) multianewarray</td>
</tr>
<tr>
<td>169 (0xa9) ret</td>
<td>198 (0xc6) ifnull</td>
</tr>
<tr>
<td>170 (0xaa) tableswitch</td>
<td>199 (0xc7) ifnonnull</td>
</tr>
<tr>
<td>171 (0xab) lookupswitch</td>
<td>200 (0xc8) goto_w</td>
</tr>
<tr>
<td>172 (0xac) ireturn</td>
<td>201 (0xc9) jsr_w</td>
</tr>
<tr>
<td>173 (0xad) lreturn</td>
<td></td>
</tr>
<tr>
<td>174 (0xae) freturn</td>
<td></td>
</tr>
<tr>
<td>175 (0xaf) dreturn</td>
<td></td>
</tr>
<tr>
<td>176 (0xb0) areturn</td>
<td></td>
</tr>
<tr>
<td>177 (0xb1) return</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reserved:</th>
</tr>
</thead>
<tbody>
<tr>
<td>202 (0xca) breakpoint</td>
</tr>
<tr>
<td>254 (0xfe) impdep1</td>
</tr>
<tr>
<td>255 (0xff) impdep2</td>
</tr>
</tbody>
</table>

Table A.3.: Part III of the Java bytecode instructions from 148 to 255. [LYBB15]
B. Mobile Applications

An end user executes nowadays different mobile applications on its mobile device like high-end video games. To this end, this chapter highlights capabilities of the mobile applications utilized in this dissertation for code offloading. Due to significant differences with regard to computational complexity and communication overhead of the mobile applications (cf. Table B.1), each mobile application benefits differently from code offloading. In detail, this dissertation utilizes a Chesspresso application (cf. Section B.2), a chess game (cf. Section B.3), a face recognition application (cf. Section B.4), and a text-to-voice application (cf. Section B.5).

B.1. “Hello, World!” Application

The Java programming language is object-oriented, where every data is an object except the Java primitive types boolean, char, short, integer, float, and double owing to performance issues. An application developer writes the application code inside classes, where the public class contained in a source file plus the suffix .java define its file name. The following listing, Listing B.1 “HelloWorld.java”, shows the popular “Hello, World!” application written in the Java programming language:

```java
class HelloWorld {
    public static void main(String[] args) {
        // Prints the String "Hello, World!" to the console
        System.out.println("Hello, World!");
    }
}
```

Listing B.1: The Java source code for the “Hello, World!” application.

B.2. Chesspresso Application

The Chesspresso application is a chess program that searches iteratively for the best move of the black player after the white player moved its pawn two steps ahead (cf. Figure B.1). It utilizes the Java library Chesspresso from Bernhard Seybold [Sey03] that is
B. Mobile Applications

<table>
<thead>
<tr>
<th></th>
<th>Size of Input Execution State</th>
<th>Code Complexity</th>
<th>Size of Output Execution State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chesspresso Application</td>
<td>6050 bytes ⇒ constant small size</td>
<td>High computational complexity</td>
<td>3023 bytes ⇒ constant small size</td>
</tr>
<tr>
<td>Chess Game</td>
<td>$80 + (n \cdot 80) + 1$ integers ⇒ small size</td>
<td>With a chess difficulty of 3 very high, with 2 moderate, and with 1 low</td>
<td>5 integers ⇒ very small, constant size</td>
</tr>
<tr>
<td>Face Recognition Application</td>
<td>Width times height times color space ⇒ huge size (in the magnitude of megabytes)</td>
<td>High computational complexity</td>
<td>$4 \cdot n$ integers, where $n$ is the number of detected faces</td>
</tr>
<tr>
<td>Text-to-Voice Application</td>
<td>Written sentence and voice name ⇒ moderate size (in the magnitude of kilobytes)</td>
<td>High computational complexity</td>
<td>Audio file ⇒ moderate size (in the magnitude of hundred kilobytes)</td>
</tr>
</tbody>
</table>

Table B.1.: Comparison of the main facts from the Java applications evaluated for code offloading on the prototypes.

Figure B.1.: The opening move for the Chesspresso application.
B.2. Chesspresso Application

open-source and provides the necessary functionality like data structures and rules to implement a chess game. After an initialization part, the Chesspresso application calls the Java method `iterativeChessCalls(...)` of the Java class `ChessDetectPosition` that is offloadable. It has the input parameters of the current chess board (cf. Figure B.1) and an array of empty Strings (cf. Listing B.2). As output, the Java method has an Integer value indicating the number of calls for the Java method `check(...)` (cf. Listing B.2). Regarding the size of the input parameters, the Java class `Position` has a size of 2951 bytes and the array of Strings a size of 72 bytes, summing up to 3023 bytes (cf. Listing B.2). As both input parameters are Java objects, the output includes both input parameters now with a size of 5902 bytes and 144 bytes – as well as the 4 bytes for the return of an integer value, summing up to 6050 bytes. Summarizing, the search algorithm from the Chesspresso application is an optimal candidate for a computation offloading. It possesses a constant, small size of input and a constant, small size of output with a high computational complexity. The following listing shows the relevant code for the Chesspresso application:

```java
package chesspresso;

import chesspresso.Chess;
import chesspresso.move.IllegalMoveException;
import chesspresso.move.Move;
import chesspresso.position.Position;

public class ChessDetectPosition {

    public int iterativeChessCalls(Position position, String[] game) {
        short[] nm1 = position.getAllMoves();
        short[] nm2; short[] nm3; short[] nm4;

        // record
        int ply = position.getPlyNumber();
        Move lastMove = position.getLastMove();
        game[ply - 1] = lastMove.getSAN();

        MainChess.TEST = game.length;

        boolean found = false;
        int counter = 0;
        try {
            for (int i1=0; i1<nm1.length; i1++) {
                System.out.println("i1: " + i1);

                position.doMove(nm1[i1]);
                nm2 = position.getAllMoves();
```
B. Mobile Applications

```java
// record
ply = position.getPlyNumber();
lastMove = position.getLastMove();
game[ply - 1] = lastMove.getSAN();

// check and count
found = check(position, lastMove, ply, game);
counter++;

// one down
for (int i2=0; i2<nm2.length; i2++) {
    System.out.println(" - i2: " + i2);
    position.doMove(nm2[i2]);
    nm3 = position.getAllMoves();
}

// record
ply = position.getPlyNumber();
lastMove = position.getLastMove();
game[ply - 1] = lastMove.getSAN();

// one down
for (int i3=0; i3<nm3.length; i3++) {
    System.out.println("i3: " + i3);
    position.doMove(nm3[i3]);
    nm4 = position.getAllMoves();
}

// record
ply = position.getPlyNumber();
lastMove = position.getLastMove();
game[ply - 1] = lastMove.getSAN();

// one down
for (int i4=0; i4<nm4.length; i4 += 2) {
    System.out.println("i4: " + i4);
    position.doMove(nm4[i4]);
    nm5 = position.getAllMoves();
}

// record
ply = position.getPlyNumber();
lastMove = position.getLastMove();
game[ply - 1] = lastMove.getSAN();

// check and count
found = check(position, lastMove, ply, game);
counter++;
position.undoMove();
}
position.undoMove();
}
position.undoMove();

boolean debug = position.undoMove();
if (debug) counter++;
```
B.3. Chess Game

The chess game enables an end user to play a chess game against a computer opponent, where the source code of the chess game is from Ulf Ochsenfahrt [Och08].

The Graphical User Input (GUI) of the chess game displays the current configuration of the chess board, takes the next move of the user, and displays the determined move of the computer opponent. To determine the next best chess move of the computer opponent, the search algorithm uses an Alpha-beta pruning. The input parameters are the current configuration of the chess board, the chess history of past moves, and the search depth (cf. Listing B.3). Notice that the search depth defines the chess difficulty of the computer opponent, influencing significantly the code complexity of the search algorithm. As output, the search algorithm returns the next best chess move of the computer opponent based on the given input parameters (cf. Listing B.3). Regarding the size of the input parameters, the chess board results in $8 \cdot 8 + 16 = 80$ integers, the chess history of $n$ past moves in $80 \cdot n$ integers, and the chess difficulty in 1 integer. Regarding the size of the output, the chess return sums up to 5 integers.

For the evaluations in this dissertation, a mobile device plays multiple rounds with different opening moves, namely 1 a3, 1 g3, 1 a4, 1 b4, 1 c4, 1 d4, and 1 h4.

Figure B.2 shows in the first row the chess board at the start of a new chess game (cf. Figure B.2a) and the opening moves that move a pawn one-step further (cf. Figure B.2b and B.2c). Moreover, Figure B.2 shows in the second and the third row the opening

```java
private boolean check(Position position, Move lastMove, int ply, String[] game) {
    return (position.isMate() &&
            lastMove.isCapturing() &&
            lastMove.getMovingPiece() == Chess.KNIGHT &&
            position.getPiece(lastMove.getToSqi()) == Chess.ROOK);
}
```

Listing B.2: The Java Source Code of the Chesspresso Application [Sey03]
B. Mobile Applications

Figure B.2.: The different configurations of the chess board evaluated for a chess game.
moves that move a pawn two-steps further (cf. Figure B.2d - B.2h).

Summarizing, the search algorithm from the chess game is an optimal candidate for a computation offloading. It possesses a small size of input and a very small, constant size of output with a high computational complexity that depends on the chess difficulty chosen and the current configuration of the chess board.

The following listing shows the relevant code for the chess game:

```java
import ...

public final class aichess4k extends JComponent implements Runnable {
    // board:
    // 0-7 board
    // 8 0-3 lastmove source x,y & dest x,y
    // 4 activeplayer 5 leftrookmoved 6 kingmoved 7 rightrookmoved
    // 9 0 movenum 1 movessincetake 2 check
    // ply:
    // 0-1 source position x,y
    // 2-3 dest position x,y
    // 4 target figure (only used for transformation moves)
    // current board
    private int[][] myboard;
    // move including transformation target if pawn ends up on final row
    private int[] myply;
    // all previous board configurations
    private int[][][] undo;
    // ai result
    private int[] resply;
    // ai difficulty (modifies search depth)
    private int difficulty;
    private void init() { ... }
    private boolean isCastleing(final int[][] board, final int sx, final int sy, final int dx, final int dy) { ... }
    private boolean isEnPassant(int[][] board, int sx, int sy, int dx, int dy) { ... }
    private boolean isNormal(int[][] board, int sx, int sy, int dx, int dy) { ... }
    void copy(int[][] from, int[][] to) { ... }
    boolean move(int[][] board, int[] ply) { ... }
    int search(int[][] board, final int steps, final int maxsteps, int
```
alpha, int beta) {
    // Check for a draw
    ...

    // Evaluate board if search depth is reached
    ...

    // Start alpha-beta-search
    for (int x = 0; x < 8; x++)
        for (int y = 0; y < 8; y++) {
            int figure = board[x][y];
            int type = figure & FIGURE;
            if (((figure & COLOR) == side) {
                for (int dx = 0; dx < 8; dx++) {
                    for (int dy = 0; dy < 8; dy++) {
                        if (((board[dx][dy] & COLOR) != side) &&
                            (isEnPassant(board, x, y, dx, dy) ||
                             isCastleing(board, x, y, dx, dy) ||
                             isNormal(board, x, y, dx, dy))) {
                            int newsteps = steps;
                            if ((board[dx][dy] != EMPTY) && (steps <
                                maxsteps)) newsteps++;

                            int min = QUEEN;
                            if ((type == PAWN) && ((dy == 0) || (dy == 7)))
                                min = BISHOP;
                            for (int k = min; k <= QUEEN; k++) {
                                int[] p = new int[] {x, y, dx, dy, k};
                                copy(board, save);
                                if (move(save, p)) {
                                    int val = -search(save, newsteps -1,
                                        maxsteps -1, -beta, -alpha);
                                    if (val >= beta) return beta;
                                    if (val > alpha) {
                                        alpha = val;
                                        bestply = p;
                                        break;
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }

    // Return resply
    ...
    ...
}

Listing B.3: The Java Source Code of the Chess Game [Och08]
B.4. Face Recognition Application

The face recognition application enables an end user to recognize faces on an image, where the source code of the face recognition is from Jon’s Java Imaging Library (JJIL) [Web08], containing various tasks for image processing.

The GUI loads or takes an image, displays it, and highlights, if any, faces recognized by rectangles. The algorithm for face detection applies a Haar cascade to the image based on multiple scales. The input parameters are the image together with a minimum (finest) scale and a maximum (coarsest) scale (cf. Listing B.4). As output, the algorithm for face detection returns an array of enclosing rectangles, locating the faces detected in an image (cf. Listing B.4). Regarding the size of the input parameters, the number of bytes of an image depends on its pixel resolution, whereas the minimum scale and the maximum scale are an integer each. Regarding the size of the output, the number of rectangles depends on the number of detected faces, where each rectangle has 4 integers, namely \( x, y, w \) (width), and \( h \) (height).

For the evaluations in this dissertation, a mobile device recognizes faces either in a color image with a width of 1258 pixel and a height of 1024 pixel or in a color image with a width of 960 pixel and a height of 1280 pixel.

Summarizing, the algorithm for face detection from the face recognition application is only a good candidate for a computation offloading in case of a good quality of the network up-link. It possesses a huge size of input and a small size of output with a high computational complexity.

The following listing shows the relevant code for the face recognition application:

```java
package jjil.algorithm;

/*
 * Gray8DetectHaarMultiScale.java
 *
 * Created on August 19, 2007, 7:33 PM
 *
 * Copyright 2007 by Jon A. Webb
 *
 */

import ...

public class Gray8DetectHaarMultiScale extends PipelineStage {
    private HaarClassifierCascade hcc;

    // maximum scale is the largest factor the image is divided by
    private int nMaxScale = 10;

    // minimum scale is the smallest factor the image is divided by
    private int nMinScale = 5;
}
```
B. Mobile Applications

```java
// scale change is the change in scale from one search to the next times 256
private int nScaleChange = 12 * 256 / 10;

public Gray8DetectHaarMultiScale(InputStream is, int nMinScale, int nMaxScale) throws jjil.core.Error, IOException {
    // Apply multi-scale Haar cascade and prepare a mask image showing where features were detected.
    public void push(Image image) throws jjil.core.Error {
        // Check image

        int nScale = Math.min(this.nMaxScale, Math.min(image.getWidth() / this.hcc.getWidth(), image.getHeight() / this.hcc.getHeight()));
        Gray8Image imMask = new Gray8Image(1,1,Byte.MIN_VALUE);
        while (nScale >= this.nMinScale) {
            // shrink the input image
            int nTargetWidth = imGray.getWidth() / nScale;
            int nTargetHeight = imGray.getHeight() / nScale;
            Gray8Shrink gs = new Gray8Shrink(nTargetWidth, nTargetHeight);
            gs.push(imGray);
            Gray8Image imShrunk = (Gray8Image) gs.getFront();
            Gray8RectStretch grs = new Gray8RectStretch(nTargetWidth, nTargetHeight);
            grs.push(imMask);
            imMask = (Gray8Image) grs.getFront();
            // scale the mask to the new size
            Gray8MaskedImage gmi = new Gray8MaskedImage(imShrunk, imMask);
            // pass the masked image to a subimage generator
            MaskedGray8SubImgGen mgsi = new MaskedGray8SubImgGen(...);
            mgsi.push(gmi);
            // now run Haar detection on each scaled image
            int nxLastFound = -hcc.getWidth();
            int nyLastFound = -hcc.getHeight();
            while (!mgsi.isEmpty()) {
                Gray8OffsetImage imSub = (Gray8OffsetImage) mgsi.getFront();
                if (imSub.getXOffset() > nxLastFound + hcc.getWidth() && imSub.getYOffset() > nyLastFound + hcc.getHeight()) {
                    if (hcc.eval(imSub)) {
                        // Found something.
                        nxLastFound = imSub.getXOffset();
                        nyLastFound = imSub.getYOffset();
                        // assign Byte.MAX_VALUE to the feature area so we don’t search it again
                        Gray8Rect gr = new Gray8Rect(...);
                        gr.push(imMask);
                        imMask = (Gray8Image) gr.getFront();
                    }
                }
            }
            nScale = nScale * 256 / this.nScaleChange;
        }
        // Stretch imMask to original image size; this is the result
    }
}
```
The text-to-voice application enables an end user to synthesize a written sentence to spoken speech, where the source code of the text-to-voice application is from FreeTTS [Lab09]. It is an open-source implementation of a speech synthesis engine, totally written in the Java programming language.

The GUI has a field to enter a sentence and a button to start the speech synthesis. The input parameters of the algorithm for text-to-speech are the written sentence and the name of the utilized voice, both as a String (cf. Listing B.5). As output, the algorithm for text-to-speech returns an audio file of the sentence spoken by the voice (cf. Listing B.5). Regarding the size of the input parameters, the number of bytes of a written sentence as well as the name of the voice depends on the number of characters. Regarding the size of the output, the number of bytes of an audio file depends on both the length of a sentence and the chosen voice.

For the evaluations in this dissertation, a mobile device transforms words from different text passages to voice, where the text passages correspond to the abstracts from [KAH+12] (ThinkAir), [CBC+10] (MAUI), and an own sentence (OWN). In detail, the relevant words from ThinkAir are

“Smartphones have exploded in popularity in recent years, becoming ever more sophisticated and capable. As a result, developers worldwide are building increasingly complex applications that require ever increasing amounts of computational power and energy. In …” [KAH+12]

from MAUI

“This paper presents MAUI, a system that enables fine-grained energy-aware offload of mobile code to the infrastructure. Previous approaches to these problems either relied heavily on programmer support to partition an application, or …” [CBC+10].
B. Mobile Applications

and from OWN

“This is a very long sentence where FreeTTS has to transform written text into spoken voice!”.

Summarizing, the algorithm for text-to-voice from the text-to-voice application is only a good candidate for a computation offloading in case of a good quality of the network down-link. It possesses a small size of input and a big size of output with a high computation complexity.

The following listing shows the relevant code for the text-to-voice application:

```java
/**
 * Copyright 2003 Sun Microsystems, Inc.
 */

import ...

public class FreeTTSHelloWorld {

public static void main(String[] args) {

    // The VoiceManager manages all the voices for FreeTTS.
    VoiceManager voiceManager = VoiceManager.getInstance();
    Voice helloVoice = voiceManager.getVoice(voiceName);

    // Allocates the resources for the voice.
    helloVoice.allocate();

    // Synthesize speech.
    helloVoice.speak("Thank you for giving me a voice. 
    + "I’m so glad to say hello to this world.");

    // Clean up and leave.
    helloVoice.deallocate();
    System.exit(0);
}
}
Listing B.5: The Java Source Code from of the Text-to-Voice Application [Lab09]```
C. System Devices

The landscape of today’s Mobile Cloud Computing includes plenty of different resources like smart phones, tablets, laptops, workstations, or server instances at the fog or the cloud. To this end, this chapter highlights capabilities of the system devices utilized in this dissertation for code offloading. Due to differences in the performance characteristics of the system devices (cf. Table C.1), each system device benefits differently from code offloading. In detail, this dissertation utilizes as an offloading client a smart phone – namely a Samsung Galaxy Nexus (cf. Subsection C.1) – a netbook – namely a Dell Inspiron Mini 10v (cf. Subsection C.2) – and a laptop – namely a Lenovo ThinkPad T61 (cf. Subsection C.3). Moreover, this dissertation utilizes as an offloading service a workstation – namely an HP Compaq 8200 Elite (cf. Subsection C.4) – and a server instance at the AWS EC2 – namely a t2.micro instance (cf. Subsection C.5). As the system devices connect via further devices like a Wi-Fi AP to other system devices and the Internet, this dissertation utilizes as network devices a Huawei E1750 Surf Stick (cf. Subsection C.6), a Linksys WRT54GL Wireless Router (cf. Subsection C.7), and a LevelOne GSW-0809 Gigabit Ethernet Switch (cf. Subsection C.8).

C.1. Samsung Galaxy Nexus

The Samsung Galaxy Nexus (GT-i9250) is a smart phone from the Google Nexus series, running the Android OS as mobile operating system (cf. Subsection 4.6.3). Released in the year 2011, it possesses the OMAP\textsuperscript{1} 4460 System-on-a-Chip (SoC) from Texas Instruments [Ins12]. The OMAP 4460 SoC has a dual-core CPU of ARM Cortex-A9, where each 32-bit core runs at 1.2 - 1.5 GHz. Beside the OMAP 4460 SoC, it has a memory of 1 GB and a storage of 16 GB. It communicates via Near Field Communication (NFC), Bluetooth 3.0 (cf. Subsection 2.2.3), Wi-Fi IEEE 802.11 a,b,g,n (cf. Subsection 2.2.1), or a 3.5 Generation cellular network (cf. Subsection 2.2.2). Last, a battery of 1750 mAh powers the Samsung Galaxy Nexus. [Sam11]

\textsuperscript{1}Open Multimedia Application Platform (OMAP)
C. System Devices

<table>
<thead>
<tr>
<th>System Device</th>
<th>Central Processing Unit (CPU)</th>
<th>Communication Network</th>
<th>Software Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samsung Galaxy Nexus</td>
<td>Dual-core CPU running at 1.2 GHz</td>
<td>Bluetooth 3.0 + IEEE 802.11 a,b,g, n + 3.5 Generation</td>
<td>Android OS, 32-bit</td>
</tr>
<tr>
<td>Dell Inspiron Mini 10v</td>
<td>Single-core CPU running at 1.6 GHz</td>
<td>Bluetooth 2.1 + IEEE 802.11 b,g + 3.5 Generation</td>
<td>Linux + OpenJDK, 32-bit</td>
</tr>
<tr>
<td>Lenovo ThinkPad T61</td>
<td>Dual-core CPU running at 2.4 GHz</td>
<td>IEEE 802.11 a,b,g,n + 3.5 Generation</td>
<td>Linux + OpenJDK, 64-bit</td>
</tr>
<tr>
<td>HP Compaq 8200 Elite</td>
<td>Quad-core CPU running at 3.4 GHz</td>
<td>up to 1 Gbit/s</td>
<td>Linux + OpenJDK, 64-bit</td>
</tr>
<tr>
<td>AWS EC2 t2.micro</td>
<td>Single-core CPU running at 2.4 GHz</td>
<td>at least 10 Mbit/s</td>
<td>Linux + OpenJDK, 64-bit</td>
</tr>
</tbody>
</table>

Table C.1.: Overview of the capabilities from the system devices utilized for performance measurements of code offloading.

C.2. Dell Inspiron Mini 10v

The Dell Inspiron Mini 10v (1011) is a netbook from Dell Technologies, running a minimal Linux 32-bit version based on Debian Wheezy as operating system. Released in the year 2009, the Dell Inspiron Mini 10v (1011) possesses an Intel Atom N270 processor, where the single-core 32-bit CPU runs at 1.6 GHz. It has a memory of 1 GB and communicates via 100 Mbit Ethernet, Bluetooth 2.1 (cf. Subsection 2.2.3), and Wi-Fi IEEE 802.11 b,g (cf. Subsection 2.2.1). [Inc09] As the Dell Inspiron Mini 10v (1011) has no built-in connectivity for a cellular network, it utilizes the Huawei E1750 surf stick connected via Universal Serial Bus (cf. Subsection C.6), providing connectivity for a 3.5 Generation cellular network.

C.3. Lenovo ThinkPad T61

The Lenovo ThinkPad T61 is a laptop from Lenovo, running a minimal Linux 64-bit version based on Debian Wheezy as operating system. Released in the year 2007, the Lenovo ThinkPad T61 possesses an Intel Core 2 Duo T7300 processor, where the dual-core 64-bit CPU runs at 2.0 GHz. It has a memory of 2 GB and communicates via 1 Gbit Ethernet, Bluetooth 2.1 (cf. Subsection 2.2.3), and Wi-Fi IEEE 802.11 a,b,g
C.4. HP Compaq 8200 Elite

Like the Dell Inspiron Mini 10v (1011), the Lenovo ThinkPad T61 has no built-in connectivity for a cellular network, why it also utilizes the Huawei E1750 surf stick connected via Universal Serial Bus (cf. Subsection C.6), providing connectivity for a 3.5 Generation cellular network.

C.4. HP Compaq 8200 Elite

The HP Compaq 8200 Elite Microtower PC is a desktop computer from Hewlett Packard, running a minimal Linux 64-bit version based on Debian Wheezy as operating system. Released in the year 2011, the HP Compaq 8200 Elite Microtower PC possesses an Intel Core i7-2600 processor, where the quad-core 64-bit CPU runs at 3.4 GHz. It has a memory of 8 GB and communicates via 1 Gbit Ethernet. [Pac11]

C.5. AWS EC2 t2.micro

The Amazon Elastic Compute Cloud (EC2) is the web service for cloud computing from Amazon (cf. Section 2.3), where an end user obtains resizable capacity of computing resources at the cloud. Due to resizing the capacity, the computing resources quickly scale up and down depending on changes in the user (computing) demands. Moreover, an end user only pays for the capacity that he or she actually used (pay-as-you-go manner). The Amazon EC2 provides different types of instances (cf. T2, M4, or C4 [Ama14]) fitting multiple use cases. This dissertation uses a t2.micro instance that possesses one virtual CPU (vCPU) running at 2.4 GHz and a memory of 1 GB. As operating system, it runs an adapted Linux 64-bit version from Amazon. [Ama14]

C.6. Huawei E1750 Surf Stick

The Huawei E1750 surf stick works in the mobile communication networks of GSM and UMTS, supporting the frequency bands for GSM of 850/900/1800/1900 MHz and for UMTS of 2100 MHz. Due to the utilization of the Qualcomm MSM7200 SoC [Qua06], the surf stick provides for GSM the improvements of GPRS (2.5 Generation) and EDGE (2.75 Generation) and for UMTS the improvement of HSPA (3.5 Generation). Thus, it downloads data with a peak data rate of 7.2 Mbit/s and uploads data with a peak data rate of 5.76 Mbit/s. [Hua17]
C. System Devices

C.7. Linksys WRT54GL Wireless Router

The Linksys WRT54GL wireless router provides a Wi-Fi AP, supporting IEEE 802.11 g,b and thus, a peak data rate of 54 Mbit/s. Beside the Wi-Fi network, the wireless router possesses a 10/100 Mbit/s WAN port and four 10/100 Mbit/s switched Local Area Network (LAN) ports based on Fast Ethernet standard IEEE 802.3u. [Lin02]

C.8. LevelOne GSW-0809 Gigabit Ethernet Switch

The LevelOne GSW-0809 Gigabit Ethernet Switch provides reliable high-performance networking with auto-learning and auto-aging of MAC addresses, where it has 8 Gigabit Ethernet ports with a wire data rate of 10/100/1000 Mbit/s. Moreover, it also supports techniques like IEEE 802.3x flow control or IEEE 802.3az energy efficiency. [Lev17]


Bibliography


Bibliography


Bibliography


Bibliography


Bibliography


Bibliography


Bibliography


Bibliography


Erklärung


__________________________  _________________________
(Ort, Datum)                  (Florian Andreas Berg)