Mobile Code Offloading
for Multiple Resources

Andreas Pamboris

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London
Abstract

Mobile devices are becoming pervasive, yet a persistent gap in hardware capabilities still separates them from desktop machines. To bridge this gap, recent research has turned to cloud-assisted execution as a way of leveraging remote resources to enhance application performance. Code-offloading systems automatically partition applications across resource-constrained devices and more powerful remote nodes to improve execution. Existing approaches, however, only focus on compute resources, ignoring memory and network limitations in mobile environments. In doing so, they prevent mobile applications from taking advantage of the larger memory and richer networking capabilities of cloud-based nodes. At the same time, they face the challenge that a large runtime overhead may offset the benefits of offloaded execution and support only applications written in managed programming languages with substantial runtime support.

In this thesis, we propose three new static code-offloading approaches that exploit all three remote resources—compute, memory and network:

1. **Compute-focused offloading** enables applications written in unmanaged programming languages with only rudimentary runtime support to benefit from remote compute resources. Using offline dynamic profiling to analyse runtime behaviour, it derives a partitioning that reduces response times by offloading compute-intensive functionality to the remote node.

2. **Memory-focused offloading** partitions application state across nodes to alleviate memory constraints and reduce offloading overheads by permanently collocating data and computation. To handle network failures, it uses a snapshot-based fault tolerance mechanism to back up state changes locally and a user-level virtual memory scheme to support execution with large state sizes after failure.

3. **Network-focused offloading** partitions mobile client applications across mobile devices and nodes at edge locations of a mobile network to minimise network traffic in radio access networks. It (i) discards unused data returned by coarse-grained API calls to Internet backend services and (ii) tunes binary object prefetching strategies to transmit only the objects that are used on the device.
This thesis is lovingly dedicated to my mother, Vasoulla Pambori, and father, George Pamboris. Their constant support, encouragement and love have sustained me throughout my life.

Την παροώσα διδακτορική διατριβή αφερώνω με ιδιαίτερη αγάπη στην μητέρα μου, Βασούλλα Παμπόρη, και τον πατέρα μου, Γιώργο Παμπόρη.
Η συνεχής υποστήριξη, ενθάρρυνση και αγάπη τους με έχει στηρίξει καθολικά την διάρκεια της ζωής μου.
Acknowledgements

This thesis presents my work for over a period of four years. I would like to thank those who have inspired and supported me in getting where I am today.

First and foremost, I would like to express my sincere gratitude to Dr. Peter Pietzuch for the opportunity to work towards a doctorate degree under his supervision. His ongoing support and invaluable guidance has led to teaching me what research is all about. I consider him a top academic professional and a great personality too. I feel extremely fortunate to have worked closely with him towards my doctorate degree and could not have wished for a better advisor.

I would also like to thank all my undergraduate and graduate professors at the University of Cyprus and University of California, San Diego, respectively, for the valuable knowledge they have passed on to me. Special thanks goes to my Masters advisor, Dr. Amin Vahdat, whose insightful guidance and abundant knowledge inspired me to pursue a doctorate degree.

As part of the Large-Scale Distributed Systems group at Imperial College London, I had the pleasure of meeting a number of exceptional researchers: Yiannis, Eva, Nicholas, Raul, Alexandros, Mateo, Paolo, Marco, Thom, Mouna, Loucas and Luo. I would like to thank them, as well as Michalis, Xi, Sei and Clint with whom I shared an office, for their enjoyable company throughout the years.

I would also like to thank all of my close friends who I consider my extended family (the list is rather long so I apologise for not naming each of you separately) for making life outside the office as fun as possible.

A special thanks goes to Dr. Alastair Donaldson and Prof. Jon Crowcroft for kindly agreeing to serve as internal and external examiners for my PhD thesis defence. Their insightful feedback has helped improve the quality of the thesis and has provided food for thought regarding future research directions.

My research was funded by Orange Labs UK (Grant Agreement P29756), and I am grateful for the support and feedback that I received from them through regular meetings.

Last but not least, I would like to thank my family, George, Vasoulla, Christina and Spyros, for their love and support throughout the years. They should know that they have greatly contributed to where I stand today.

— London, September 2013
Declaration of Originality

This thesis presents my work in the Department of Computing at Imperial College London between October 2009 and September 2013. I declare that the work presented in this thesis is my own, except where otherwise acknowledged.
Copyright Declaration

The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work.
Contents

1 Introduction 23
  1.1 Cloud-assisted execution .............................................. 25
  1.2 Problem statement .................................................. 27
  1.3 Research contributions ............................................. 28
  1.4 Thesis outline ..................................................... 32

2 Background 33
  2.1 The mobile environments ........................................... 33
    2.1.1 Mobile devices ................................................ 34
    2.1.2 Mobile platforms ............................................. 35
    2.1.3 Mobile networks .............................................. 39
  2.2 Support for remote access programmability ...................... 42
    2.2.1 Mobile middleware ........................................... 42
    2.2.2 Cyber-foraging ............................................... 47
  2.3 Automatic application partitioning ................................ 50
    2.3.1 Thin-client approaches ..................................... 51
    2.3.2 Partitioning at the granularity of application components and classes 52
    2.3.3 Partitioning at the granularity of functions and operations ...... 54
  2.4 Summary .......................................................... 59

3 Compute-Focused Offloading 61
  3.1 Overview .......................................................... 61
  3.2 Design space ...................................................... 63
    3.2.1 Static vs. dynamic code offloading ......................... 63
    3.2.2 Platforms and programming languages ...................... 64
    3.2.3 Optimisation goals ......................................... 64
    3.2.4 Partitioning granularity .................................. 64
    3.2.5 Fault tolerance ............................................. 65
  3.3 Low-level application partitioning ................................ 65
    3.3.1 Dynamic resource profiling ................................ 65
    3.3.2 Partitioning algorithm .................................... 68
  3.4 RPC-based code offloading ....................................... 70
    3.4.1 ANYWARE architecture .................................... 70
CONTENTS

3.4.2 Profiling tools ................................................. 71
3.4.3 RPC compiler .................................................. 74
3.5 Evaluation ......................................................... 78
  3.5.1 Experimental set-up ....................................... 78
  3.5.2 Performance analysis ...................................... 79
3.6 Discussion ........................................................ 81
3.7 Summary .......................................................... 81

4 Memory-Focused Offloading ........................................ 83
  4.1 Overview ......................................................... 83
  4.2 Design space .................................................... 85
    4.2.1 Optimisation goals ....................................... 86
    4.2.2 Application state migration vs. partitioning .......... 86
    4.2.3 Partitioning granularity .................................. 86
    4.2.4 Fault tolerance ........................................... 87
  4.3 Application state partitioning ................................ 87
    4.3.1 Dynamic resource profiling ............................... 87
    4.3.2 Partitioning algorithm .................................. 89
    4.3.3 Application transformation ............................... 91
  4.4 Snapshot-based fault tolerance ................................ 94
    4.4.1 State snapshots .......................................... 94
    4.4.2 Snapshotting mechanisms ................................. 95
    4.4.3 Incremental state snapshots ............................. 97
    4.4.4 Virtual memory scheme .................................. 98
    4.4.5 Related work on checkpointing and program rollback. 98
  4.5 Evaluation ....................................................... 99
    4.5.1 Experimental set-up ..................................... 99
    4.5.2 Application response time ............................... 100
    4.5.3 Memory consumption ...................................... 104
    4.5.4 Post-failure performance ................................. 105
  4.6 Discussion ...................................................... 106
  4.7 Summary ......................................................... 106

5 Network-Focused Offloading ......................................... 109
  5.1 Overview ......................................................... 109
  5.2 Design space .................................................... 112
  5.3 Mobile client applications .................................... 113
    5.3.1 Mobile client communication ............................ 114
    5.3.2 Mobile client application architecture ................ 116
  5.4 EdgeReduce design ............................................. 118
    5.4.1 Filtering of backend API data ........................... 119
    5.4.2 Replacing data objects with futures ................... 119
5.5  \textsc{EdgeReduce} implementation ........................................ 120
  5.5.1  Source-level program analysis ....................................... 120
  5.5.2  Source code transformation .......................................... 121
5.6  Evaluation ........................................................................ 124
  5.6.1  Experimental set-up .......................................................... 125
  5.6.2  Network bandwidth usage .................................................. 125
  5.6.3  Application performance ................................................... 127
5.7  Discussion ....................................................................... 130
5.8  Summary ....................................................................... 130

6  Conclusion ........................................................................ 131
  6.1  Thesis summary ................................................................. 132
  6.2  Future work ................................................................... 134

Bibliography ........................................................................ 137
# List of Figures

1.1 Example of how cloud-assisted execution can benefit a mobile chess client. 26
2.1 Comparison of classes of devices in terms of CPU speed and RAM availability. 35
2.2 Structure of a mobile network. 39
2.3 Taxonomy of code offloading approaches. 50
3.1 Example of a CPU consumption graph. 67
3.2 Architecture of the ANYWARE code offloading system. 70
3.3 Example of a partitioned function. 77
3.4 Response times of unpartitioned and ANYWARE-partitioned execution for different CPU thresholds. 80
3.5 CPU utilisation over time for unpartitioned and ANYWARE-partitioned execution. 81
4.1 Architecture of the CLOUDSPLIT code offloading system. 88
4.2 Memory consumption per application class using DTrace. 90
4.3 Example of a partitioned class. 92
4.4 Allocator class for managing objects life-cycle. 93
4.5 Synchronous snapshotting strategy. 95
4.6 Asynchronous snapshotting strategy. 97
4.7 Response time breakdown for the Latrunculi game application. 101
4.8 Response time breakdown for the iSpreadsheet application (simple workflow). 102
4.9 Response time breakdown for the iSpreadsheet application (complex workflow). 104
4.10 Memory consumption over time. 104
4.11 Response times for different state sizes. 105
4.12 Post-failure overhead for iSpreadsheet. 105
5.1 Data returned by the Twitter backend service API and its subsequent use in a Twitter client application. 115
5.2 General model-view-controller design pattern. 116
5.3 Model-view-controller design pattern used in mobile client applications. 117
5.4 Overview of the operation of EDGEREDUCE. 118
5.5 Optimisation technique for replacing binary data objects with futures. 119
5.6 Example of coalescing multiple getter calls. 122
5.7 Use of transient Data Object to reduce remote calls. 123
5.8 Bandwidth usage of unpartitioned and EdgeReduce-partitioned execution. 126
5.9 Response times of unpartitioned and EdgeReduce-partitioned execution. 127
5.10 On-demand object retrieval time for the Groupon client application. 128
5.11 RAN transmission times for varying amounts of backend data traffic (Groupon client). 129
## List of Tables

2.1 Characterisation of popular smartphones, tablets, laptops and desktops. . . . 34

3.1 AnyWare vs. representative code offloading systems. . . . . . . . . . . . . . 63

4.1 CLOUDSPLIT vs. representative code offloading systems. . . . . . . . . . . . . 86

5.1 EDGEREDUCE vs. representative code offloading systems. . . . . . . . . . . . . 112
List of Algorithms

1. Partition application – *Compute-focused approach* .......................... 69
2. Profile CPU usage ................................................................. 73
3. Profile main memory usage ....................................................... 73
4. Split application using RPC ....................................................... 76
5. Partition application – *Memory-focused approach* ......................... 91
Chapter 1

Introduction

Stationary personal computers (PCs) are gradually giving way to smaller, yet still powerful, mobile computing devices, with all evidence pointing to this trend continuing for the foreseeable future [Sto13]. Today, mobile devices have taken many forms, including smartphones, tablets, e-readers, netbooks and notebooks, each with different characteristics and capabilities. This transition from PCs to mobile devices has led to the development of a plethora of third party mobile applications to satisfy the ever-growing demands of consumers. Millions of mobile applications nowadays, each tailored to the specific requirements of a given mobile device, are made available in dedicated application marketplaces such as Apple’s App Store [App13a], Google’s Google Play [Goo13d] and Microsoft’s Windows Phone Store [Mic13].

However, the very features of mobile devices that make them attractive to consumers, such as device portability, easy connectivity and mobile data processing and storage, often have dependencies and restrict each other. Consider the trade-off between portability (as indicated by physical shape and size) and power (as indicated by resource availability) as a representative example. The smaller the device the less powerful it is in terms of compute, memory and networking capabilities—form factor constraints being the limiting factor. Nevertheless, smaller sizes typically imply greater portability. Such trade-offs are what primarily have resulted in different classes of mobile devices in an ongoing pursuit for satisfying all consumer expectations. Despite mobile technologies rapidly maturing, the fact of the matter remains that a persistent gap in hardware resources separates different classes of devices, which explains the varying potency of different versions of applications developed with specific target hosts in mind.

As a result, mobile marketplaces today are overwhelmed by a blinding variety of popular applications, which consume increasing amounts of computation, memory and communication resources on mobile devices. For example, gaming applications featuring complex AI functionality and photo/video editing software require heavy processing capabilities; augmented reality applications (e.g. the upcoming Google Glass system [Goo13c]) process large amounts of real-time sensor data in memory; and mobile client applications (e.g. social networking
clients) exchange large amounts of data with Internet backend services. Yet smartphone applications in these domains remain noticeably inferior to their desktop counterparts, both in terms of functionality and performance.

These application limitations are caused by many factors, one of them being the fact that computational demands are often disproportionate to devices’ compute capabilities. For many classes of mobile applications, available memory also becomes a limiting factor [Cra13]. For example, on Apple’s iPhone 4S smartphone, applications are left with just 213 MB of usable main memory and those that exhaust the available memory are automatically terminated by the operating system. Furthermore, the growth of mobile data, recently described by mobile network operators as a “data tsunami”, threatens to overwhelm 3G networks [ENC10]. This means that the network becomes a bottleneck for applications that rely on frequent costly data exchanges with backend services. Particularly in densely populated urban areas, this causes problems for consumers in the way of dropped calls and slow network connectivity. Even the next generation of 4G or LTE networks are unlikely to meet the exponentially growing demand for capacity [CNR10]. Besides impacting the performance of applications, the increase in mobile data entails significant data and operational costs to mobile users and network operators.

While application performance and functionality are limited by the hardware resources of mobile devices, the rapid growth of data centres and cloud computing [AFG+09] present new opportunities for overcoming these limitations. Hardware resources and applications are now offered as services over the Internet by many providers. Massively scalable and cost effective resources can be accessed on demand in order to achieve efficiency at a larger scale, higher reliability and most importantly the illusion of infinite hardware resources, thus removing infrastructure as a barrier to the scaling of applications.

The advantages of cloud computing are now exposed to mobile users. Mobile devices are able to connect to remote infrastructures, maintain these connections over time and reliably exchange data with the cloud, i.e. resources available over a network ranging from a single machine to a large-scale data centre. This has opened up new avenues for reducing application response times and energy expended on mobile devices by intelligently distributing computation in what is referred to as cloud-assisted execution. Cloud-based resources are leveraged at runtime to enhance the computational capabilities of resource-constrained mobile devices. As an example, code offloading systems [CBC+10, CIM+11, KAH+12, GJM+12] automatically partition applications across local (mobile devices) and remote nodes (e.g. virtualised base stations [IBM13]). These enable applications to offload compute-intensive functionality to more powerful compute resources for increased performance. Conceptually, such systems transform single-machine execution into distributed execution, which accounts for the network connection to the remote node, the processing capabilities of both local and remote nodes and the application’s runtime behaviour.

In this thesis, we argue that existing code offloading approaches are inadequate for the current mobile setting. They maintain a narrow view of the types of cloud resources that can
be exploited to augment mobile applications, essentially treating remote nodes as external compute resources. As a result, they overlook a set of additional goals that can be met such as minimising memory and network consumption on mobile devices. At the same time, they incur significant runtime overheads, which limit the performance gains achieved by code offloading. This is due to the fact that they (1) treat application state in a distributed execution setting as a monolithic entity that needs to be migrated along with every offloaded task and (2) make runtime offloading decisions, which requires monitoring resource consumption and changes in the execution environment during execution. In doing so, they also rely on certain properties of the programming language such as type safety and support from the underlying runtime system. This restricts applicability to only applications written in managed programming languages. However, with the looming wave of low-end devices, such as Google’s Chromecast media streaming adapter [Goo13b], mini versions of top-of-the-line smartphones and tablets [CNN13, Ter13] and computerised smart watches [Mas13b], providing support for unmanaged applications on smaller embedded computing devices appears to be key to the wide adoption of code offloading practises in the future.

In the remainder of this chapter, we provide an overview of cloud-assisted execution and how it is realised in code offloading approaches proposed to date. We also discuss the limitations of existing approaches in more detail and summarise the contributions of the thesis.

1.1 Cloud-assisted execution

Cloud-assisted execution promises to overcome performance limitations of applications running on resource-constrained devices by allowing them to offload execution to more powerful remote nodes. While the idea of exploiting remote resources opportunistically to augment smartphone capabilities was first introduced in the early 2000s (cyber-foraging [BFS+02]), only recently has it been realised in automatic offloading systems such as MAUI [CBC+10], CloneCloud [CIM+11], ThinkAir [KAH+12] and COMET [GJM+12]. Such systems minimise application response times and/or energy consumption on the mobile device by partitioning applications across a local and a remote node, with compute-intensive functionality seamlessly running at the remote side.

In Figure 1.1, we show how code offloading can benefit a mobile chess client [Ble08] that runs on a smartphone, tablet or laptop with access to a more powerful remote node. Such a game application, featuring a complex game AI that is used to decide the opponent’s next move, poses as a representative application for this setting. As expected, the game AI places extreme demands on rather limited compute resources and energy reserves available on a mobile device. The task of the code offloading system is to identify the application’s compute-intensive AI functionality (function game_AI()) and offload it seamlessly to a more powerful remote infrastructure for faster and more energy efficient execution. This renders the application more interactive, while also prolonging the battery life of the mobile device.

At a high level, current code offloading systems decide on a set of offloading and reintegration
points for a given application: code with its associated application state is migrated to a remote node for execution, and eventually migrated back to the mobile device. The majority of current code offloading systems are dynamic [CBC+10, KAH+12, GJM+12], in that they defer the decision-making to execution time. This requires continuous monitoring of the resource consumption on the mobile device and the properties of the execution environment to adapt offloading policies in response to runtime changes (e.g. in network performance, energy availability etc.). Static approaches [CIM+11], on the other hand, precompute a set of code partitionings for a given application and execution environment, and decide on the partitioning to be used upon deployment. While this has the disadvantage of not being able to react to changes at runtime, it avoids the overhead associated with making offloading decisions each time execution reaches e.g. a function call, as well as the overhead of continuously monitoring resource consumption and environment changes. Therefore, static approaches are more suitable for applications running on low-end devices with severe CPU constraints and minimal runtime support. Any additional overhead incurred by code offloading is critical to the performance of these applications.

What is common to all current code offloading systems is that they rely on certain properties of the programming language and the underlying runtime system to carry out code offloading. These primarily consist of (1) programming language support for strong type-safety to capture the portion of the application state that needs to be migrated with every offloaded function call (i.e. the application state that could be accessed remotely during the remote call) and (2) runtime support for code mobility to migrate running programs from one node
to another. Therefore, existing approaches target applications written in strongly-typed managed programming languages such as Java and C#, which are backed up by a rich runtime system—with support for code mobility—on mobile platforms such as Android and Windows Mobile. This precludes their applicability to unmanaged applications such as Objective-C applications on the iOS platform.

1.2 Problem statement

In this section, we identify shortcomings in the way current code offloading approaches address the challenges of cloud-assisted execution. These shortcomings relate to their current narrow view of the types of cloud resources considered for augmenting mobile device capabilities, as well as their inefficient techniques for managing application state and offloading computation at runtime. We discuss the two major drawbacks of current approaches in more detail below:

1. No support for memory and network resources

Existing code offloading approaches only leverage remote compute resources for faster and more energy efficient application execution. With the advent of cloud computing, however, we are also presented with opportunities to overcome memory- and network-related limitations by allowing mobile applications to access the larger memory of remote nodes, as well as their high-bandwidth network connectivity to the outside world. Limitations pertaining to available device memory [Cra13] and over-utilised mobile networks [CNR10] undoubtedly pose barriers to improving mobile application performance and functionality.

A solution that also exploits the cloud’s memory and network resources could thus (1) enable memory-intensive application workloads to exceed a mobile device’s memory and (2) help minimise mobile network traffic to benefit both consumers (by reducing data charges) and network operators (by relieving contention in mobile networks to reduce operational costs and retain customers).

2. High runtime overheads and limited applicability

The way current approaches perform code offloading and manage application state across local and remote nodes incurs significant runtime overhead. In many cases, the overhead is enough to offset the gains achieved by cloud-assisted execution completely. We identify the following two main sources of runtime overhead in current state-of-the-art code offloading approaches:

(a) Overhead of migrating application state with every offloading operation

When offloading computation to a remote node, current approaches capture, serialise, transmit over the network and deserialise the portion of the application state that may be accessed during remote execution twice—once per direction of communication between the local and remote node. Furthermore, any changes to
the application state need to be merged back into the local node’s address space upon completion of the offloaded task. Each of these actions implies a runtime overhead.

(b) Overhead of monitoring resource consumption and the execution environment when making runtime offloading decisions

This applies to dynamic offloading solutions, which constitute the majority of approaches proposed to date. Dynamic solutions consume additional compute cycles for making offloading decisions every time execution reaches a possible offloading point (usually a function call) based on the profiling information collected at runtime.

Besides the large runtime overheads they incur, existing approaches rely heavily on runtime support for code mobility and support only applications written in strongly-typed programming languages. While these requirements are met by e.g. Java’s object model on the Android platform, the lack of such support for unmanaged programming languages such as C or Objective-C renders existing approaches inapplicable to applications for platforms such as the iOS platform.

1.3 Research contributions

This thesis identifies opportunities to address the limitations of current code offloading approaches. Driven by the observation that both application performance and functionality are affected by multiple resources consumed by applications, we approach the challenges of cloud-assisted execution from a different angle. We consider all three main resources—compute, memory and network—for code offloading. Besides reducing application response times or energy consumption, we also focus on empowering applications by lifting device memory constraints and minimising mobile network traffic to improve user experience on a number of new fronts.

At the same time, we provide solutions to current limitations of code offloading approaches that relate to their high runtime overheads—limiting performance gains—and the assumptions made regarding programming languages and runtime support—restricting applicability. Unlike prior approaches, we focus on code offloading for applications written in unmanaged programming languages, namely C and Objective-C, with only rudimentary runtime support. We adopt a static partitioning model. While its offloading policies cannot be adapted at runtime, it avoids the overheads associated with dynamic approaches. This choice is consistent with our aforementioned goal.

We introduce three new static code offloading approaches, each focused around exploiting compute, memory and network cloud-based resources, respectively:
Compute-focused offloading

We propose an approach that statically partitions applications across a local and a remote node to allow them to offload compute-intensive functionality. The application partitioning is decided based on fine-grained *offline dynamic profiling* of its runtime behaviour. We focus on applying code offloading to applications written in unmanaged programming languages, with no particular code structure other than functions. The C programming language is a representative choice of unmanaged programming languages.

The contributions of this work are:

1. *Offline dynamic profiling:* We describe an approach for analysing the runtime behaviour of C applications using dynamic profiling. This is done offline and requires no modifications to the application itself. We employ low-overhead dynamic code instrumentation techniques to collect fine-grained profiling information about an application’s resource consumption. This is used to decide an application partitioning that minimises response time. Profiling is done while the application executes a realistic workload that captures its average runtime behaviour in terms of CPU and memory usage.

2. *Partitioning of applications written in unmanaged programming languages:* We propose a methodology for deciding a partitioning for applications written in the C programming language based on profiling information of their runtime behaviour. The split is done along boundaries that are inferred by identifying each function’s contribution to the overall CPU consumption of the application, as well as function call and shared memory dependencies.

3. *Code offloading using remote procedure calls:* We describe how to realise a desired partitioning by rewriting the source code of a C application to change local into remote function calls, while managing application state accordingly. At runtime, the application transparently executes remote procedure calls (RPCs) [BN84] between the local and remote nodes to transfer control of execution from one to the other.

We demonstrate this approach with AnyWare, a system that automatically partitions C applications to execute compute-intensive functionality on more powerful remote nodes. We evaluate AnyWare with a complex open-source C application and show that it can deliver a speedup of up to $1.32 \times$ in execution time in an environment with severe CPU constraints.

Memory-focused offloading

We propose a static partitioning approach that allows applications to benefit from the remote node’s larger memory, in addition to its faster compute resources for lowering application response times. This is accomplished by means of *partitioning application state* across the local and remote nodes, contrary to typical *state migration* approaches that always maintain a local copy on the device—migrated back and forth with every offloading operation. This has
two main advantages: (1) it allows applications to consume memory beyond the capacity of the local node; and (2) it reduces the amount of data that offloaded calls transmit repeatedly because they can reuse already available state on the remote node. This eliminates the runtime overheads associated with frequent state migration. A new challenge, however, is that execution must continue after access to the remote state was lost due to failure. This requires a mechanism for recovering remote state on the local node after failure.

The contributions of this work are:

1. **Application state partitioning**: Based on offline dynamic profiling, we propose an optimisation-based partitioning algorithm that splits application state between two nodes. We describe how this is realised for applications written in Objective-C, an unmanaged object-oriented language. Each application object is placed permanently either on the local or the remote node. Access to remote objects is supported transparently via proxy objects, which relay method invocations using RPCs.

2. **Snapshot-based fault tolerance**: We introduce a new fault tolerance mechanism based on snapshots, which allows the local node to recover missing application state after losing network connectivity to the remote node. Our approach takes periodic, consistent snapshots of changes to the local and remote application states and stores them in the local node’s flash memory. Depending on the amount of application state modified with each offloaded call, the snapshotting overhead may have an impact on the performance gains achieved by code offloading. To this end, we propose two separate strategies to cover both lightweight and heavyweight applications in terms of memory usage, each with different tradeoffs. In a *synchronous* strategy, a snapshot is taken after each offloaded call, which allows failed calls to re-execute locally immediately after failure. An *asynchronous* strategy permits more sporadic snapshots, which are transmitted asynchronously to the local node. After failure, the application state is rolled back to the last complete snapshot and local execution resumes from an earlier point in time. This is all achieved without modifications to the iOS platform or Objective-C.

3. **User-level virtual memory**: To support application state sizes that are larger than the local node’s memory capacity after failure, we propose a simple virtual memory scheme under iOS at user level, again without modifications to iOS or Objective-C. Objects from remote state snapshots remain stored in flash memory and are loaded into main memory on demand, when accessed by the application.

To illustrate our approach, we have developed CloudSplit, a code offloading system for Objective-C applications that realises the above techniques. We evaluate our prototype implementation with two real-world iOS applications to show that it supports workloads with large memory footprints and outperforms conventional state migration approaches by a factor of $15 \times$ in application response times.
Network-focused offloading

We propose a code offloading approach that reduces the network traffic associated with mobile client applications such as social networking, photo sharing and e-commerce clients. Such applications rely on frequent interactions with Internet backend services, which often entail unnecessary data transfers due to the coarse granularity of backend API calls and aggressive prefetching strategies used. Our approach offloads the application logic that is responsible for processing API response data to remote nodes at edge locations of a mobile network, thus allowing for unused data to be discarded before traversing the radio access network (RAN). This benefits both mobile users—by reducing increased data usage charges—and network operators—by reducing network contention in limited radio access networks.

The contributions of this work are:

1. Filtering of backend API data: Using static code analysis, we identify application components that interact with backend services and process the response data sent to the mobile device. We derive a partitioning that places these components on a remote node at the network edge. By parsing the data returned by backend API calls and storing the results as application data objects at the network edge, we are able to discard unused data transmitted to the mobile device and eliminate the overhead of inefficient encoding formats used.

To reduce the overhead of multiple remote calls between the local and remote nodes, we employ two optimisation techniques: (i) coalescing multiple remote calls during the parsing of backend responses due to repeated transmissions of object fields; and (ii) creating transient data objects on the remote node to reduce the number of remote calls that initialise data objects, which are placed on the local node. These transient objects are then transferred to the local node in a single remote call.

2. Replacing binary objects with futures: A significant portion of the data retrieved by mobile client applications consists of large binary data objects such as images. A common characteristic of such applications is that they employ aggressive prefetching strategies, which result in transfers of large amounts of binary objects with only a fraction of them being used subsequently.

Our approach intercepts such transfers remotely, before they traverse the RAN to reach the device. Binary objects remain stored at the remote node and are replaced by smaller futures, which are returned to the local node. The actual binary objects are only retrieved automatically when about to be used by the client application using the corresponding futures as reference. This process is transparent to the user.

We have realised our network-focused approach with EDGEREDUCE, a code offloading system for Objective-C applications that meets the above goals without modifications to the iOS platform or Objective-C. We evaluate EDGEREDUCE with three real-world iOS applications and show that overall reductions of up to 8.2× in mobile network traffic are possible.
1.4 Thesis outline

The remainder of this thesis is organised as follows:

Chapter 2 discusses related work. It first provides an overview of the current mobile setting to identify the limitations in application performance and the corresponding opportunities offered by more powerful remote resources during execution. This includes an investigation of the hardware capabilities of different mobile device classes, the application support provided by current mobile platforms and the limitations of mobile networks. It then discusses previous approaches that provide support for developing new distributed mobile applications: mobile middleware technologies that hide the complexity of distributed application development; and cyber-foraging approaches that propose new application architectures to allow for opportunistic exploitation of remote resources in the vicinity of mobile devices. The chapter finishes with a discussion of more recent code offloading approaches that automatically partition mobile applications without requiring major developer intervention. The most representative code offloading approaches to date are used to compare against our compute-, memory- and network-focused offloading approaches in the remainder of the thesis.

Chapter 3 presents our compute-focused offloading approach, which is realised by the AnyWare system. It describes the sequence of steps to automatically transform unmanaged C applications for cloud-assisted execution. These comprise our offline dynamic profiling, application partitioning and source-code rewriting techniques to realise a partitioning. The chapter finishes with an evaluation of our prototype implementation of AnyWare using a real-world open-source C application.

Chapter 4 presents our memory-focused offloading approach, which is realised by the CloudSplit system. It describes how we handle the profiling and partitioning of Objective-C applications on the basis of application state partitioning, our snapshot-based fault tolerant mechanisms to handle network failures and the user-level virtual memory scheme to cope with large state sizes after failure. This approach is evaluated against state-of-the-art state migration approaches using two real-world iOS applications.

Chapter 5 describes our network-focused offloading approach, which is realised by the EdgeReduce system. It discusses the potential for data traffic reduction based on unused data returned to mobile client applications. It then describes the design of EdgeReduce and how a partitioning is implemented using the source code of client applications written in Objective-C. It focuses on the two optimisations used to overcome the additional overhead of the increased number of remote calls, as well as the methodology for tuning aggressive prefetching strategies. The chapter concludes with an evaluation of EdgeReduce using three real-world iOS applications.

Finally, Chapter 6 concludes the thesis by summarising the work presented and outlining future research directions.
Chapter 2

Background

The fact that mobile devices are yet to meet the computational ability of desktop machines is well perceived by consumers in the way of degraded mobile application performance and functionality. To understand the limitations that pose barriers to the proliferation of more sophisticated mobile applications, this chapter first provides an overview of current mobile environments in Section 2.1. This includes an overview of the capabilities of some of the most popular mobile devices, the application support provided by two dominant mobile platforms and the architecture of today's mobile networks. We also discuss previous work that focuses on the high latencies and limited bandwidth experienced in current mobile networks. The goal is to describe the challenges that code offloading and mobile client applications face.

In Section 2.2, we discuss previous work on the development of future mobile applications with cloud deployment in mind. We discuss middleware solutions that have been proposed in the past to ease the distribution of mobile applications, as well as cyber-foraging approaches that realise the vision of opportunistic resource utilisation.

Finally, Section 2.3 discusses the more recent trends towards automatic code offloading approaches. Contrary to the above, these techniques aim at partitioning mobile applications for cloud-assisted execution, with minimal to no developer involvement. The work presented in this thesis mainly falls into this research area. Therefore, a comparison of our work with the most representative systems described in this section will be given in subsequent chapters.

2.1 The mobile environments

We investigate today's mobile environments to realise application performance limitations that are intrinsic to them and identify opportunities to overcome these limitations. This investigation spans three different facets of mobile environments: the hardware capabilities of current mobile devices; the support provided by popular mobile platforms to mobile applications; and the limitations of current mobile networks, along with proposed solutions for overcoming these limitations.
Background

<table>
<thead>
<tr>
<th>Smartphones [PCm13c]</th>
<th>CPU speed</th>
<th>RAM</th>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple iPhone 5 [GSM12c]</td>
<td>1.3 GHz</td>
<td>1 GB</td>
<td>3G/4G/WiFi</td>
</tr>
<tr>
<td>Samsung Galaxy S4 [GSM13e]</td>
<td>1.9 GHz</td>
<td>2 GB</td>
<td>3G/4G/WiFi</td>
</tr>
<tr>
<td>HTC One [GSM13b]</td>
<td>1.7 GHz</td>
<td>2 GB</td>
<td>3G/4G/WiFi</td>
</tr>
<tr>
<td>Google Nexus 4 [GSM13c]</td>
<td>1.5 GHz</td>
<td>2 GB</td>
<td>3G/WiFi</td>
</tr>
<tr>
<td>Nokia Lumia 928 [GSM13d]</td>
<td>1.5 GHz</td>
<td>1 GB</td>
<td>3G/4G/WiFi</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tablets [PCm13d]</th>
<th>CPU speed</th>
<th>RAM</th>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple iPad 4th Generation [GSM12b]</td>
<td>1.4 GHz</td>
<td>1 GB</td>
<td>3G/4G/WiFi</td>
</tr>
<tr>
<td>Google Nexus 7 [GSM13a]</td>
<td>1.5 GHz</td>
<td>2 GB</td>
<td>3G/4G/WiFi</td>
</tr>
<tr>
<td>Samsung Galaxy Note 10.1 [GSM12e]</td>
<td>1.4 GHz</td>
<td>2 GB</td>
<td>3G/WiFi</td>
</tr>
<tr>
<td>Microsoft Surface [GSM120]</td>
<td>1.3 GHz</td>
<td>2 GB</td>
<td>WiFi</td>
</tr>
<tr>
<td>Amazon Kindle Fire HD 8.9 LTE [GSM12a]</td>
<td>1.3 GHz</td>
<td>1 GB</td>
<td>3G/4G/WiFi</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Laptops [PCm13b]</th>
<th>CPU speed</th>
<th>RAM</th>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple MacBook Pro</td>
<td>2.5 GHz</td>
<td>8 GB</td>
<td>WiFi/Wired</td>
</tr>
<tr>
<td>Lenovo G580</td>
<td>2.5 GHz</td>
<td>4 GB</td>
<td>WiFi/Wired</td>
</tr>
<tr>
<td>Razer Blade</td>
<td>2.2 GHz</td>
<td>8 GB</td>
<td>WiFi/Wired</td>
</tr>
<tr>
<td>Dell Inspiron</td>
<td>1.9 GHz</td>
<td>4 GB</td>
<td>WiFi/Wired</td>
</tr>
<tr>
<td>Microsoft Surface Windows 8 Pro</td>
<td>1.7 GHz</td>
<td>4 GB</td>
<td>WiFi/Wired</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Desktops [PCm13a]</th>
<th>CPU speed</th>
<th>RAM</th>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple iMac 27</td>
<td>3.4 GHz</td>
<td>8 GB</td>
<td>Wired</td>
</tr>
<tr>
<td>Dell XPS 27 Touch</td>
<td>3.1 GHz</td>
<td>8 GB</td>
<td>Wired</td>
</tr>
<tr>
<td>Falcon Northwest Fragbox</td>
<td>3.5 GHz</td>
<td>32 GB</td>
<td>Wired</td>
</tr>
<tr>
<td>HP Z420</td>
<td>3.6 GHz</td>
<td>16 GB</td>
<td>Wired</td>
</tr>
<tr>
<td>Dell Optiplex 9010 AIO</td>
<td>3.1 GHz</td>
<td>8 GB</td>
<td>Wired</td>
</tr>
</tbody>
</table>

Table 2.1: Characterisation of popular smartphones, tablets, laptops and desktops.

2.1.1 Mobile devices

We first give an overview of the capabilities of three widely used mobile device classes: smartphones, tablets and laptops. Table 2.1 shows the hardware specifications—in terms of CPU speed, available RAM and network capabilities—of five representative devices for each of these classes [PCm13c, PCm13d, PCm13b]. We also provide the corresponding specifications of five representative desktop machines [PCm13a] for comparison.

For any computing device, considerations of size, battery life, heat dissipation and other form factor elements have implications on hardware capabilities such as processor speed and memory size. Figure 2.1 shows the average CPU speed and available RAM per device class based on the representative devices from Table 2.1. Figure 2.1(a) demonstrates that although smartphone and tablet devices have similar processor speeds, laptops and desktops exhibit much faster CPUs—by a factor of approximately $1.5 \times$ and $2.3 \times$, respectively. A similar trend is observed for main memory in each device class. Figure 2.1(b) shows that laptop and desktop machines have $3.5 \times$ and $9 \times$ more main memory available for applications, respectively, compared to smartphones and tablets.

Regarding network capabilities, smartphones and tablet devices can only use wireless networking technologies. With just a few exceptions, today’s hand-held networked devices are all 3G-, 4G- and WiFi-enabled. As reported by Sprint, a major US network provider, 3G and 4G mobile networks offer speeds of approximately 0.5-1.5 and 3-6 Mbps on average, respectively [Spr13]. Better performance is achieved over WiFi networks: using the IEEE 802.11b standard, a maximum theoretical bandwidth of 11 Mbps is supported; using the IEEE 802.11a and 802.11g standards, the theoretically attainable bandwidth goes up to 54 Mbps. WiFi performance, however, is distance sensitive, meaning that performance degrades as devices move farther away from WiFi access points. Contrary to smartphones and tablet devices, lap-
Hardware limitations pertaining to mobility constraints have an impact on the relative performance and functionality of applications targeting different device classes. This is evidenced by the fact that mobile—but not laptop or desktop—application marketplaces are filled with small-footprint single-use programs [New12]. This explains why users of Apple’s laptop and desktop machines are willing to pay more for software than iPhone users—among the top 100 applications in corresponding marketplaces, the average selling price for laptop and desktop applications is $11 \times$ higher than that of mobile applications. Furthermore, on the iPhone, simple games are the dominant category of applications, whereas on Apple’s equivalent laptop and desktop machines, more complex utility and productivity applications are most popular. All of this suggests that applications developed for laptops and desktops comprise more capable fully-featured software for which consumers are willing to spend more.

Discussion

In conclusion, mobile hardware achieves low performance relative to server or desktop hardware. While mobile technologies continue to evolve over time, device classes differ in terms of CPU, memory and networking capabilities. From a consumer’s viewpoint, a mobile device can never be too small, light or battery-efficient. By focusing technology advances along these lines, the availability of compute, memory and network resources on mobile devices will always remain a compromise. To this end, offloading solutions can help by enabling resource-constrained devices to leverage the richer hardware resources of remote machines without compromising the form factor.

2.1.2 Mobile platforms

Android [Goo13a] and iOS [App13b] are currently the most popular mobile platforms among consumers, with a combined market share that exceeds the ninetieth percentile of smartphone
Background

users [Mas13a]. We describe these two platforms in terms of the support that they provide to mobile applications. First, we briefly discuss the programming languages that are supported on each platform. We then focus on different properties of languages and runtime systems on each platform, which existing offloading solutions have to leverage. These properties include type safety, code mobility, object serialisation and programming reflection.

Application development

The Android platform supports various devices with different hardware capabilities, sizes and features. Android applications are usually developed using the Java programming language, though C and C++ are also supported. Java applications on Android devices are compiled to byte-code and then converted to the Dalvik Executable (.dex) format\(^1\) to be executed on the Dalvik virtual machine, which is Android’s application-level, register-based variation of Java’s VM. The Android source code and SDKs are publicly available.

In contrast, iOS is only available on a small number of devices, specifically the different generations of iPhone smartphones and iPad tablets. The programming language of choice for iOS development is Objective-C. This is a language that is based on the C syntax, but with extensions for object-oriented concepts such as classes, inheritance, Smalltalk\(^2\)-style messaging and dynamic typing.

Type safety

Type safety refers to the property that guarantees that any operation performed by an application is executed on values of the appropriate data type. This can be enforced (i) statically, by delegating the task to the compiler, which uses static analysis to identify code that may lead to type errors, e.g. treating an integer as a floating-point value or calling non-existent methods of objects; (ii) dynamically, by associating type information with values at runtime, which are consulted on demand to detect imminent errors and result in runtime exceptions; or (iii) using a combination of both static and dynamic type-checking approaches.

On the Android platform, Java enforces strong type safety by employing a static type checker at compile time. This, however, is complemented by a byte-code verifier in the VM runtime because Java allows type casts and other constructs that cannot be fully checked statically, e.g. Java’s covariant rule for sub-typing arrays [MBL97]. In contrast, Objective-C applications are not strongly typed. A weak form of dynamic type checking is provided by the iOS runtime to yield appropriate exceptions when detecting type errors during execution.

---

\(^1\)The .dex files installed on Android devices are the Dalvik-compatible counterparts of JVM’s .class files.

\(^2\)Smalltalk is one of the first object-oriented programming languages.
Code mobility

Code mobility can be defined as the ability to change the bindings between code fragments and the actual location where they are executed dynamically [CPV97]. It requires both programming language and runtime support.

The two extremes of the continuum representing degrees of code mobility are weak and strong code mobility [FPV98]. The former allows an executing unit to be bound dynamically to code coming from a different node. However, no execution state (e.g. the program stack and register values) is transferred across the network. The latter requires suspending execution on the originating site, moving execution state to a destination site and resuming execution there.

On Android, Java and its VM runtime provide inherent support for weak code mobility, by allowing classes to be loaded dynamically from remote locations. A special class loader is used to load a class from a Dalvik executable file, link it to the current execution environment and initialise its class data. Since data and code move separately in Java, the VM offers additional runtime facilities such as Java’s object serialisation (described next in more detail), which allows a graph of objects to be transformed into a binary stream that can be loaded dynamically from a remote site.

In Section 2.3.3, we show how existing code offloading solutions (namely MAUI [CBC+10] and ThinkAir [KAH+12]) rely on this type of weak code mobility support to offload code. We also discuss approaches that leverage the language constructs and runtime facilities described above to provide support for strong code mobility (e.g. as used by CloneCloud [CIM+11] and COMET [GJM+12]).

Code mobility is not supported in Objective-C applications on the iOS platform. A simplified interface for delivering software does exist in the form of bundles, which are standardised hierarchical structures that consist of executable code and any resources used by it [App10a]. These bundles, however, are not loadable at runtime (as in the case of Mac OS X) to extend the behaviour of an application dynamically.

Object serialisation

Object serialization refers to the process of encoding object state into a format that can be stored or transmitted over a network. This state representation can be used to reconstruct application objects in the future either in the same or another environment. Providing support for object serialisation is often complicated by the fact that complex objects make extensive use of references, i.e. pointers to other application objects, which also need to be serialised in a recursive manner. This recreates semantically identical clones of graphs of inter-linked objects at runtime.

On the Android platform, Java supports automatic object serialization, which is handled internally unless the developer explicitly overrides the serialization process. This simply
Background

requires that an object is marked as implementing the `java.io.Serializable` interface. A standard encoding translates object fields into a byte stream. Non-transient and non-static objects that are referenced by an object to be serialised are also recursively encoded into the stream. The process fails only if an object in the complete graph of non-transient object references is not marked as serialisable.

On the iOS platform, the process of serialising objects requires manual effort. Objective-C provides only the basic primitives for serialising an object graph at runtime. Developers need to ensure that their application classes comply with the `NSCoding` protocol, which declares the two methods that a class must implement for encoding and decoding objects. These methods need to be explicitly implemented by the developer, ensuring that an object being encoded or decoded is also responsible for encoding or decoding its instance variables.

Programming reflection

Programming reflection is the ability of an application to examine and/or manipulate its object structures and behaviour during execution. This involves inspecting or modifying object values and methods, as well as any meta-data associated with it.

On the Android platform, Java exposes the features of a class at runtime: applications can enumerate and access class fields and methods using dedicated runtime API constructs [Ora13]. It also allows for performing dynamic operations, which are not hard-coded but determined at runtime. This provides support for modifying an application’s default behaviour on the fly. Similarly, iOS supports some form of programming reflection by providing Objective-C applications with runtime functions for either querying for features of a class or an object, adding new class methods or even creating new classes at runtime [App10b].

Discussion

As will be discussed in Section 2.3.3, existing code offloading approaches explicitly target applications written in managed programming languages such as Java or C# on mobile platforms such as Android, which support the programming language constructs and runtime facilities described above. Either all, or just a subset, of these properties are essential for the operation of current offloading systems. They allow for automatically transferring application state to a remote node, executing application code remotely on demand and merging state changes on the mobile device to resume with local execution. By relying on such properties, existing offloading approaches are not applicable to applications written in unmanaged programming languages with rudimentary runtime support, such as Objective-C applications on the iOS platform.
2.1.3 Mobile networks

Mobile networks play an important role determining the efficiency of mobile client applications and code offloading solutions. Mobile client applications that frequently interact with backend services exhibit degraded performance due to the limitations of mobile networks in terms of increased latency and limited bandwidth. Offloading systems rely on mobile networks to deliver remote execution requests and associated state to a remote node and return the results and updated state upon completion of the task.

This section provides insights into how mobile networks operate. This is important for realising the performance bottlenecks of offloading systems. It also motivates our network-focused offloading approach for reducing mobile network traffic. We discuss further solutions that were proposed in the past to reduce latency and bandwidth usage in networks connecting mobile devices to the outside world. These comprise optimisations to the transport and application layer protocols currently used, as well as intelligent caching and redundancy elimination techniques that have been proposed.

Architecture overview of mobile networks

As shown in Figure 2.2, mobile networks typically follow a hierarchical structure, which conceptually comprises three abstract layers: edge, backhaul and core networks [WJP+13, EGH+11].

The edge network is the radio access network (RAN) and consists of a set of base stations and user equipment (UE), i.e. mobile devices. UEs communicate directly with base stations in their vicinity over wireless communication channels, e.g. according to the 3G/4G standards.

The core network at the other end of the hierarchy interconnects devices on edge networks with the public Internet via gateway routers to IP networks. Core networks have high capacity, and some control-plane tasks take place in core networks, such as the tracking and paging of UEs, hand-overs and authenticating users.

Between the core and edge networks lies the backhaul network. It includes multiple radio network controllers (RNCs), which are responsible for controlling connected base stations and relaying data to edge or core networks. RNCs also manage radio resources and the
encryption of user data.

Data traffic between mobile devices and the Internet traverses all three networks: UEs request and receive data via the base stations over the RAN; the backhaul network forwards data traffic between the base stations and the controlling RNC; finally, gateway routers route data between the core network and the rest of the Internet.

Typically network capacity can be limited in edge and backhaul networks [Tec10]. In the backhaul network, mobile network operators have therefore invested in fiber or high-frequency, point-to-point wireless links to handle increasing data demands. In edge networks, however, increasing capacity is bounded by radio resource constraints of 3G/4G networks, such as their limited frequency spectrums [RA13]. In addition, adding more base stations in a geographical region entails a high operational cost and is often constrained by frequency interference and placement problems. Deployments of smaller range base stations such as Femtocells and Picocells [CHS08] can add more capacity to RANs, but they require significant investment by mobile operators.

**Optimisation of transport- and application-layer network protocols**

Today, TCP and HTTP are the default transport and application-level protocols used in the web, respectively. Research on making both protocols faster in mobile networks has been under-way for decades. At the transport layer, solutions such as the Stream Control Transmission Protocol (SCTP) [NIAS06] and the Structured Stream Transport (SST) protocol [For07] have been proposed to replace TCP in order to achieve reduced web latency. SCTP provides multiplexed streams, with each such stream using its own sequencing space. As a result, multiple independent HTTP transactions are not delayed due to inter-transaction head-of-line blocking, i.e. packets being held-up by the first packet in an HTTP pipeline. Along the same lines, SST streams offer independent data transfer and flow control, permitting multiple parallel transactions without head-of-line blocking. It further provides a hierarchical stream structure, allowing applications to create lightweight child streams from existing ones.

Solutions that reduce web latency by changing the transport layer of the protocol stack are difficult to deploy in practise. They require changes to existing infrastructure, which is why such approaches have seen low adoption over the years. Perhaps the most prominent application-level solution for addressing the shortcomings of HTTP is the SPDY protocol [Pro13], designed specifically for achieving low latency. SPDY is deployed between the application and transport layers in the protocol stack to allow unlimited concurrent interleaved streams over a single TCP connection. Interleaving multiple requests on a single channel is more efficient because fewer network connections need to be made. To overcome bandwidth constraints, SPDY implements request priorities: a client assigns priorities to requests, thus preventing contention caused by non-critical requests when high-priority ones are pending. SPDY also compresses—and in some cases eliminates—request and response HTTP headers. This results in reductions in network usage between clients and servers.
Caching and redundancy elimination techniques

Caching popular content has been proposed to reduce latency and bandwidth in backhaul networks [WJP+13]. Typical forward caching techniques [EMS94, EGH+09, EGH+11] use dedicated middleboxes in backhaul networks for intercepting HTTP requests. In doing so, they first check if a local copy of the requested content exists (unless a fresh copy is requested explicitly or the content is marked as not cacheable). If the cached content is stale, the web server is checked to discover modifications since the content was cached. If the content was not changed it is served locally; otherwise a fresh version is requested from the web server. As a result, caching suppresses redundant transfers of the same content from remote web servers, thus reducing latency and bandwidth across links connecting backhaul networks to the corresponding web servers.

More advanced approaches for redundancy elimination (RE) propose protocol-independent techniques for eliminating redundancy in network traffic [SW00, AGA+08, AMAR09]. These are used to identify the same web content—possibly mirrored on different servers, named by different URLs or even delivered using different protocols—by processing the payload of packets. On a high-level, RE identifies contiguous byte sequences in packets that are also present in a cache. This assumes a shared cache architecture, with synchronised caches located at both ends of a bandwidth-constrained channel. On the sender’s side, representative fingerprints for each packet are computed, which are compared against a fingerprint store that holds the fingerprints of packets included in the cache. When a match is found, the corresponding packet is retrieved from the cache. Using this packet as a starting point, the maximal region of redundant content is obtained by expanding it in both directions byte by byte. The matched segments are finally replaced by pointers to the cache. The smaller encoded packets are finally sent over the network, thus reducing network traffic.

Discussion

The capacity of RANs, i.e. 3G and 4G networks, in current mobile networks is unable to keep up with the exponential growth in mobile traffic. This suggests that RANs need to be treated as a bottleneck resource. Nevertheless, existing code offloading approaches over-utilise such networks due to their inherent design that requires transferring potentially large amounts of application state with every offloading operation. This leads to increased overheads, which limit the performance gains achieved by code offloading, suggesting that code offloading practices that are not designed around typical state migration techniques should be explored.

Furthermore, mobile applications that retrieve large amounts of data from Internet backend services add to the problem of contention in RANs. In Chapter 5, we describe a network-focused offloading approach that reduces network traffic in RANs. A comparison of this approach with existing solutions that reduce mobile network traffic, as described in this section, will be given in Chapter 5.
2.2 Support for remote access programmability

In this section, we discuss research that regards remote execution as a stand-alone programming paradigm to allow the proliferation of new distributed mobile applications. Unlike automatic code offloading approaches that partition existing mobile applications, the approaches described in this section require major application redesigns and developer involvement.

We first provide an overview of mobile middleware platforms, which expose an isolated layer of specialised APIs to conceal the complexity of distributed application development. We next discuss previous work on cyber-foraging, proposing new application architectures that opportunistically leverage remote resources for increased application performance.

2.2.1 Mobile middleware

Numerous middleware solutions have been proposed in the past to facilitate the development of distributed applications. A middleware interconnects the different components that comprise a distributed application and provides abstractions for managing component interactions across multiple machines, avoiding the complexity of low-level network primitives [MCE02]. A middleware design needs to consider factors such as: the type of device that it is intended for, e.g. powerful desktop machines (referred to as stationary devices) vs. mobile devices with limited resources; the type of network connectivity between communicating machines, e.g. reliable high-bandwidth wired vs. unpredictable wireless connectivity; and the execution context that influences an application’s runtime behaviour, e.g. fairly static in stationary systems vs. extremely dynamic in mobile systems.

Current middleware systems can be divided into two main categories: middleware for stationary and for mobile distributed systems. They differ in their computational requirements, the communication paradigms supported and the context models exposed to applications. Here, we focus on middleware that explicitly target mobile environments. We first briefly describe the overall design space of traditional middleware for stationary distributed systems, which led to mobile middleware solutions. We then discuss the different types of mobile middleware systems proposed to date based on the taxonomies used in related literature [MCE02, GK03].

Middleware for stationary distributed systems

A middleware platform for stationary distributed systems belongs in either of three main classes: object-oriented, message-oriented and transaction-oriented middleware, depending on how it handles the communication between distributed application components. Examples of object-oriented middleware consist of the CORBA component model [Vis97], Microsoft COM [Rog97] and Java/RMI [PM01]. Such systems use remote procedure calls (RPCs) as the means of communication between distributed objects, which call into one another’s instance methods. They can thus only support synchronous communication, in which caller objects block until they receive a response from the corresponding callee objects.
Message-oriented middleware, e.g. Java Message Queue [MHCL00] and MQSeries [WAS99], support asynchronous communication, with clients and servers communicating by exchanging messages. A message contains either a request with its associated input parameters or the corresponding result of such a request. Though they decouple clients from servers by allowing execution to proceed after a request was submitted, they often require large amounts of memory for storing queues of messages received but not yet processed [MCE02].

Finally, transaction-oriented middleware, such as CICS [Hud94] and Tuxedo [Hal96], are designed for database-like applications. Multiple operations are grouped together to form transactions, which are offloaded to servers for remote execution, thus guaranteeing atomicity and high reliability. This approach supports both asynchronous and synchronous communication patterns depending on the semantics of an application.

Though widely adopted in stationary distributed environments, the above approaches exhibit high computational demands, which makes them unsuitable for mobile environments. To this end, new technologies tailored to the specific requirements of a mobile environment (e.g. limited availability of compute, memory and network resources, frequent network disconnections etc.) have been proposed. In the remainder of this section, we describe three different types of mobile middleware: context-aware and reflective, tuple-space and data sharing middleware.

Context-aware and reflective middleware

A special class of mobile middleware combines context-awareness with the principle of reflection [Smi82]. Such approaches attempt to create light-weight and reconfigurable middleware systems that provide contextual information to applications, e.g. user location, device characteristics and network availability. They allow modifications to themselves by changing dynamically the application’s internal behaviour, i.e. modifying existing and/or adding new features. We describe four representative middleware systems that fall into this category.

OpenCorba [Led99] is an example of a reflective approach that supports meta-class programming, i.e. the ability to differentiate between policy and mechanism. Policy is defined by base-level classes that denote the behaviour of objects. Meta-classes (i.e. classes of classes) are used to define the behaviour of base-level classes. This clear separation of what an object does from how it does it allows for altering an application’s behaviour at runtime. For example, in order to adapt remote method invocation mechanisms based on changes in network conditions, two meta-classes are provided: a default meta-class supports the Java remote method invocation (RMI) API; another meta-class supports object migration for retrieving remote objects in order to minimise remote communication under low network conditions. The dynamic adaptability of object behaviour is realised by altering the association between a base-level class and its corresponding meta-class during execution, without upsetting existing architecture.

Open-ORB [BCA*01] proposes a similar reflective architecture, in which each application component is associated with a meta-space that includes a number of meta-models. Meta-
models are responsible for the different aspects of the middleware implementation, regarding both the structure (internal and external representation) and behaviour (managing resources and the execution environment) of components. They are accessed through a meta-object protocol (MOP) that provides support for discovering the different services offered by components, manipulating the graph structure of interconnected components, adding additional functions to a component’s interface and reconfiguring resource management activities at runtime.

*Globe* [SHT99] is a middleware platform specifically designed to facilitate large-scale distributed applications. It introduces the concept of distributed shared objects (DSOs), i.e. conceptual objects that are distributed over multiple physical machines through which different processes communicate with each other. A DSO provides the interfaces that define an object’s set of methods and also encapsulates strategies for replication and distribution. To communicate through a DSO, a process needs to bind to the object by generating a new object instance in its address space to act as the DSO’s local representative.

*Nexus* [FKV00] is a location-aware middleware that provides an infrastructure to support heterogeneous communication environments. It has four different cooperating components: a user interface to communicate with the Nexus platform through mobile devices; a sensor component that combines various positioning systems to supply the Nexus platform with location information; a communication component to handle data exchanges between different Nexus components; and a distributed data management component to ensure data interoperability across different applications.

In general, context-aware and reflective middleware systems provide the mechanisms for creating distributed mobile applications that can modify their behaviour dynamically according to contextual information of the execution environment. However, they provide no policy, relying on the developer to decide how to distribute execution across nodes. They do not explicitly address the problem of limited resources on mobile devices but only provide the means for realising context-aware and reflective distributed applications. Nevertheless, the various techniques described in this section could be leveraged in order to address performance limitations of current mobile applications.

**Tuple-space middleware**

Tuple-space is a different kind of middleware designed to cope with network disconnections, which is a frequent occurrence in mobile environments. It naturally supports asynchronous communication in order to exploit connectivity when possible and at the same time support offline operation. It builds on the notion of tuple-spaces [Gel85], i.e. globally shared memory, which are used by processes to communicate with each other. A tuple-space contains multiple tuples (i.e. vectors of fields) that can be created, read and written by processes. Each tuple has its own life span, independent of that of the process that created it, permitting decoupled communication in space and time. Next, we discuss three representative middleware solutions that use tuple spaces.
In LIME [PMR99], each mobile unit has its own local tuple space, as well as an interface tuple-space (ITS) that is permanently attached to it. Each ITS contains the tuples that all units wish to share. Its contents are dynamically recomputed with every new connection or departure of a mobile unit. A unit accesses tuples in its ITS using operations for reading, withdrawing and adding new tuples, which are provided by Linda, a model of coordination and communication among multiple processes that operate on shared, virtual, associative memory [Gel85]. LIME therefore operates on different projections of the shared tuple-space. Information regarding the system’s configuration is kept in a read-only tuple-space and execution context changes may trigger different actions. Therefore, LIME provides a limited form of context-awareness but does not support reflection.

TSpaces [WMLF98] combines tuple-spaces and database technologies, providing a flexible communication model with query capabilities for data retrieval. It supports non-blocking and blocking transaction-based operations, which are decoupled from data. Therefore, operations can be added or removed without modifying the corresponding database. TSpace servers are centralised servers that listen for requests and maintain the tuple-spaces, which are communicated to clients.

JavaSpaces [Mic98] acts as a virtual space for objects to be shared by multiple processes in a distributed environment. Unlike previous approaches, tuples consist of instances of any Java class. Developers are thus given interfaces to create and persist objects to storage using tuple-space primitives. Tuple-spaces are hosted by server machines and their integrity is ensured by providing transaction-based communication paradigms, just as in the case of TSpaces.

Overall, tuple-space middleware provides mainly solutions for supporting asynchronous communication in mobile environments by exploiting the decoupled nature of tuple-spaces. It thus deals with intermittent connectivity between communicating endpoints in mobile distributed systems. As in the case of context-aware and reflective middleware, it does not provide an offloading policy and is not explicitly designed with the intention of improving application performance by allowing access to more powerful cloud-based resources. In fact, the systems described above impose additional resource requirements by, for example, requiring at least 60 Mbytes of main memory [GK03], which is a significant fraction of the available memory of current mobile devices [Cra13].

Data sharing middleware

A different class of middleware focuses on the challenges of data sharing in mobile distributed execution environments. These systems are designed to increase data availability by replicating data on multiple sites. At the same time, they provide the mechanisms to enforce eventual consistency by resolving conflicting updates to the same data at runtime. We describe four examples of middleware systems belonging to this category.

Coda [SKK+90] serves as a distributed file system intended for large-scale distributed execution environments. It provides fault tolerance by replicating files across multiple servers and
allows for disconnected operation using temporary caching to substitute replication servers when inaccessible. The data replicated across multiple sites consists of a set of files corresponding to a sub-tree of the shared file system. Server machines form the core infrastructure of Coda and can be accessed by clients. Coda servers are assumed to be secure, trusted and continuously monitored by operators. Clients that are offline for long periods of time can only access their local cached data. Upon reconnection to the system, any modifications to the cached data are propagated transparently to the set of affected replicas. By default, the system is responsible for detecting and resolving update conflicts but it also supports application-specific resolvers.

Odyssey [Sat96] extends Coda with context-awareness and reflection semantics. Applications register their interest in certain local resources. They define bounds on the availability of each resource that they require during execution, as well as the action to be taken when these requirements are not met. A separate component is responsible for monitoring resource usage at client sites in order to notify the corresponding applications when violations in the expected resource availability occur. In this way, applications are able to adapt their policies for accessing the file system according to execution context changes that affect them.

Dedicated system components implement the different access policies used by applications, thus providing customised middleware behaviour. Some limitations of this approach relate to the coarse granularity of data units (namely files) exchanged between resource-constrained mobile clients and the encoding of such files into byte streams, which complicates the process of detecting and resolving conflicts.

Bayou [DPS+94] is a similar approach for supporting applications that share data in distributed environments. Reads and writes are performed by any server without requiring coordination between multiple replicas. Eventually, however, all operations are propagated to all servers. Applications in Bayou are aware of the possibility of accessing weakly-consistent data and thus are directly involved in the process of conflict detection and resolution. For this, Bayou provides the means to applications for defining conflicts and the corresponding policies to be used for conflict resolution, which are automatically applied by the system. Every write operation is accompanied by a query and its expected result. Conflicts are detected by running the query against the server copy of the data and comparing the result to the expected value. When a conflict is detected, the server invokes a merge procedure unless user intervention is required. Data consistency is eventually guaranteed by enforcing an exact ordering on pending write operations across servers and requiring deterministic mechanisms for conflict detection and resolution.

XMIDDLE [MCZE02] is another system developed for mobile hosts to share data. Unlike previous approaches, data is stored in tree structures directly on mobile devices, i.e. it does not assume dedicated servers for data storage and replication. XMIDDLE uses a more fine-grained data replication unit that considers any branch of a tree structure. Linking to a tree on a remote host resembles the mounting of a network file system to access data that resides on a remote disk. Disconnected hosts retain replicas of the shared data and continue to manipulate
this data at will. Upon reconnection of hosts, non-conflicting updates to different replicas are merged automatically. When conflicts exist, application-specific resolution policies are used during the merging process, as provided by application developers.

Data-sharing middleware systems focus on maximising the availability of data in mobile distributed systems by means of data replication and mechanisms for enforcing eventual consistency. While this middleware class does not address the challenges of limited compute, memory and network resources on mobile devices, the techniques described could be used to enhance more recent code offloading approaches that aim at improving application performance by, for example, replicating and allowing concurrent access to application data files on local and remote nodes in the case of multi-threaded mobile applications.

Discussion

In this section, we have described different types of middleware that conceals the complexity of distributed application development in mobile environments. These approaches provide mechanisms but no policy for distributing execution. As it stands, they are intended for new applications to be developed with specific distributed deployment patterns in mind and do not address the performance limitations of existing mobile applications.

All approaches described require a minimal middleware core on a mobile device that allows applications running on different devices to monitor environment context changes, dynamically modify their behaviour, share data etc. While many such middleware systems have been developed to date, none of them addresses all challenges involved with distributed mobile application development, such as frequent environment changes, data-sharing requirements, intermittent connectivity etc. Instead, each platform addresses only a subset of the limitations intrinsic to mobility. In addition, these approaches have seen low adoption due to the substantial CPU or memory requirements that they entail, which impose additional overhead to applications that run on resource-constrained mobile devices [GK03].

2.2.2 Cyber-foraging

Cyber-foraging was first introduced by Balan et al. [BFS+02] as a new form of pervasive computing that opportunistically leverages computing resources in the vicinity of mobile devices to offer an enhanced mobile experience. This is accomplished by enabling a new class of mobile applications that are capable of offloading resource-intensive computation to more powerful machines when available.

Cyber-foraging entails several challenges. It requires efficient mechanisms for automatic discovery of remote machines (referred to as surrogates) in the vicinity of mobile devices at runtime. Complex dynamic decision-making processes, which decide whether or not offloading computation is beneficial during execution, need to account for offloading costs and the resource availability in surrounding surrogates. Furthermore, client devices and surrogate
machines need to establish mutual trust, while efficient techniques for transferring control of execution between them are required. The remainder of this section discusses some representative cyber-foraging systems that address these challenges (or a subset of them).

RPC-based cyber-foraging

Spectra [FPS02] is a system built on top of the Odyssey middleware for adaptive scheduling of computing tasks. Applications running on mobile devices register their operations with a Spectra client running locally, along with a set of possible execution plans per operation, i.e. variations of the actual implementation of an operation with possibly varying fidelities (application-specific metric of quality). An execution plan makes use of any number of services, i.e. pieces of code installed on Spectra servers. Servers may also be instantiated on mobile devices for local execution. These services are defined manually by the developer using a simple API. They are invoked directly from within the application using RPCs, and each such invocation spawns a separate process on the corresponding Spectra server.

A Coda file system is used to support data sharing between Spectra clients and servers, providing eventual consistency guarantees under low bandwidth conditions. In addition, clients and servers are equipped with components for monitoring CPU usage, energy consumption, network conditions and the state of the cache of the Coda file system. The system uses this information, along with the history profile of each operation and its execution plans to choose the optimal execution plan for deployment when an operation is about to be invoked.

Chroma [BSPO03] is a descendant of Spectra that uses an alternative way for defining execution plans, referred to as tactics. Tactics are high-level descriptions of meaningful module-level partitions of an application, as provided by the developer. They specify the remote calls that can be used by an application and the exact sequence of calls that carry out a certain task.

To further improve performance, Chroma employs a number of optimisation techniques based on parallel execution. First, it runs a tactic on multiple surrogates in parallel—the first to return signals the completion of a task. In addition, it splits the input data passed to a service, invokes the service on multiple surrogates with different portions of the input data and, finally, combines the results obtained. Developers need to provide Chroma with the corresponding functions for splitting input data and merging partial outputs. Finally, it deploys multiple tactics on different surrogates and chooses the result obtained from the tactic with the highest fidelity out of those that finish within a specified time frame.

In Chroma, users are also given the option of providing utility functions to quantify the trade-off between multiple optimisation goals. Using configuration files, the system supports user-defined policies such as favouring reductions in energy consumption over application response time. Tactics are chosen accordingly at runtime.
VM-based cyber-foraging

A different approach to cyber-foraging uses virtual machines (VMs) in the role of remote execution environments [GC04]. Unlike previous approaches, in which surrogates are defined statically in configuration files on the client side, this work proposes a surrogate discovery protocol to locate potential surrogates at runtime. Initially, surrogates register with a service discovery server, providing it with a description of their respective capabilities. Clients query this server at runtime to learn about surrogate availability. When a query returns with a surrogate’s network location, it is used by the client to negotiate a contract with the specified surrogate. Finally, the surrogate creates a new VM image and returns the corresponding VM IP address to the client. The client then installs the required services onto the VM image and thereafter has access to the surrogate’s resources. The application itself is responsible for accessing the VM for remote execution by migrating entire processes.

Slingshot [SF05] is another VM-based cyber-foraging system, in which surrogates can assume either of two roles: a first-class replica that is always available over the Internet and used when no nearby surrogate is available; and second-class replicas that are deployed on WiFi hotspots to offer reduced response times. Second-class replicas receive the corresponding VM images from first-class replicas when about to be used by an application for remote execution. Second-class replicas are instantiated after the client sends a migration request to the corresponding local surrogate. The local surrogate contacts the first-class replica and requests a copy of the current VM image. Data on disk are hashed and only the corresponding mapping is sent during migration. Thereafter, individual data blocks are received and stored in a local cache of the second-class replica on demand. This helps save network bandwidth and storage when initialising new second-class replicas.

Alternatively, Slingshot also allows clients to maintain snapshots of the first-class replica to be sent to second-class replicas directly. These are accompanied by logs of operations performed since the latest available snapshot. The second-class replica replays all events that are not reflected in the available snapshot and thus is able to reconstruct the up-to-date VM image without contacting the first-class replica.

At runtime, both first-class and any second-class replicas perform resource-intensive operations in parallel. The application uses the first result that is returned. If this is provided by a second-class replica, the result is compared against the corresponding result returned at a later stage by the first-class replica. This is to verify the trustworthiness of the second-class replica. In case of an inconsistency between the two results, future cooperation with the second-class replica is discontinued.

Discussion

As in the case of mobile middleware platforms, cyber-foraging systems require that developers familiarise themselves with specialised development frameworks for building applications
according to new distributed models. This allows for opportunistically leveraging nearby remote resources to achieve enhanced application performance. Nevertheless, the offloading of computation to remote surrogate machines needs to be explicitly included as part of applications by use of either RPC or VM migration techniques.

Relying on developers to define offloading policies requires understanding the resource requirements of application components, the relevant performance gains achieved by offloading execution and how these are affected by environmental changes. This type of information is often hard to estimate in advance, and thus user-defined policies may lead to sub-optimal or even decreased application performance.

### 2.3 Automatic application partitioning

We have discussed research areas that offer new techniques for developing future mobile applications that are capable of running across multiple machines for better application performance. However, the increasing number of existing mobile applications that exhibit degraded performance due to the limited availability of local resources has led to application partitioning techniques for automatically transforming applications to benefit from cloud-assisted execution. This offers several advantages: (i) developers do not need to familiarise themselves with new programming frameworks; (ii) it avoids a redesign of existing mobile applications to conform with distributed deployment patterns, and (iii) the performance overheads due to runtime system support can be controlled by tailoring a partitioning to the requirements of a given application domain.

In this section, we discuss different approaches for automatic application partitioning. In Figure 2.3, we classify these approaches according to the partitioning granularity, i.e. the smallest unit of code that can be offloaded for remote execution, and their ability to adapt offloading policies at runtime, i.e. static vs. dynamic solutions.
2.3.1 Thin-client approaches

The simplest form of code offloading treats mobile devices as thin clients, which are used for (i) relaying user input to a virtual device image instantiated on a powerful remote server and (ii) displaying screen updates. This offloads the entire functionality of an application to a remote server using a virtual machine (VM) approach. In this section, we describe two such thin-client approaches that allow resource-intensive mobile applications to run in more powerful remote execution environments.

Virtual Smartphone over IP [CI10] provides the means for creating customisable virtual smartphone images in remote infrastructures. It offers an environment that hosts multiple virtual smartphone images, each of which is assigned to a particular user. Using a dedicated front-end server, users are able to create, configure, destroy and establish a remote session to an appropriate VM image. Once a session is established, mobile applications may be installed and run on the corresponding VM images. A network file system is used for persistent file storage.

A client application is installed on smartphone devices, which relays user input to the VM device clone and receives the screen output. The interaction between smartphones and virtual smartphone images is carried out using conventional VNC-based technology [RSFWH98]. Touch screen events and other sensor readings on the client side are relayed to the corresponding virtual sensor drivers on the remote node automatically, whereas graphic pixels of updated screen images are delivered to the smartphone for display.

VM-based Cloudlets [SBCD09] is a new system architecture that exploits powerful machines in the vicinity of mobile devices (referred to as cloudlets) accessed over a wireless LAN. Using VMs, users can instantiate applications on cloudlets, with mobile devices typically acting as thin clients and running all computation remotely. Dedicated user-level processes run on both mobile devices and cloudlets and provide transparent support for service discovery and network management. At start-up, VM state is sent to nearby cloudlet infrastructures using a dynamic VM synthesis technique. Both the mobile device and cloudlet precompute a base VM, i.e. a VM in which a minimally-configured guest OS is installed. Mobile devices extract a small VM overlay and send it to the cloudlet. The VM overlay is created by differencing the state of the VM after launching the desired application (launch VM) with that of the base VM. The cloudlet then applies this overlay on its own copy of the base VM to obtain the corresponding launch VM and proceeds with execution.

Discussion

Thin-client approaches such as the ones described above belong in the category of static offloading solutions: the only alternative that they provide to local execution is that of running the entire application in a VM device clone. They rely on high-bandwidth, ubiquitous network connectivity for remote execution, and it is impossible to continue execution after loss of connectivity without having access to the remote VM’s state. The large amount of
Background

Data transfers between mobile devices and VM device clones in the way of screen image updates makes them less efficient than more flexible offloading approaches, which offload computation at the granularity of e.g. individual function calls and avoid excessive network communication. This is especially the case for highly-interactive mobile applications such as game applications that process user requests and update their displays frequently.

2.3.2 Partitioning at the granularity of application components and classes

To address the shortcomings of thin-client approaches, more advanced offloading systems have been proposed, which allow for more flexible application partitionings that operate at the level of application components, i.e. reusable software packages or modules in binary form that encapsulate a set of related functions or data, and classes. Here, we discuss three representative systems that fall into this category.

Coign [HS99] is a system for automatic partitioning and distribution of applications that are built from distributable components that conform to Microsoft’s Component Object Model (COM). It first constructs a graph model of inter-component communication by profiling execution under typical usage scenarios. Profiling information is collected by instrumenting the application’s binary code to intercept component interface calls. In doing so, Coign measures the amount of data communicated between different components. At the same time, it generates a network profile for the given execution environment by sampling the communication delays of message exchanges over the network. This information is used to infer the communication delays due to inter-component interactions, if the communicating components were to be placed on different machines. In addition, Coign also identifies placement constraints by analysing application binaries and component communication patterns. For example, components that access a set of known GUI or storage APIs are placed on the corresponding machines implementing these APIs.

Coign applies a graph-cutting algorithm to partition the application. Its goal is to minimise delays due to network communication. A partitioning is realised by leveraging the COM binary standard, which allows for transparently interposing middleware layers between communicating components for true location transparency. Application code remains identical for in-process, cross-process and cross-machine communication, thus allowing Coign to automatically distribute components without requiring source code modifications.

Another such system is Wishbone [NTG+09], which partitions resource-intensive sensor network applications across a number of sensor nodes and servers. It produces an optimal partitioning for high-rate data processing applications that minimises network bandwidth and CPU load. It assumes that applications are written as a collection of stream operators in a high-level stream-processing language and can be configured as a data-flow graph.

Wishbone first profiles the execution of each operator on either real hardware or simulation against sample data that is supplied by the developer. It assumes that the input data rates and communication patterns of sensors are predictable. This yields information regarding
the CPU and communication requirements of each operator on a given platform. Using this information and the application’s data-flow graph, it employs an integer linear program (ILP) solver to decide a partitioning, which minimises a combination of network bandwidth and CPU consumption.

**J-Orchestra [TS09]** is an automatic partitioning system for Java applications. It rewrites applications in byte code format to realise a partitioning that distributes objects across multiple machines. This process uses a compiler to substitute direct object references with references to proxy objects. Proxy objects hide the details of whether the actual object is local or remote. If remote methods need to be invoked, the proxy object propagates the method calls over the network using RPCs.

J-Orchestra has a partitioning mechanism but no policy, i.e. the user has to specify the location of application classes. In addition, since the system provides support for object migration, the user can associate mobility properties to application objects by specifying migration policies to describe when and how objects should migrate. Object migration occurs in response to run-time events such as when passing object parameters to remote method calls in order to exploit data locality by moving these objects to the callee’s address space.

Manually specifying the location and mobility properties of Java classes is an error-prone process that may yield inefficient or even incorrect partitionings. To ensure this does not happen, J-Orchestra includes a profiler tool that, based on offline application profiling, reports information about the inter-dependencies of classes. It also incorporates heuristics to provide an indication to users as to which classes should be placed together and where. A classification algorithm is used to verify further the correctness of a user-chosen partitioning. This algorithm analyses classes to identify dependencies that affect their placement and mobility properties, such as implementations using platform-specific code and accesses to instances of a class from within such code.

J-Orchestra overcomes the challenge of supporting transparent reference indirection for Java system classes (i.e. unmodifiable code) to avoid constraints regarding the placement of instances of such classes. It achieves this by dynamically wrapping direct object references to convert them into indirect proxy references when they are passed from unmodifiable to modifiable code (and vice versa).

**Discussion**

All of the above approaches are static. They derive a partitioning that cannot be adapted according to execution environment changes at runtime. No approach supports offline operation after network failure. In addition, they target high-level applications, either built from COM components, written as a collection of stream operators using a high-level stream-processing language or implemented in the Java programming language.
2.3.3 Partitioning at the granularity of functions and operations

More recent offloading approaches offload computation at a finer granularity, namely functions or even individual operations. Such approaches explicitly target mobile platforms and consider scenarios that involve partitioning mobile applications to execute across two nodes, i.e. a mobile device and a more powerful remote node. This section discusses four such representative code offloading systems, which are most similar to the work presented in this thesis. They are used for comparison in Chapters 3, 4 and 5, where we describe our compute-, memory- and network-focused offloading approaches.

CloneCloud

CloneCloud \cite{CIM11} is a system that automatically partitions mobile applications to execute across a mobile device and a cloud-based device clone. It operates on the binaries of Java applications running in Java VMs. CloneCloud statically analyses an application’s binary code and dynamically profiles its execution under given workloads in an offline mode. The information collected is used to derive a fast and energy-efficient partitioning for the execution environment considered. This partitioning is then realised by rewriting application executables to migrate application threads from mobile devices to their corresponding clones (and vice versa) at runtime, at the granularity of individual function calls.

To identify legal choices for migration and reintegration points of execution, CloneCloud uses static analysis. It considers three partitioning constraints: (i) functions accessing machine-specific features need to be placed on those machines, as identified by compiling a list of methods that provide these features via the VM API; (ii) functions sharing native state must be collocated, as identified by inferring all native function calls declared in application classes; and (iii) nested migration should be prevented, as identified by constructing a static control-flow graph that captures direct and transitive method interactions.

Dynamic profiling is used to construct a cost model for characterising a partitioning’s performance in terms of execution time and energy consumption. This process is repeated for different randomly chosen workflows, while running the application in its entirety on both the mobile device and the device clone. Each profiling run outputs a graph that captures function call dependencies between different class methods. This graph also encapsulates information regarding execution times of all application methods, as well as application state sizes before and after each method call. To collect this information, method entry and exit points are instrumented in order to measure (i) method execution times and (ii) the size of the captured state that possibly will be transferred from the mobile device to the device clone and back. The latter is estimated by simulating the state migration process at each profiled method, i.e. performing the steps required to capture migrated state when invoking and returning from a method call. The transfer latency cost per byte, i.e. the time to capture, serialise, transmit, deserialise, and reinstantiate state, is assumed to be the same for all objects and is measured during offline profiling. This is used to estimate the latency overhead.
incurred by each possible migration point, given the captured state sizes before and after the corresponding migrated method call. To estimate the energy consumption of a partitioning, a simple energy cost model considers CPU activity, display state and network state, which is estimated experimentally.

After profiling, an optimisation solver decides on the optimal legal partitioning, i.e. one that respects the aforementioned constraints and minimises an objective function, either with regards to execution time or energy consumption. A partitioning is chosen from a set of precomputed partitions according to the conditions of the execution environment upon application deployment. The partitioning is realised by instrumenting the application binary with special VM instructions that: suspend thread execution when a migration point is reached; package the thread’s virtual state, which includes reachable heap objects from objects currently on the stack, program counter values, registers and the stack; and transfer the state to the synchronised device clone. On the device clone, a new thread is instantiated accordingly and starts executing. When a reintegration point is reached, the same process takes place in reverse order. However, now all virtual state is sent to the mobile device and is merged with local state before resuming with local execution.

**MAUI**

*MAUI* [CBC+10] is an offloading system that minimises the energy expended on mobile devices by offloading computation to a cloud infrastructure at the granularity of function calls. During execution, MAUI continuously profiles application methods and the network connectivity to the cloud infrastructure. The profiling information is fed into an optimisation engine that decides which application methods should be offloaded for maximum energy savings given the current network connectivity constraints.

MAUI relies on developers to annotate the methods that should be considered candidates for remote execution. These methods must not implement user interface features or interact with I/O devices such as hardware sensors and external components that would be affected by re-execution. These annotations are done using the custom attribute feature of the .NET Common Language Runtime (CLR), which allows the association of meta-data with application methods. The MAUI runtime then uses the .NET Reflection API to identify annotated methods by searching through the executable for the corresponding attributes.

At compile time, MAUI generates wrappers for each candidate remote method, which add an additional input and return value to the original method implementation. The wrappers transfer the required application state from the device to the remote node and back for offloading. The state sent to the remote node with every offloaded method call corresponds to any data that may be referenced during remote execution. This includes the current object’s member variables (including nested object types), as well as all static classes and public static member variables. To capture this state, MAUI leverages the type-safe nature of the .NET runtime for traversing in-memory data structures.
Offloading is realised by generating two proxies for the smartphone and the remote node, respectively. The proxies transfer control of execution and associated data between the device and the remote node based on the decisions made by an optimisation solver at runtime. The local proxy performs state serialisation before an offloaded method call and deserialises the returned application state. At the remote side, when a method that requires local execution is invoked, a server-side proxy performs the necessary serialisation and transfers control back to the smartphone.

The decisions as to when to offload a method call are influenced by the device’s energy consumption characteristics, the CPU consumption of individual methods and the environment’s network characteristics (e.g. available bandwidth, latency and packet loss). The energy consumption characteristics are measured before execution by attaching a power meter to the device’s battery and running synthetic benchmarks to build a simple energy model. This model associates CPU and network utilisation to local energy consumption. Using this association, the optimisation solver can predict the energy consumed by a method as a function of the number of CPU cycles that it requires, as well as the energy consumed when offloading the method.

To characterise methods with respect to their CPU and offloading requirements, each method is instrumented to continuously measure the number of required state transfers when offloaded, its runtime duration and the number of CPU cycles consumed when executed. To account for both the latency and bandwidth characteristics of the network, a representative amount of data is sent to the remote node, and the duration of the transfer is measured. This is repeated whenever MAUI offloads a method, or every minute, to obtain a fresh estimate.

The above information is used to solve an optimisation problem that minimises energy consumption on the mobile device subject to latency constraints. This process is repeated periodically to adapt to changing conditions of the execution environment.

In MAUI, network failures are detected using a timeout mechanism. Once detected, a failed remote method call is repeated locally or using another available remote node if the failed method call was caused by server failure.

**ThinkAir**

*ThinkAir* [KAH+12] is an offloading system that combines features from both CloneCloud and MAUI. It dynamically offloads individual method calls of Java applications that run in VMs to a remote node for reduced application response times and energy consumption. It extends previous approaches by providing support for on-demand resource allocation and exploits parallelism using multiple VM device clones.

As in the case of MAUI, ThinkAir requires developers to mark candidate methods for remote execution using a library provided for this purpose. A *code generator* is executed against the modified code to generate method wrappers for candidate remote methods that integrate with ThinkAir’s framework. An *execution controller* transparently handles the decision-making
process regarding when to offload a method at runtime and manages the communication with the cloud-based VM clone. Decisions are based on profiling data collected at runtime by dedicated profilers, which are used to characterise the execution environment and to keep track of the history of previous method invocations.

Remote execution is achieved using Java reflection: the calling object is sent to the device clone along with its associated local state. The caller blocks while waiting for results and updated state. Connectivity failures are handled as in MAUI: a failed remote method call is repeated locally, while probing the server for connectivity to resume with offloaded execution.

The cloud-side counterpart of the execution controller manages client connections and executes offloaded method calls. It first acquires the application executable from a client and then waits for execution requests. On receiving a request, it is responsible for scaling the computational power of VMs and allocating multiple VMs per request on demand depending on user requirements. This allows developers to exploit parallel execution more naturally by splitting a computation across multiple VMs.

Profiling of applications involves hardware, software and network profiling. A hardware profiler monitors the state of the CPU, screen and WiFi and 3G network interfaces to determine the energy cost model to be used. A software profiler uses the Android Debug API to record information regarding method execution times, the number of executed instructions and method calls and the memory allocated by each thread. A network profiler tracks the state of the network and recalculates network metrics by measuring round trip times and the amount of data exchanged to infer relevant offloading costs when changes in network conditions occur. In addition, it collects information about parameters for the WiFi and 3G network interfaces, such as the number of packet transmissions per second and transmission rates, to estimate network performance. ThinkAir also incorporates an existing energy model to estimate the energy consumed by a method at runtime.

The above information is used to decide whether or not a candidate remote method should be offloaded to the remote node. Different policies determine this decision, either favouring reduced execution times, energy consumed or some weighted combination of both.

**COMET**

COMET [GJM+12] combines offloading with distributed shared memory (DSM) techniques to offload multi-threaded mobile applications to cloud-based infrastructures in order to reduce execution times. It is built on top of the Dalvik VM, which is modified to support DSM at the granularity of object fields. COMET operates on the binaries of Java applications and thus requires no source code modifications.

To support concurrent accesses to shared memory from threads running at different locations, COMET employs a variation of the lazy release consistency DSM protocol, which is more efficient in terms of communication required between the mobile device and the remote node. When a thread “acquires” access to application state due to thread migration, lock acquisition
or access to a condition variable or volatile memory, all dirty application data is transferred to
the executing thread. This technique keeps the heap, stack and locking states synchronised
across the communicating endpoints. This DSM approach takes advantage of the partial
ordering of reads and writes in Java’s memory model: memory accesses within a single thread
are ordered according to their chronological order; across threads this ordering is inferred by
the acquisition and release of the corresponding locks, condition variables or volatiles used to
synchronise concurrent memory accesses. Conflicting updates to data fields caused by race
conditions are resolved by use of heuristics such as favouring the maximum of all conflicting
values. This is enabled by Java’s managed runtime, which ensures that reads and writes can
only happen at non-overlapping memory locations of known widths.

Synchronising VMs across endpoints requires the merging of virtual heaps and stacks. For
the heaps, all dirty fields of class objects, local objects on the stack, global fields and other
reachable objects are transferred to the thread “acquiring” access to application state. Each
object is associated with a bit-set specifying its dirty fields, which is updated with any write
operation. The stack, including method calls, the program counter and the values of machine
registers, is sent in its entirety and used when migrating a thread after synchronisation.

To handle Java locks, an owner is assigned to each lock. A lock attempt to an object belonging
to the other partition results in the requesting thread either migrating to the owning partition
or acquiring ownership of the lock. Condition variables and volatiles are handled in a similar
fashion.

COMET’s scheduler decides when to migrate threads from the device to the cloud infrastruc-
ture. It relies on past behaviour: it tracks the duration of a thread’s execution on the mobile
device without invoking native method calls and migrates the thread when this exceeds some
configurable time interval. Initially, this is set to twice the round trip time between the two
endpoints, and is replaced gradually with twice the average time to synchronise VMs.

To recover from network failures, the mobile device needs to be able to resume all threads
locally. Therefore, the server is required to send an update of all of its thread stacks when
synchronising VMs. A failure is detected if an existing connection is terminated forcibly or
a cloud-based server stops responding to heartbeats.

Discussion

With the exception of CloneCloud, all other approaches described in this section are dy-
namic. Compared to MAUI and ThinkAir, COMET employs a simpler scheduling approach,
 focusing on the problem of how to offload rather than what to offload. Nevertheless, it could
be extended easily to support offloading policies similar to those of MAUI and ThinkAir,
which are influenced by changes in network conditions, by employing profilers to monitor the
execution environment.

In terms of fault tolerance, all approaches except for CloneCloud describe how they can
handle network failures by re-executing a failed method call locally (MAUI and ThinkAir)
or resuming remote threads on the mobile device (COMET). This comes for free because in all approaches the application state is available on the mobile device at all times during execution. This suggests that CloneCloud could also adopt a similar approach for handling intermittent connectivity in mobile environments.

Furthermore, what is common in all four approaches is the fact that they rely heavily on properties of managed programming languages and runtime support from the JVM and CLR runtimes. First, they require programming language support for strong type-safety to capture the application state that needs to be migrated to the remote node along with a request for remote execution. In addition, they assume runtime support for code mobility to move code, data and execution state from one location to another and resume execution from a well-defined execution point. MAUI and ThinkAir also exploit the runtime support for programming reflection of the CRL and JVM runtimes, respectively, to identify candidate methods for remote execution dynamically—according to developer annotations—and to modify method behaviours accordingly. Support for object serialisation is also used to quantify the data transmission costs involved with state migration before making an offloading decision.

2.4 Summary

This chapter first gave an overview of current mobile environments to identify limitations that prevent mobile applications from exhibiting functionality and performance comparable to desktop applications. We began with a discussion of the hardware capabilities of mobile devices and desktop machines. We showed that mobile hardware remains restricted due to form factor constraints, which motivates our work on code offloading. We next described the two dominant mobile platforms, Android and iOS, focusing on the support that they provide to mobile applications. We investigated four features of mobile platforms that are essential to the operation of offloading systems: support for type safety, code mobility, object serialisation and programming reflection. The significant differences between Android and iOS platforms with respect to the above reveal why current code offloading approaches target only applications written in managed programming languages with their runtime support.

After that, we described mobile networks, which have an impact on the performance of (i) code offloading systems that deliver requests and associated data to remote nodes and (ii) mobile client applications that interact with backend services. We presented the general architecture of mobile networks and identified performance bottlenecks pertaining to the excessive bandwidth usage of mobile users. This helped realise the importance of limitations of existing code offloading systems, which by design exchange large amounts of application state with every offloaded operation. It also motivated the need for solutions that minimise user traffic, especially over the network links between mobile devices and base stations. We concluded this section with optimisations that were proposed to address network limitations.

In addition, we discussed previous work on providing support for developing distributed mobile applications. We gave an overview of mobile middleware platforms that help mask
the complexity of distributed application development. We showed that such approaches provide mechanisms but no policies for distributed application deployment. Though mobile middleware explicitly targets mobile environments, each platform focuses only on a subset of the challenges intrinsic to mobility.

We also discussed cyber-foraging approaches for applications that opportunistically leverage nearby resources. Again, these approaches expect from developers familiarity with new complex distributed application architectures and require policies for offloading execution at runtime, which is an error-prone task.

The last section of this chapter discussed more recent code offloading approaches that automatically handle the partitioning and distribution of existing applications with minimal to no developer intervention. We classified such approaches according to the granularity at which they partition mobile applications. The simplest form of offloading approaches treats mobile devices as thin clients: the entire application logic is executed remotely on a device clone, while the mobile device is used only to relay user input and display screen updates. These solutions are static, relying on high-bandwidth ubiquitous network connectivity for efficiency, and cannot react to network failures. More flexible partitioning systems operate on already modularised applications and partition these at the level of application components. These approaches are also static and not designed with fault tolerance in mind.

The final class of code offloading approaches offloads computation at the finer granularity of functions or individual operations. We described static and dynamic code offloading systems that assume the same scenarios as this thesis: these include resource-constrained mobile devices, more powerful remote nodes, intermittent network connectivity between them and mobile applications with high resource demands. What is common in these approaches is that they maintain a copy of the entire application state on the mobile device. State is then migrated back and forth with every offloading operation. Therefore, existing approaches can handle network failures by repeating failed remote method calls locally. Another common characteristic is that they rely on properties of managed programming languages and their runtime support. We defer a detailed comparison of our work with these approaches to the chapters that introduce our compute-, memory- and network-focused offloading approaches.
Chapter 3

Compute-Focused Offloading

This chapter describes a static code offloading approach that exploits the compute resources of more powerful remote nodes to offer increased application performance. Using low-overhead dynamic code instrumentation techniques, it obtains an accurate characterisation of the application’s runtime behaviour prior to deployment in order to decide and implement a partitioning that yields reduced application response times.

3.1 Overview

One of the main contributions of this approach is that it lifts the restrictions of existing code offloading systems regarding the types of applications and mobile platforms that they are able to support. The motivation behind the work stems from the observation that a large set of popular legacy applications are written in unmanaged programming languages such as C, Objective-C and C++ with only minimal runtime support. These applications often exhibit compute-intensive behaviour, preventing them from executing on devices with limited resources. Although in principle cloud-assisted execution should support this, existing offloading systems fail in this regard by relying on runtime support for code mobility and strong type safety guarantees. They are, therefore, confined to supporting only applications written in managed programming languages that comply with these requirements, e.g. Java and C#.

To address the limited applicability of current approaches, we explicitly target applications written in unmanaged programming languages. To this end, we opted to work with the C programming language. C lacks support from a large runtime system, has no code structure other than that of functions and offers direct memory addressing. While C is not a typical mobile language, this choice perfectly demonstrates how cloud-assisted execution can benefit applications that are written in languages with almost no abstraction from a machine’s instruction set architecture.

We propose a static offloading approach that derives a well-informed application partitioning based on fine-grained profiling information collected offline. As a result, we avoid the overhead
of dynamic solutions, associated with adapting offloading policies at runtime (as discussed in Section 1.1). While a drawback is that it cannot react to environmental changes, a static approach is better suited for unmanaged applications with limited runtime support, possibly running on low-end devices with severe CPU constraints. To the best of our knowledge, CloneCloud [CIM+11] is the only other static code offloading approach proposed to date. Compared to CloneCloud, we make no assumptions on programming language facilities, nor do we require runtime support for code offloading. In addition, our techniques for profiling require no modifications to applications.

We want unmanaged applications to benefit from cloud-assisted execution in an automated fashion without the need for a major application redesign. Any attempt at automating this process is complicated by the fact that applications written in unmanaged programming languages lack an inherent notion of application components, which could be offloaded to execute remotely. Any structural subdivisions in these languages, e.g. functions, objects or files, tend to have strong dependencies between them in terms of function calls and shared memory state. This makes splitting applications into a local and a remote part especially challenging.

Our solution to this challenge comprises three steps:

1. We first analyse the application’s resource consumption using offline dynamic profiling. During this step the application executes a realistic workload that captures its runtime behaviour in terms of CPU and memory usage. Profiling operates on the binary version of the application using dynamic code instrumentation techniques.

2. Based on the profiling results, we statically partition an application into a local and a remote partition by identifying functions that would benefit from remote execution. The goal is to reduce application response times using the remote node’s faster compute resources for executing compute-intensive application functionality, while keeping offloading overheads to a minimum.

3. Finally, the application’s source code is transformed according to the partitioning decided. We inject new source code to change local into remote function calls using RPCs, while managing application state accordingly.

This approach is realised by AnyWARE, a code offloading system that statically partitions C applications to execute across a local and a remote node in accordance to the steps described above. We evaluate AnyWARE on a complex open-source C application to show that it manages to reduce the execution time of applications running on resource-constrained networked devices. Our experimental results show that under severe CPU constraints, AnyWARE achieves a speedup by a factor of 1.32× for the Transmission BitTorrent client [Tra05].

The remainder of this chapter is organised as follows: Section 3.2 compares our approach against related work. Section 3.3 introduces our techniques for profiling and partitioning a C application. Section 3.4 describes the overall design of AnyWARE and the process of realising
a partitioning by transforming C applications for cloud-assisted execution. In Section 3.5, we present the evaluation results after partitioning a real-world application using AnyWare.

We discuss the limitations of our approach in Section 3.6 and summarise this chapter in Section 3.7.

### 3.2 Design space

In this section, we position our compute-focused approach against current state-of-the-art offloading systems. This comparison is summarised in Table 3.1.

#### 3.2.1 Static vs. dynamic code offloading

As discussed in Chapter 2, code offloading systems can be divided into two main categories: *dynamic* and *static*. Dynamic systems, such as MAUI [CBC+10], ThinkAir [KAH+12] and COMET [GJM+12], decide at runtime when to offload execution to a remote node based on runtime profiling and properties of the environment, such as network performance, battery reserves and CPU load. In contrast, static systems, such as CloneCloud [CIM+11], precompute a code partitioning for a given application and execution environment.

Dynamic solutions have the advantage that they can adapt offloading policies in response to changes in network performance, energy availability etc. This incurs, however, an overhead for making offloading decisions each time execution reaches a given point such as a function call. In addition, changes in resource consumption and the environment must be monitored, adding further overhead.

In this work, we target unmanaged C applications. Due to the lack of runtime support for code mobility and the weak type safety in C, a static approach is more feasible. It also allows AnyWare to support applications running on low-end devices with severe resource constraints, for which any source of additional overhead has crucial impact on performance. AnyWare precomputes a set of application partitionings prior to execution, corresponding to different application workloads and network conditions (e.g. WiFi vs. 3G connectivity). When the application is launched, an appropriate partitioning is chosen.
3.2.2 Platforms and programming languages

Current offloading systems explicitly target mobile platforms that support applications written in managed programming languages with substantial runtime support. CloneCloud uses application-layer virtual machines (VMs), such as the Java VM, to migrate code to remote nodes. It also relies on strongly-typed programming languages for capturing the portion of the application state that needs to be migrated with an offloading operation. MAUI and ThinkAir also rely on the latter, as well as programming reflection to examine and modify the runtime behaviour of applications for remote execution. COMET uses DSM to manage memory consistency, which requires support by the runtime system. In addition, to allow for concurrent access to the same state without requiring communication between readers and writers, it relies on Java’s object model with type safety guarantees to resolve conflicts in updated state.

In contrast, AnyWare removes assumptions regarding the types of applications and mobile platforms targeted by providing support for unmanaged applications. It targets applications written in the C programming language. Despite its low-level capabilities, C was designed to encourage cross-platform programming, which is evidenced by its current availability on many platforms, from embedded micro-controllers to supercomputers. Portable C applications that are written in a standards-compliant way can be compiled for a wide variety of platforms.

3.2.3 Optimisation goals

The optimisation goals of current code offloading solutions include reducing application response times to improve user experience and energy consumption to prolong battery lifetime. MAUI focuses on minimising energy consumption, subject to a latency constraint, while COMET reduces execution time. CloneCloud minimises either execution time or energy consumption. ThinkAir allows for user-defined policies that minimise any weighted combination of both.

Although these two optimisation goals are not strictly positively correlated, reducing response times typically also leads to energy savings [CIM+11]. Therefore, AnyWare aims to lower response times. Its optimisation goal (see Section 3.3.2) can be altered to reduce energy consumption, as long as relevant cost information is available in the profiling phase.

3.2.4 Partitioning granularity

MAUI, ThinkAir and CloneCloud partition applications at the granularity of function calls, while COMET supports offloading of individual statements. We assert that offloading computation at the granularity of function calls is fine-grained enough to provide the flexibility for supporting essentially any unmanaged application, typically structured around functions. Since AnyWare is designed with this intention in mind, it also offloads individual compute-intensive function calls to a remote node.
3.2.5 Fault tolerance

Another important aspect of the design of current code offloading systems relates to the way that they handle intermittent network connectivity. MAUI and ThinkAir are designed to revert to local execution after failure until network connectivity is restored. Since the entire application state is available on the local node, a failed offloaded call is simply re-executed locally. Similarly, since COMET’s DSM model maintains consistent state replicas, all offloaded threads can be resumed locally after failure. The impact of network failures is not discussed in CloneCloud but its design permits a similar solution.

AnyWARE currently falls short of supporting uninterrupted execution in the face of network failures. While the local node also maintains a copy of the entire application state at all times, a solution along the lines described above is not directly applicable. The reason for this is the fact that C does not support exception handling natively. Therefore, there is no easy way to detect and react to a failed remote call on the fly. To circumvent this problem, an implementation that emulates exceptions and their semantics is required, thus allowing for catching the corresponding exceptions before the application is forcibly terminated. Some third-party library implementations [Cal13, Eng07] claim to provide similar semantics for the C programming language, relying on the setjmp and longjmp functions for saving and restoring a program’s environment. However, we leave an investigation of how these could be incorporated into our current design for future work.

3.3 Low-level application partitioning

In this section, we describe the process of partitioning a C application to execute across a local and a remote node. This involves two steps: dynamic resource profiling and optimisation-based application partitioning. During the former, we first obtain information about the application’s CPU consumption during execution, as well as function call and shared memory dependencies. During the application partitioning step, based on the profiling information collected, we decide on the parts of an application to be offloaded for remote execution.

3.3.1 Dynamic resource profiling

Profiling is an offline process that measures the resource consumption of an application under a set of workloads. It classifies individual application functions according to the amount of resources that they consume during execution. Profiling operates on binary versions and therefore does not require application modifications.

Each profiling run measures a particular execution trace given a workload. Multiple profiling runs can reflect different workloads that an application may experience, which are chosen to cover its average usage. The goal is not to explore exhaustively all possible workloads but to identify parts of the application that would benefit most from being offloaded to a more
powerful remote node, consistently across all profiling runs considered. The results of each profiling run are merged by computing the average of the measurements obtained.

Profiling granularity

An important choice is that of the profiling granularity for measuring the resource consumption of an application. Functions in a file usually interact with each other frequently. This could be exploited by treating each file as a separate component of the application, which simplifies profiling by reducing the amount of profiling data. However, functions within a file may vary in terms of their computational complexity, thus leading to cases in which many low-cost functions are marked for remote execution. This could have a negative impact on application performance due to the substantial network overhead of unnecessary remote calls. Therefore, we decide to profile applications at the function level. As this is more fine-grained, it provides a better insight into the compute-intensive parts of an application that would benefit the most from remote execution.

Profiling methodology

In order to obtain an accurate characterisation of an application’s CPU and memory consumption, dynamic profiling needs to be transparent, i.e. it should affect application performance as little as possible. To this end, our approach uses DTrace [CSL04] for collecting the appropriate information required to derive a partitioning. DTrace is a comprehensive framework developed by Sun Microsystems that allows for fine-grained dynamic tracing of applications in real time. It exploits probes, i.e. points of instrumentation in supported operating system kernels. Probes can be composed to create custom probes that are queryable at runtime. Custom probes are compiled to binary code and patched dynamically into a running kernel, with only a modest runtime overhead. Information regarding CPU times, memory accesses and network usage are just a few examples of what DTrace can provide. As it imposes a minimal overhead to the running process that it attaches to, DTrace is an ideal candidate for this task.

Resource consumption graphs

The output of each profiling run is modelled as a resource consumption graph. This graph is used to divide the application’s functions into two distinct sets: local functions that are executed on the resource-constrained device; and remote functions that are executed on the remote node.

Each vertex in the graph corresponds to a different function of the application and is assigned a vertex weight that identifies its average (over multiple profiling runs) execution time. Edges correspond to the average dependency between functions in terms of (i) calls between functions; (ii) data passed from one function to another in the form of function call parameter
and return values; and (iii) application state shared between functions via global variables. Each edge is assigned an edge weight to indicate the strength of the dependency between two connected vertices in the graph.

More formally, the model used to describe the resource consumption of a given application is a directed graph $G = \{F, T, C, D\}$. $F$ corresponds to the set of application functions. $T$ is a function that provides the total execution times for each function, averaged over all profiling runs. $C$ specifies call dependencies, i.e. $C(x, y)$ states the average number of calls from function $x$ to $y$. $D$ provides the amount of data that would be exchanged between the local and remote partitions, for all pairs of functions that call each other if these were placed on different partitions. More precisely, $D(x, y)$ gives the sum of the sizes of (i) $y$’s arguments, (ii) its return value and (iii) the global state right before and after all remote function calls from $x$ to $y$. Finally, a special class called native amalgamates all library functions that offer I/O and device-specific functionality. Edges between vertices and native are used to prevent the placement of functions that call native functions on the remote side.

Profiling example

We show an example of a resource consumption graph in Figure 3.1. It is a directed graph showing which functions call each other, how frequently they do so and the amount of data transfers each function call requires (assuming the caller and callee functions are placed on different partitions). Each node in the graph corresponds to a different application function and is marked by a weight that specifies its average execution time—excluding the time spent executing nested function calls, i.e. calls originating from within the function.

Using this graph, one can identify functions $B$ and $C$ with weights $500 \times$ larger than the next largest vertex weight as the functions that dominate CPU utilisation. While at first sight,
both functions pose good candidates for remote execution, function call dependencies also affect the performance of partitioned execution. The partitioning algorithm should therefore minimise the number of RPCs to avoid excessive cross-partition communication. In addition, the amount of data exchanged between nodes also has an impact on performance, for example when the network connectivity between local and remote partitions is bandwidth-bound.

As function $D$ is connected to native, it is said to be pinned to the device due to its interaction with device-specific functions. A partitioning that places functions $B$ and $C$ in remote would result in a total of 1020 RPCs, i.e. 10 from main to $B$, 1000 from $B$ to $D$ and 10 from $D$ to $C$, with a total of 2280 KB of application data (80, 2000 and 200 KB, respectively) being transmitted over the network. This translates to a high communication overhead between the local and remote nodes. To avoid this overhead, an alternative partitioning could only place function $C$ in remote, thus reducing the amount of remote calls to 110 (100 originating from $B$ and 10 originating from $D$) and the amount of data transferred to 280 KB (80 and 200 KB, respectively). An intelligent partitioning algorithm must therefore account for all factors that affect application performance in order to derive a partitioning that yields maximum performance gains.

3.3.2 Partitioning algorithm

We employ a heuristic-based approach to decide a suitable partitioning for an application given its resource consumption graph. The application functions are partitioned into two sets: those intended for local execution on the resource-constrained device (local) and those to be executed remotely on a machine with faster compute resources (remote). The goal is to minimise application response times by offloading compute-intensive functions to the remote node, while ensuring that any additional communication overhead introduced remains low. Therefore, a partitioning is evaluated on the basis of the following three properties:

$P_1$: Functions executing remotely consume a significant fraction of the total CPU resources required at runtime.

$P_2$: The number of cross-partition function calls remains low.

$P_3$: The amount of data exchanged between functions across partitions remains low.

Using these properties as reference, different partitionings can be evaluated on their effectiveness in reducing application response time. $P_1$ is used to identify the set of functions that are indeed compute-intensive. $P_2$ and $P_3$ further refine a partitioning by ensuring that highly dependant functions in terms of the number of calls between them and the amount of shared application state are placed on the same partition.

Our approach operates under the assumption that the CPU of the remote node is significantly faster than that of the mobile device. Furthermore, the remote node is capable of reserving an adequate CPU slice per application served so as to guarantee faster execution for offloaded
Algorithm 1: Partition application – *Compute-focused approach*

**input**: A resource consumption graph $G$

**output**: A set of remote application functions

1. $\text{maximum}_O = 0$
2. $\text{best}_\text{remote} = \emptyset$
3. **foreach** $\text{remote} \subseteq F - \{\text{native}\} \text{ do}$
   1. $\text{local} = F - \text{remote}$
   2. $\text{remote}_\text{execution}_\text{time} = \sum_{x \in \text{remote}} T(x)$
   3. $\text{ingress}_\text{calls} = \sum_{x \in \text{remote}, y \in \text{local}} C(y, x)$
   4. $\text{egress}_\text{calls} = \sum_{x \in \text{remote}, y \in \text{local}} C(x, y)$
   5. $\text{data}_\text{transmissions} = \sum_{x \in \text{remote}, y \in \text{local}} (D(x, y) + D(y, x))$
   6. $O = \frac{\text{remote}_\text{execution}_\text{time}}{(\text{ingress}_\text{calls} + \text{egress}_\text{calls}) \times \text{data}_\text{transmissions}}$
4. **if** $O > \text{maximum}_O$ **then**
   1. $\text{maximum}_O = O$
   2. $\text{best}_\text{remote} = \text{remote}$
5. **return** $\text{best}_\text{remote}$

Under these assumptions, we formalise the goal of the partitioning algorithm using the following maximisation objective function:

$$O(\text{remote}) = \frac{\text{remote}_\text{execution}_\text{time}}{(\text{ingress}_\text{calls} + \text{egress}_\text{calls}) \times \text{data}_\text{transmissions}}$$

The $\text{remote}_\text{execution}_\text{time}$ variable refers to the sum of execution times of functions in $\text{remote}$. $\text{ingress}_\text{calls}$ and $\text{egress}_\text{calls}$ are the number of calls from functions in $\text{local}$ to functions in $\text{remote}$ and the number of calls from functions in $\text{remote}$ to functions in $\text{local}$, respectively. Finally, $\text{data}_\text{transmissions}$ refers to the amount of application data that is exchanged between the local and remote partitions. A set of functions that maximises $O$ corresponds to functions with collectively high CPU usage and low data and function call dependencies with other functions: $O$ is proportional to the execution time of functions in $\text{remote}$ and inversely proportional to (i) the number of calls to and from $\text{remote}$ and (ii) the amount of data transferred over the network during code offloading.

The pseudo-code in Algorithm 1 describes how an application is partitioned given its resource consumption graph $G$. It considers all possible valid placements of application functions on the remote partition ($\text{remote}$) to compute their corresponding objective function values. In line 5, it computes the sum of the execution times of all functions placed on the remote node.
Lines 6 and 7 compute the number of function calls from local to remote and remote to local, respectively. In line 8, it calculates the amount of data exchanged between the local and remote nodes, i.e. the sum of edge weights of all edges connecting functions in local with those in remote. This data consists of the corresponding function call arguments and return values, as well as the global state right before and after each remote call. The algorithm then computes the objective function $O$ for remote (line 9). The set of remote functions that maximises $O$ is eventually chosen for remote execution (lines 10–13).

3.4 RPC-based code offloading

We realise our approach in AnyWare, a system that transforms C applications for cloud-assisted execution. This section starts by providing a high-level overview of AnyWare’s architecture, describing each of its components separately and how they interact with each other. We then present AnyWare’s offline profiling module and finish by describing how it automatically rewrites application source code to realise a partitioning.

3.4.1 AnyWare architecture

We show the architecture of AnyWare in Figure 3.2. AnyWare initially receives the source code and binary of a C application, along with a set of driver scripts that automatically execute the application with common workloads. Its three main components are the Profiler, the Partitioner and the RPC compiler.
The Profiler executes the application with different input workloads, while profiling its runtime behaviour using DTrace. Combining the dynamic profiling information collected, along with information deduced by statically analysing the application’s source code, it creates the resource consumption graph described in Section 3.3.1. This graph is then passed to the Partitioner. The Partitioner applies the partitioning algorithm from Section 3.3.2 to partition the set of application functions into local and remote subsets. Finally, the RPC compiler implements the split at the source code level by making appropriate changes to convert local into remote function calls where needed, while handling global application state. The remote server and client versions of the source code are compiled, and the binaries are handed over to the corresponding local and remote nodes for execution.

3.4.2 Profiling tools

The Profiler’s objective is to gather the information required to generate an application’s resource consumption graph. Part of this information is generated by statically analysing the application source code. The rest is collected while executing the application using DTrace. This step is repeated for common application workloads in order to capture average runtime behaviour. Here, we describe ANYWARE’s static code analyser, as well as the custom DTrace programs used for dynamic profiling.

Static analyser

Some aspects of an application’s behaviour are independent of the workload executed and thus could be inferred by statically analysing the application source code. Using static code analysis, we avoid the unnecessary overhead associated with collecting information that is constant across all profiling runs at runtime.

More specifically, the Profiler uses static analysis to identify the amount of fixed-sized data passed to and returned from application functions in the form of non-pointer type function arguments and return values. In addition, it computes the sum of sizes of all non-pointer type global variables. This amounts to a fixed-sized portion of the global application state that needs to be transmitted to the remote node with a remote function call. What is left is also accounting for the application state that is dynamically allocated during execution. This, however, requires dynamic profiling because it depends on the application workload and thus cannot be known a priori.

Given a list of all library functions that implement I/O and device-specific functionality (created manually once per platform), the Profiler uses static code analysis to identify which functions directly call into any of these native functions. This information is used to connect the corresponding functions with the special node native in the application’s resource consumption graph, thus preventing them from being chosen for remote execution.

Finally, to avoid having to traverse complex data structures with pointer-typed fields when passing arguments to remote calls, and therefore eliminating assumptions regarding the type-
safe nature of application code, we enforce an additional partitioning constraint: functions that accept pointers to such data structures as arguments cannot be offloaded to the remote node. These functions are identified automatically by compiling a list of all data structures that contain pointer-typed fields, which are then checked against the types of arguments accepted by each function before partitioning. For functions that accept such arguments, an edge connecting them to native is added to the resource consumption graph.

As opposed to a more rigorous static analysis tool that employs more advanced compiler techniques, for example lexical analysis, preprocessing, parsing and semantic analysis, ANY-WARE’s static analyser comprises simple scripts that implement string matching algorithms based on regular expressions. These are used to identify string patterns within the larger application source files according to the description above. The downside of this approach is that some cases of complex code, for example, code that contains preprocessor directives, which are resolved before the compilation of code, may not be correctly accounted for. For the purposes of this work, we settle for the simplicity of a string matching approach. Our focus are the challenges of dynamic profiling and automatic transformation of unmanaged applications to realise partitioned execution.

DTrace programs

The DTrace tracing programs used for dynamic profiling are written in a C-like programming language called the D programming language. A typical D program consists of a list of one or more instrumentation points, otherwise referred to as probes. Each probe is associated with a condition and an action, which is executed whenever the condition is met. Examples of a probe firing would be the starting of a process or the opening of a file. The actions taken when a probe fires allow for monitoring and analysing the runtime status of an application. For example, DTrace allows access to the call stack and context variables. It also can keep track of statistics on the application’s resource usage during execution.

The Profiler is given a set of driver scripts to run the application under identified workloads. While for command-line applications, generating such scripts is a trivial task, it is more complicated for applications with a GUI. In this case, tools such as Apple’s Automator [App11] and GNU Xnee [Lab12] can be used to record and replay user actions to execute the GUI application automatically.

CPU profiling. Algorithm 2 shows the pseudo-code for the D program used to generate CPU-related data. At a high-level, it records the times of all function call invocations and returns for each application thread. These are used to calculate the total time each user function spends executing on the CPU, as well as function call dependencies.

Probes 1 and 2 fire when a user function probefunc starts and terminates execution, respectively. DTrace supports a number of built-in variables to assist tracing a running system. One of these variables is utimestamp, which gives the current value of a nanosecond timestamp counter. It corresponds to the amount of time that the current thread has been running on
Algorithm 2: Profile CPU usage

**input**: A process id (pid)

**output**: A log file of CPU profiling information

1. **Probe 1** User function (probefunc): entry
   
   Log(vtimestamp, tid, “Entry”, probefunc)

2. **Probe 2** User function (probefunc): return
   
   Log(vtimestamp, tid, “Return”, probefunc)

3. **Probe 3** Process pid: end
   
   Ouptut log file

Algorithm 3: Profile main memory usage

**input**: A process id (pid)

**output**: A log file of memory profiling information

1. **Probe 1** Memory allocation (malloc(), calloc(), realloc()): return
   
   ⇒ input parameter: size
   
   ⇒ return value: base_addr

2. Log(vtimestamp, “New”, base_addr, size, ufunc(ucaller))

3. **Probe 2** Memory deallocation (free()): entry
   
   ⇒ input parameter: base_addr

4. Log(vtimestamp, “Free”, base_addr, ufunc(ucaller))

5. **Probe 3**: Process pid: end

6. Ouptut log file

The CPU minus the time spent in DTrace predicates and actions. tid is another such variable that reports the thread ID of the executing thread. In line 2, the vtimestamp and tid values are logged for each “dynamic” function that starts execution, along with the corresponding condition type (“Entry”) and the function’s name (probefunc). We distinguish between “dynamic” and “static” functions because a static function may be instantiated many times in parallel, such as in the case of a recursive function. Similarly, when a user function returns, i.e. Probe 2 fires, the vtimestamp and tid values are logged, along with a different condition type (“Return”) and the function’s name (line 4).

Finally, Probe 3 fires when the application is about to exit. All the data gathered during execution is output (line 6) to be used for constructing the resource consumption graph. Given the per-thread ordered sequence of function invocations and returns, along with the actual timings of each such event, the Profiler is able to calculate: (i) the CPU times for each user function (i.e. per function CPU times excluding time spent in nested function calls); and (ii) the function call graph of each profiling run.

**Memory profiling.** Algorithm 3 describes the D program that generates the memory-
related data, which is used to estimate the amount of data transfers each partitioning entails. We assume that with any remote call invocation, all active dynamically allocated memory needs to be transmitted to the remote node as it may be accessed during remote execution. Similarly, when returning from a remote call, the same state needs to be transmitted back to the caller partition.

Therefore, the D-program in Algorithm 3 intercepts all dynamic memory allocations and deallocations in order to log the sizes and base addresses of each memory region created or destroyed at runtime. This information is used to compute the total amount of active dynamic memory right before and after each function call. Probe 1 fires when either a \texttt{malloc()}, \texttt{calloc()} or \texttt{realloc()} function, i.e. any system call that dynamically allocates memory at runtime, is about to return. In line 4, for each such memory allocation we log: (i) the current timestamp (\texttt{vtimestamp}); (ii) a condition type identifying the creation of a new memory region (“New”); (iii) the base address of the newly allocated memory region (\texttt{base_addr}), as returned by all functions; (iv) the size of the memory region (\texttt{size}), as given as input to each function; and (v) the caller user function responsible for the new memory allocation (\texttt{ufunc(ucaller)}). \texttt{ucaller} is a built-in DTrace variable that provides the program counter location of the current user thread when a probe fires, while \texttt{ufunc()} is a DTrace function that returns the function name given a program counter.

Attempts to free dynamically allocated memory are intercepted by Probe 2. Here (line 7), we log: (i) the current timestamp; (ii) a condition type identifying the deallocation of a memory region (“Free”); (iii) the corresponding base memory address (\texttt{base_addr}), as given as input to the call; and (iv) the name of the user function responsible for freeing the memory region (\texttt{ufunc(ucaller)}).

Probe 3 fires when the application is about to exit. The data collected at runtime is output (line 9). This data is used in a post-processing step, along with the CPU-related data and the information obtained via static code analysis, to calculate the amount of data that would have been transferred over the network for each function call if it were converted to a remote call.

3.4.3 RPC compiler

The RPC compiler implements the split of the original application into two partitions for the local and remote nodes, respectively. This is done automatically by inserting code at key points of the source code to convert local into remote calls. This section describes how \textsc{AnyWare} implements a partitioning for C applications.

RPC framework

We use the CRPC framework [Bab09] for marshalling and unmarshalling remote function parameters and implementing remote function calls. CRPC is an open source RPC system
with C language support. It extends the standard C language with new modifiers to declare client- and server-side functions. CRPC includes a wrapper-compiler that supports C base and custom data types. It allows sending data referenced by pointers of any type with remote calls implemented on top of the TCP transport protocol.

Injecting RPC calls

Using the pseudo-code in Algorithm 4, we describe the process of splitting a C application into two parts. This process requires the application source code (source code), the two sets of local and remote functions (local and remote) and a list of global application variables (globals) derived from static code analysis. In the remainder of this section, we refer to *globals as the subset of globals that includes all pointer-typed variables, and ¬* globals as the subset of non-pointer-typed global variables.

Initially, the application source code is duplicated in line 1: the two copies, local_source_code and remote_source_code, correspond to the source code of the programs intended for the local and remote nodes, respectively. Lines 2–21 describe how local_source_code is transformed according to a given application partitioning. For each function to be executed remotely, a new function remote_function is declared. The function remote_function is used to transfer control of execution to the remote node when function is invoked. It has the same return type and accepts the same input parameters as function, with the addition of pointers to the contents of globals and the sizes of all of remote_function’s pointer-typed parameters (lines 5–7). These sizes are obtained at runtime by intercepting all function calls that are responsible for dynamic memory allocation and deallocation, namely calls to malloc, calloc, realloc and free, and associating the base addresses returned with the corresponding memory sizes using a global hash table named pointer_to_size.

The __remote modifier informs the CRPC wrapper-compiler that this is a function intended for remote execution (line 8). Furthermore, __attribute__((__format_ptr(X[Y]))) is used to inform the CRPC wrapper-compiler about the appropriate size parameters associated with pointer-typed function arguments (line 9). This information is used to send copies of variable-length parameters that are allocated in consecutive memory locations, for example dynamically allocated arrays, to the remote node. Attribute parameters are written in an array-like notation. For example, “X[Y]” denotes that the size for the pointer at position X is stored in the variable at position Y. Finally, an appropriate call to remote_function is used to substitute function’s original execution body (line 10).

Similar steps are taken for declaring remote_function in remote_source_code to manage incoming calls on the remote node (lines 32–37). Instead of the __remote modifier, however, a new storage class qualifier __local is used to denote functions that are invoked from within a different node. The RPC compiler also provides a definition for this new function in remote_source_code (lines 38–42). All pointer-typed global variables that may be accessed by remote_function, or any other function it calls, first need to be made to point to the
Algorithm 4: Split application using RPC

input: An application source code (source_code), a set of local functions (local), a set of remote functions (remote) and a set of global variables (globals)

output: A local and a remote code partition

1. Create two copies of source_code: local_source_code and remote_source_code
2. Local partition: local_source_code
   - foreach function in remote_code do
     - Declare new function remote_function:
       - return type: function's return type
       - parameters: function's parameters + *globals + pointers to ¬* globals + sizes of all pointer-typed parameters
       - add __remote modificator before declaration
       - use __attribute__((__format_ptr)) to associate parameters with sizes
     - Substitute function's body with call to remote_function
   - foreach function in local do
     - Declare new function local_function:
       - return type: function's return type
       - parameters: function's parameters + *globals + pointers to ¬* globals + sizes of all pointer-typed parameters
       - add the __local modificator before declaration
     - Define local_function as follows:
       - Assign parameters to corresponding *globals variables
       - Assign values pointed at by parameters to corresponding ¬* globals variables
       - Call function with appropriate parameters
       - Replace values pointed at by parameters with corresponding ¬* globals values

Remote partition: remote_source_code
3. Remove main() function
4. foreach function in local do
   - Declare new function local_function:
     - return type: function's return type
     - parameters: function's parameters + *globals + pointers to ¬* globals + sizes of all pointer-typed parameters
     - add __remote modificator before declaration
     - use __attribute__((__format_ptr)) to associate parameters with sizes
   - Substitute function's body with call to local_function

5. foreach function in remote do
   - Declare new function remote_function:
     - return type: function's return type
     - parameters: function's parameters + *globals + pointers to ¬* globals + sizes of all pointer-typed parameters
     - add the __local modificator before declaration
   - Define remote_function as follows:
     - Assign parameters to corresponding *globals variables
     - Assign values pointed at by parameters to corresponding ¬* globals variables
     - Call function with appropriate parameters
     - Replace values pointed at by parameters with corresponding ¬* globals values
corresponding arguments of the remote function call (line 39). Furthermore, non-pointer-typed global variables are assigned the values referenced by the corresponding arguments in \( \neg \ast \) \textit{globals} (line 40). Next, a call to \textit{function} with the appropriate input parameters is added (line 41). The final step involves updating all values of non-pointer-typed global variables on the local node according to changes that occurred during remote execution (line 42).

Figure 3.3 shows an example of a partitioned function according to the steps described in Algorithm 4. The \textit{function} accepts one argument (\textit{int arg1}) and returns an integer value. Furthermore, we assume two global variables, i.e. \textit{int global1} and \textit{dt \ast global2} (\textit{dt} is an application data structure), which may be accessed by both the local and remote partitions. Therefore, \textit{remote\_function} accepts three more parameters, i.e. pointers to \textit{global1} and \textit{global2}, as well as the size of the allocated memory pointed at by \textit{global2} (line 3), which transfer global state to the remote node and back. On the local node, a call to \textit{function} is relayed to the remote node via a corresponding call to \textit{remote\_function} (line 6). First, however, the size of \textit{global2} is obtained using the \textit{pointer\_to\_size} hash table, which maps pointers to the corresponding sizes of dynamically allocated memory (line 5).

On the remote node, incoming calls to \textit{remote\_function} first initialise the global application state according to the corresponding function parameters (lines 11 and 12), then invoke the original \textit{function} (line 13) and finally ensure that global state on the local node is updated after completion by updating parameters that point to non-pointer-typed global variables before returning (line 14).

So far, we have described how functions in \textit{remote} are implemented using the CRPC framework. In the reverse direction, implementing remote function calls originating from the remote partition (i.e. functions in \textit{local}) is a symmetrical process, described in lines 11–31 of Algorithm 4. Note that the \texttt{main} function in \texttt{remote\_source\_code} is removed, since the CRPC wrapper-compiler adds its own (line 23). Finally, both versions of the original source code, i.e. \texttt{local\_source\_code} and \texttt{remote\_source\_code}, are compiled against the CRPC wrapper-compiler. This extends a program’s source code with network-specific code for the marshalling and unmarshalling of parameters, as well as implementing remote calls with RPCs. Its output is an ordinary C program, which is then compiled with the GCC compiler.
3.5 Evaluation

We evaluate our approach on a complex open-source application, the Transmission BitTorrent client [Tra05], to show that ANYWARE reduces significantly application response times under severe CPU constraints. We further characterise how performance improvements are affected by available CPU resources on the local node.

We begin this section by explaining our choice of application and describing the experimental set-up. We then present and analyse the results of our experiments.

3.5.1 Experimental set-up

Transmission BitTorrent client

Transmission is an open-source, cross-platform, peer-to-peer (P2P) BitTorrent client written in the C programming language. For our experiments we use the Transmission GTK+ client version 1.92, which is described in more detail below:

1. Transmission is a complex C application and consists of approximately 74,690 lines of code, split over 336 source files.

2. It contains an advanced GUI realised using the GTK+ toolkit, in addition to a non-trivial implementation of the BitTorrent P2P protocol, including features such as encryption, peer exchange, magnet links, DHT, UPnP and NAT port forwarding.

3. It has a significant resource footprint—after starting, it allocates 13 MBytes of memory before beginning any downloads.

4. It has few dependencies to other third-party libraries except for the GTK+ and X11 toolkits.

While a BitTorrent client is not an ideal candidate for evaluating the effectiveness of ANYWARE when compared to more compute-intensive applications such as image, audio and video processing clients, our choice of application was constrained by the limited availability of non-trivial open-source C applications. Nevertheless, showing that ANYWARE can benefit even applications with lesser CPU requirements testifies to the applicability of our compute-focused approach for a wide range of applications.

Experiment test-bed

We run our experiments using two identical Intel Core 2 Duo processor machines at 2.26 GHz and 2 GB of DDR3 memory. The machines emulate a networked resource-constrained device and a more powerful remote node, respectively. The network round trip time (RTT) between
the two machines throughout our experiments is on average 4.4 ms over an IEEE 802.11g WiFi network.

To emulate a resource-constrained device and have control over its CPU resources we use cputhrottle [No10], a Mac OS X command-line utility that limits the CPU usage of a process. Cputhrottle accepts two inputs, a process ID and a maximum percentage of the available CPU this process is allowed to consume. It makes use of the task_info, task_threads, task_suspend, and task_resume system calls to enforce CPU throttling. The first two collect CPU usage statistics on the process, whereas the last two suspend and resume the process accordingly, so as to constrain its CPU usage to the maximum percentage specified. With cputhrottle, we effectively manage to reproduce the effect of executing a process on a less performing CPU, therefore emulating the various types of CPUs in low-end mobile devices.

**Profiling workloads**

To dynamically profile Transmission’s runtime behaviour, we consider the following three typical BitTorrent client workloads:

- **W₁:** Launch the application, add two new torrent files for download, wait until both downloads complete and exit.
- **W₂:** Launch the application, add two new torrent files for download, pause both downloads, exit, relaunch the application, resume both downloads and exit on completion.
- **W₃:** Launch the application, add two new torrent files for download, browse through metadata information for both downloads and exit.

The above workloads cover actions an average user may perform when using Transmission, i.e. downloading torrent files, resuming downloads after exiting the application (existing data is checked before a download resumes) and finally, inspecting downloads by browsing the corresponding options offered by the application.

**3.5.2 Performance analysis**

Profiling Transmission under the aforementioned workloads identifies 12 application functions to be executed remotely. The bulk of these functions handle data encryption, distributed hash table management and binary encoding functionality, all of which are computationally intensive and relatively separate from the rest of the application’s functionality.

For the local node, we pick three different maximum CPU percentages to use with cputhrottle, emulating devices with low, moderate and high CPU capabilities, respectively. We set the maximum CPU percentage to 6%, 12% and 18%, based on the observation that the unmodified Transmission application (when run without cputhrottle limiting its CPU consumption) consumes, on average, approximately 18% of the available CPU.
We first run the unpartitioned Transmission application on the local node, using cputhrottle with all three different configurations. We measure the time to download two torrent files of 290 MB each. We ensure that the network conditions remain the same throughout all experiments by saturating the network link, after limiting bandwidth to a maximum of 750 KBps. The same experiments are repeated using the partitioned version of the application. During each experiment we also monitor CPU consumption over time.

Figure 3.4 shows the time taken for the two downloads to complete, for both the unpartitioned and partitioned versions of Transmission and different CPU throttling. When restricting CPU usage to 6% (low) on the local node, the partitioned version significantly outperforms the unpartitioned version of Transmission by downloading data amounting to 580 MB approximately 4.5 minutes faster, i.e. achieving a speedup of $1.32 \times$.

As expected, increasing the CPU power available on the local node gradually reduces performance gains. Computation on the local node has less to gain from being offloaded to the remote node, i.e. the gap between local and remote compute resources becomes less significant. Therefore, for a limit of 12% on the overall CPU usage locally, we observe a speedup of $1.16 \times$. Finally, with a limit of 18% on CPU usage (i.e. Transmission’s average CPU utilisation without CPU throttling), the partitioned version is outperformed marginally by the unpartitioned version due to the additional offloading overhead. This overhead is caused by RPC delays and the cost of transmitting data over the network.

The benchmark in Figure 3.5 presents the CPU consumption on the local node for both the unpartitioned and partitioned versions of Transmission. For this experiment, we use cputhrottle with the low CPU capabilities setting (i.e. a maximum of 6% of the total CPU usage is available locally). As shown in the graph, approximately 35 seconds into the experiment, we add the two torrent files to be downloaded, wait for 1 minute and then pause both downloads. A minute later, we resume both downloads and wait until they complete. The CPU utilisation throughout the experiment for both versions of the application remains at 6% when downloading (full CPU utilisation) and drops to zero when paused or completed. When the downloads are paused, it takes slightly longer for the unpartitioned version to drop to zero due to its greater local workload. Overall, the entire files are downloaded approximately 4 minutes earlier by the partitioned version because it is able to offload compute-intensive
functions to speedup execution.

3.6 Discussion

This section states the current limitations of our compute-focused offloading approach, which are left for future work. First, we assume that network connectivity persists between the local and remote nodes throughout execution, i.e. our approach currently does not mask network failures. As explained in Section 3.2.5, a simple solution to this challenge would require repeating a remote call locally after failure. However, the fact that C does not support exception handling natively complicates matters, requiring the use of third-party libraries that implement exceptions and their semantics. This would allow to react to a failed remote call before the application is terminated.

Second, we currently support only single-threaded applications. With multiple threads, additional challenges arise associated with how application state is managed across the local and remote nodes during remote execution. With AnyWARE, every offloaded function call requires the transmission of a copy of the global application state to the node about to execute the function. For single-threaded applications, code is guaranteed to execute only on one of the two partitions at any point in time. With multiple threads, this is not the case because different threads may execute on both the local and remote nodes in parallel. Therefore, both active copies of the application state could be modified concurrently. This may lead to inconsistencies and thus requires efficient synchronisation to ensure that both copies can co-exist. A challenge is to ensure that any impact on performance caused by such synchronisation techniques does not outweigh the performance gains achieved by code offloading.

3.7 Summary

In this chapter, we have described a code offloading approach that exploits the more powerful compute resources of remote nodes for increased application performance. This app-
Compute-Focused Offloading

approach statically partitions applications to execute across a local and a remote node based on fine-grained profiling information. This information is collected by dynamically profiling application execution under different workloads. The profiling requires no modifications to the application and yields a partitioning that reduces application response times.

Unlike existing approaches, we focus on providing support for applications written in unmanaged programming languages with minimal runtime support, targeting low-end mobile devices with severe CPU constraints. Contrary to dynamic code offloading solutions, we avoid the runtime overheads of continuously monitoring resource consumption on the mobile device and the changes in execution environment for making offloading decisions at runtime. In doing so, we give up the ability to react to runtime changes on the fly. A static approach, however, is more suitable for unmanaged applications running on low-end devices.

We have realised our approach with AnyWare, a system that automatically partitions C applications to overcome performance limitations by leveraging faster remote compute resources. AnyWare performs scenario-based, dynamic profiling of applications using DTrace and static code analysis to obtain information regarding the application’s runtime behaviour. It models this information using resource consumption graphs, decides on a partitioning that offloads compute-intensive functions to a remote node and, finally, implements the partitioning with minor source code changes using RPCs. We evaluated AnyWare on a complex, open-source application and showed that it is capable of achieving a speedup of $1.32\times$ in execution time in an environment with severe CPU constraints.
Chapter 4

Memory-Focused Offloading

In this chapter, we describe a static code offloading approach that exploits the larger main memory of cloud-based resources in conjunction with their faster CPUs. It alleviates device memory constraints, currently restricting an application’s memory usage to the memory available on mobile devices. In addition, it improves the performance gains achieved by current offloading approaches by collocating computation and application data, thus eliminating runtime overheads associated with repeated state migration.

4.1 Overview

While current offloading approaches only focus on exploiting the compute resources of remote nodes to compensate for the lack of adequate processing power on a mobile device, the latter’s limited availability of main memory often becomes a bottleneck to the functionality of applications too [Cra13]. For example, while Apple’s iPhone 4S smartphone has 512 MB of physical memory, iOS applications are only left with 213 MB of usable main memory—the difference being reserved for the operating system. With no kernel-level virtual memory mechanism in place, applications that consume more than this amount of memory are forcibly terminated by iOS. Considering that a single 8-megapixel photo has over 30 MB of bitmap data, this translates to an image processing application being terminated for having just seven photos resident in memory.

Today’s code offloading systems cannot overcome memory limitations on mobile devices because they perform application state migration: state primarily resides on the local node, and, for each offloaded function call, parts of the application state are transferred to the remote node for computation and then migrated back after the computation has finished. This, however, has the benefit of simplifying failure handling after network connectivity between the local and remote node was lost—a common occurrence in mobile networks. Since the remote node only maintains “soft” state, execution can simply fall back to the local node immediately after failure.

Apart from restricting the total application state to the available device memory, repeated
Memory-Focused Offloading

State migration is also wasteful: potentially the same state has to be serialised, transmitted over the network and deserialised multiple times, for repeated offloaded function calls. As we show in Section 4.5, these overheads may reduce the gains in application performance due to code offloading, in some cases offsetting them completely.

In contrast, we propose to partition application state permanently between the local and remote nodes, which has two main advantages: (1) it enables an application’s memory usage to exceed the total main memory of the local node; and (2) it means that offloaded calls transmit less data because they can reuse already available state on the remote node. State partitioning is therefore most useful for applications with a large memory footprint or for interactive applications, which carry out many repeated offloaded calls based on user input, each accessing the same state. A major challenge, however, is that an offloading system with partitioned state must continue execution after access to the remote state was lost due to intermittent network connectivity. In addition, state partitioning should only require modifications to the mobile application and not the underlying platform or operating system. In particular, an approach that requires changes to the OS kernel would see low adoption on established mobile platforms such as iOS or Android.

While we retain our goal of supporting applications written in unmanaged programming languages, we now assume applications that conform to the object-oriented programming paradigm. Since objects encapsulate state with associated computation, this decision is inline with our design choice to partition application state between the local and remote nodes and distribute computation accordingly. To this end, we have chosen to work with Objective-C applications on the iOS mobile platform. As in the case of AnyWare, we employ a static partitioning approach, which is more suited for partitioning unmanaged applications and avoids the overheads associated with making offloading decisions at runtime.

The contributions of this approach are summarised below:

1. **Application state partitioning:** We use an optimisation-based partitioning algorithm that splits application state between the local and remote nodes. This is based on offline profiling of the application under common workloads that capture its average runtime behaviour. Each object is placed permanently either on the local or on the remote node, such that (i) compute-intensive application functionality is executed remotely and (ii) the local node’s main memory is never exhausted at runtime. Access to remote objects is supported transparently via proxy objects, which relay invocations using RPCs.

2. **Snapshot-based fault tolerance.** We propose a new fault tolerance mechanism for recovering remote application state locally, after losing network connectivity to the remote node. This mechanism is based on consistent snapshots of changes to the local and remote application state, which are taken and stored periodically on the local node. A synchronous strategy takes snapshots after each offloaded call, therefore allowing failed calls to simply be re-executed locally. For more data-intensive applications, an
asynchronous strategy permits more sporadic snapshots, thus reducing the associated snapshotting overheads. However, after failure, the application state may require rolling back to the last complete snapshot available on the local node, thus resuming execution from an earlier point in time. This is all achieved without modifications to iOS or the Objective-C language.

3. **User-level virtual memory.** After network failure, we must support application state sizes that are larger than the available memory on the local node. Since iOS does not offer a kernel-level virtual memory mechanism with on-demand paging for applications, we implement a simple virtual memory scheme at the user level. Objects from remote state snapshots remain stored in flash memory and are loaded into main memory when accessed by the application.

We realise our approach in **CloudSplit**, a system that statically partitions Objective-C applications on the unmodified iOS platform to benefit from cloud-assisted execution with state partitioning. We evaluate a prototype implementation of **CloudSplit** using two real-world iOS applications, a board game and a spreadsheet application. We show that, by partitioning application state, **CloudSplit** can support large memory sizes and reduce response times by up to $15 \times$ compared to conventional state migration approaches. Our snapshot-based fault tolerance mechanism incurs only a modest overhead and supports a choice between reducing network usage or the amount of lost application state. After failure, local execution exhibits only a 25% performance degradation caused by the on-demand loading of objects from flash memory as part of our user-level virtual memory scheme.

Next we discuss the design space for code offloading systems, motivating and contrasting our approach. Section 4.3 presents the **CloudSplit** design, with a focus on application state partitioning. Section 4.4 describes the snapshot-based fault tolerance mechanism, as well as the user-level virtual memory scheme. We present evaluation results in Section 4.5 and state the current limitations of this approach in Section 4.6. We finally summarise this chapter in Section 4.7.

### 4.2 Design space

We compare our memory-focused approach with existing offloading approaches in Table 4.1. In some aspects of the design space taxonomy provided, the goals set for **CloudSplit** follow those met by AnyWare. Here, we focus on the relevant advances of offloading with application state partitioning over our compute-focused approach and discuss only the additional features that this new code offloading paradigm entails.
Memory-Focused Offloading

4.2.1 Optimisation goals

Similar to AnyWare, CloudSplit aims to reduce application response times, which typically also implies reductions in energy expended on the mobile device [CIM+11]. In addition, however, CloudSplit also provides support for memory-intensive applications, in a way that exploits the larger amount of memory available at a remote node. Application workloads that would otherwise exhaust the device’s main memory can be executed, since CloudSplit partitions application state between the local and remote nodes accordingly.

4.2.2 Application state migration vs. partitioning

Current offloading systems, including MAUI, ThinkAir and CloneCloud, do not partition application state permanently but always keep the entire state on the local node. This restricts their total memory consumption to the available device memory, preventing them from supporting new types of memory-intensive mobile applications. Repeated state migration during offloading also becomes an issue for interactive applications such as games and business applications, which perform many repeated function calls that e.g. update a large game state or edit a document in response to user input. During offloading, the serialisation, transmission and deserialisation of the same state degrades performance.

To reduce the overhead of state migration, COMET uses DSM to maintain replicas of the state and propagates only changes when offloading. This relies on support for DSM by the runtime system, which is unavailable under iOS, and suffers from the intrinsic cost of DSM.

In contrast, CloudSplit is designed with state partitioning in mind. Applications can maintain a large amount of state on the remote server, without being restricted by the memory capacity of the mobile device. A virtual memory scheme in CloudSplit enables applications with a large memory footprint to continue executing locally after network failure.

4.2.3 Partitioning granularity

Unlike current approaches that partition applications at the granularity of function calls or even individual statements, CloudSplit partitions applications at a class level because classes encapsulate application state with associated computation. Although a finer partitioning granularity provides flexibility, this design choice avoids the need to identify state
associated with object methods or splitting the state of objects. The placement of a class is
determined by the combined behaviour of its methods.

4.2.4 Fault tolerance

Intermittent network connectivity is typical in mobile environments. Any code offloading
system must therefore handle network failures gracefully, with low impact on application
execution. We realise that the major limitation of AnyWare, compared to existing code
offloading solutions, is the fact that it currently ignores network failures by assuming strong
network connectivity between the local and remote nodes for the entire session of the partitioned
application.

Due to the partitioned application state, achieving fault tolerance is more challenging for
CloudSplit than existing approaches, which simply need to re-execute a failed remote call
locally—the entire application state is kept on the local node at all times. With CloudSplit,
part of the application state is maintained remotely, therefore, loss of connectivity makes it
inaccessible to the local node. For this reason, CloudSplit employs a new snapshot-based
fault tolerance mechanism to recover missing state after failure (see Section 4.4).

4.3 Application state partitioning

Applications executed under CloudSplit eventually undergo automatic source code rewrit-
ing for partitioning. Objective-C programs are composed of a set of classes. Each application
class contains a number of fields and methods, which access these fields. Classes are instan-
tiated as objects at runtime, which combined together constitute the application state.

CloudSplit partitions applications at a class granularity, i.e. all objects of the same class
are assigned to a given partition. By distributing objects among the local and remote nodes,
CloudSplit effectively partitions application state while also distributing computation.

As illustrated in Figure 4.1, CloudSplit comprises three main components: (1) The Pro-
filer collects information about an application’s CPU and memory consumption. (2) This
information is used by the Partitioner to derive a partitioning that improves application
response times and respects device memory constraints. (3) The Compiler then realises the
partitioning by rewriting the application’s source code accordingly.

4.3.1 Dynamic resource profiling

CloudSplit uses offline dynamic profiling, combined with static code analysis, to measure
the resource consumption (in terms of CPU and memory usage) and to identify the call
dependencies of an application under a set of workloads. The Profiler classifies application
classes with respect to the amount of resources that they consume. It collects information
about (i) the average execution times of methods on both the local and remote nodes and
Figure 4.1: Architecture of the CloudSplit code offloading system.

(ii) the amount of memory used by objects of each class. As this process resembles in many aspects the profiling techniques used in AnyWare, in this section we mostly focus on the parts that differ from them.

CloudSplit dynamically profiles each application twice; a local profiling run $lp$ gives the application’s performance before partitioning; a remote profiling run $rp$ executes the application using a device emulator on the remote node. This provides an upper performance bound that a partitioning may achieve when the whole application executes remotely, ignoring communication overheads.

Profiling methodology

The Profiler uses three techniques to obtain profiling information. First, just as in the case of AnyWare, it identifies objects’ interactions with platform-native functionality, such as GUI libraries or the hardware sensors, as well as the size of fixed-sized arguments and return values of function calls, using static code analysis.

Second, it automatically adds instrumentation to the application’s source code to measure the duration of each method. To correctly account for the time spent in nested method invocations, it uses a per-thread global stack to record durations. At the start of each
method, the contents of the call stack are used to construct a class call graph, which is output on completion of a profiling run. While for AnyWare this information is collected using DTrace, this was not an option for CloudSplit due to limitations of the iOS platform. In particular, iOS does not allow for custom DTrace programs to attach to a running program during a local profiling run.

Finally, as part of the remote profiling run, the Profiler utilizes DTrace to collect information about the memory consumption of objects.

Profile output

The output of each profiling run \( p \) is a resource consumption graph, \( G = (C, T^L, T^R, E) \), where \( C \) is a set of application classes, \( T^L \) and \( T^R \) provide execution times for classes and \( E \) specifies call dependencies. For a class \( x \in C \), \( T^L(x) \) and \( T^R(x) \) return the average execution times of class \( x \) on the local and remote nodes, respectively. \( E(x, y) \) is a pair \((\text{calls}_{x \rightarrow y}, \text{data}_{x \rightarrow y})\) that states that an average of \( \text{calls}_{x \rightarrow y} \) calls from methods in class \( x \) to \( y \) occurred; each call used an average of \( \text{data}_{x \rightarrow y} \) bytes in its arguments.

The Profiler also uses DTrace to output a memory consumption relation \( M \). Given a class \( x \in C \) and a time \( t \), \( M(x, t) \) denotes the amount of memory consumed by all objects of class \( x \) at \( t \). This information is collected by intercepting all memory allocations and deallocations at runtime, as described for AnyWare, and is used to ensure that the total amount of memory consumed by objects assigned to the local node does not exceed its capacity.

In Figure 4.2, we show an example memory consumption relation, plotted as a time series graph of the amount of memory consumed by each of an application’s classes during execution. Given a candidate partitioning, adding the y-axis values of all classes placed on the local node, for all timestamps, provides evidence of the local partition’s demands in memory throughout execution. In this example, assuming the local node is an iPhone 4s smartphone (with just 213 MB of main memory left to the application), a partitioning that places all four classes on the local node would be discarded, as it would have led to the application being terminated by iOS at time \( t \). This is because all four classes collectively utilize 300 MB of main memory (60 MB for class A objects, 160 MB for class B objects, 50 MB for class C objects and 30 MB for class C objects), which is more than the available memory on the local node.

4.3.2 Partitioning algorithm

Based on the Profiler output, the Partitioner decides how to partition classes so that the overall execution time is minimised, while considering the remote communication overhead. Each class is assigned to a local or remote set, which contain the classes hosted by the local and the remote node, respectively. A valid partitioning must satisfy the constraints that (1) a class belongs either to local or remote, but not both; (2) the local node’s memory capacity is sufficient to accommodate all local objects. (We assume that the remote node’s capacity is
Memory-Focused Offloading

Figure 4.2: Memory consumption per application class using DTrace.

effectively unlimited); and (3) classes in remote do not call native library functions that are only available on the local node.

Let $V$ be the set of all valid partitionings, i.e. $V$ contains pairs of the form $v = (\text{local, remote})$ that satisfy the above constraints. We use a function $T_{\text{net}}(x)$ to express the time needed to make a remote call over a network $\text{net}$, such as WiFi or 3G, with $x$ bytes for arguments. $T_{\text{net}}$ is derived experimentally by benchmarking the execution time of remote calls with different arguments.

The Partitioner’s objective is to select a partitioning $v \in V$ that minimises overall execution time $O$:

$$O(v) = \sum_{x \in \text{local}} T^L(x) + \sum_{y \in \text{remote}} T^R(y) + \sum_{x \in \text{local} \land y \in \text{remote}} \left( \text{calls}_{x\rightarrow y} \times T_{\text{net}}(\text{data}_{x\rightarrow y}) + \text{calls}_{y\rightarrow x} \times T_{\text{net}}(\text{data}_{y\rightarrow x}) \right)$$

The execution time $O$ for a partitioning $v$ is the sum of: (i) the execution time on the local node, i.e. the sum of all $T^L(x)$ for classes in local; (ii) the execution time on the remote node, i.e. the sum of all $T^R(y)$ for classes in remote; and (iii) the communication delay of the remote calls, caused by objects residing on different nodes. This is equal to the number of remote calls between any such pairs of classes, $\text{calls}_{x\rightarrow y}$, multiplied by the average delay of each call, $T_{\text{net}}(\text{data}_{x\rightarrow y})$.

The pseudo-code in Algorithm 5 describes how an application is partitioned given (i) its resource consumption graph $G$ and (ii) the type of network connectivity between the local and remote nodes $\text{net}$, e.g. WiFi or 3G. This algorithm computes the total execution times $O$ of all possible valid partitionings to choose the partitioning with the lowest such value, while respecting the local nodes memory limits.

Lines 5–9, check whether the given partitioning violates device memory constraints. Using
Algorithm 5: Partition application – Memory-focused approach

**input**: A resource consumption graph \((G)\) and a type of network connectivity \((\text{net})\)

**output**: A set of remote application classes

\[
\begin{align*}
\text{minimum}_O &= 0 \\
\text{best}_\text{remote} &= \emptyset \\
\text{foreach } \text{remote} \subseteq C - \{\text{native}\} \text{ do} & \\
\text{local} &= C - \text{remote} \\
\text{foreach } \text{rp} \text{ do} & \\
\text{foreach } t \text{ in } \text{rp} \text{ do} & \\
\text{local memory consumption} &= \sum_{x \in \text{local}} M(x,t) \\
\text{if } \text{local memory consumption} > \text{MEMORY\_CAPACITY} \text{ then} & \\
\text{go to } 3 & \\
\text{remote execution time} &= \sum_{x \in \text{remote}} T^R(x) \\
\text{local execution time} &= \sum_{x \in \text{local}} T^L(x) \\
\text{offloading cost} &= \sum_{x \in \text{remote}, y \in \text{local}} \left( \text{calls}_{x \rightarrow y} \times T_{\text{net}}(\text{data}_{x \rightarrow y}) + \text{calls}_{y \rightarrow x} \times T_{\text{net}}(\text{data}_{y \rightarrow x}) \right) \\
O &= \text{remote execution time} + \text{local execution time} + \text{offloading cost} \\
\text{if } O < \text{minimum}_O \text{ then} & \\
\text{minimum}_O &= O \\
\text{best}_\text{remote} &= \text{remote} \\
\text{return } \text{best}_\text{remote}
\end{align*}
\]

the memory consumption relation \(M\), we check whether for any remote profiling run, the amount of main memory consumed by all local objects at any point during execution exceeds the local node’s capacity (lines 5–7). If this is the case, the current partitioning is discarded (lines 8–9). In lines 10–13, we compute the partitioning’s execution time \(O\). In the end, the partitioning that yields the lowest execution time (lines 14–16) is returned (line 17).

### 4.3.3 Application transformation

Next we describe how the Compiler transforms the Objective-C application to realise the partitioning output by the Partitioner. It splits the source code into a local and a remote code partition to be deployed on the local and remote nodes, respectively.

**Proxy objects**

Objects interact transparently across partitions using proxy objects. A proxy to a remote object relays method calls to the object residing on the other partition using remote procedure
calls. While the remote node executes a call, the caller thread blocks waiting for a response message before resuming execution. Proxy objects are small and their size is independent of the remote object state.

After state partitioning, an object either resides on a partition or is represented by a proxy object. As a consequence, pointers to objects passed as arguments to a remote call must be converted by the callee to the associated local or proxy objects. The same applies to pointers returned by remote calls, which must be converted accordingly by the caller.

To make these conversions, the Compiler assigns a unique object identifier to each object upon its creation, which is a pair of values: its memory address on the partition that it resides on and a boolean value identifying the partition to ensure uniqueness. Objects can be referenced in cross-partition interactions through identifier hash tables maintained on each code partition, which map identifiers to local and proxy objects. When processing RPC calls, the referred objects are retrieved based on their object identifiers.

**Source code rewriting**

The Compiler rewrites classes as shown in Figure 4.3. In this example, we consider two classes, A and B, that interact across partitions. Class A contains two methods, \textit{method1} and \textit{method2}, which accept one parameter each: pointers to class B and class A objects, respectively; \textit{method1} returns a pointer to a class B object and \textit{method2} returns a pointer to a class A object. The figure illustrates how class A is transformed into a local class and a proxy class, placed on the local and remote nodes, respectively.

\textit{Local classes} retain their original fields and methods (lines 1–4), with some modifications to handle incoming remote calls. Each method receives a corresponding \textit{entry point method} to serve incoming RPC requests. Given an object identifier, it retrieves required local and proxy objects from the identifier hash table and invokes the method of the local object (lines 5–13).
Class Allocator

1: RPC* requestRPC (string class_name) {
2:     switch(class_name) {
3:         case "A":
4:             A* obj = Allocate new class A object;
5:             RPC* rpc = Create new class RPC object for obj;
6:             return rpc;
7:         case "B":
8:             ...
9:    
10:    void release (int objID) {
11:        void* obj = get_proxy(objID);
12:        if (obj == NULL) obj = get_object(objID);
13:        obj.release();
14:    }
15:    
16:    void retain (int objID) {
17:        void* obj = get_proxy(objID);
18:        if (obj == NULL) obj = get_object(objID);
19:        obj.retain();
20:    }

Figure 4.4: Allocator class for managing objects life-cycle.

Proxy classes contain the object identifier (line 14) of the object residing on the other partition, and a reference rpc to an RPC object used to initiate remote calls (line 15). Our CLOUD-SPLIT prototype implementation uses the Internet Communications Engine (ICE) [Zer05], an object-oriented RPC toolkit with support for Objective-C. It automatically generates RPC classes with the same method signatures as the underlying classes to serialise and deserialise method parameters. In the proxy class, the two methods are replaced with wrappers that execute the remote calls. The remote calls are invoked through the RPC object, after translating method parameters with object pointers to the corresponding object identifiers (lines 18 and 25). Returned pointer values are translated to local (line 20) or proxy objects (line 26) using the identifier hash table before returning them to the caller.

Object life-cycle

In Objective-C, the lifetime of an object is managed by means of reference counting. NSObject is the root class of most Objective-C class hierarchies, through which objects inherit a basic interface to the runtime system. One of NSObject’s fields is retainCount and denotes the number of ownership claims on an object. When a new object is allocated, it is returned with a retainCount of one. When a method acquires ownership of an object, it calls its retain method (inherited from NSObject) to increment the retainCount; relinquishing ownership is done by calling the release method (also inherited from NSObject). When retainCount is zero, the Objective-C runtime system calls the object’s dealloc method to free the allocated memory.

The reference counting mechanism, however, is unaware that CLOUD-SPLIT distributes objects
Memory-Focused Offloading

across the local and remote nodes. To allocate and deallocate partitioned objects correctly, the Compiler therefore synthesises a new Allocator class, described in Figure 4.4. It provides a requestRPC method that, given a class name, allocates a new object on the remote partition and returns the associated RPC object (lines 1–8). This is then wrapped by the corresponding proxy class on the partition issuing the request. Calls to release and retain on local and proxy objects are intercepted and forwarded to the associated proxy and local objects across partitions, respectively. This is done via the corresponding methods provided by the Allocator class (lines 10–20).

4.4 Snapshot-based fault tolerance

Application state partitioning makes it harder to mask failures. When reverting execution to the local node after failure, it is necessary to recover the remote state that is now inaccessible. CLOUDSPLIT employs a fault tolerance mechanism, in which the local node periodically retrieves a snapshot of the remote state and uses it to resume execution after failure.

This raises four challenges: (i) how to take consistent snapshots across local and remote state; (ii) how frequently to take snapshots, balancing freshness with overhead; (iii) how to implement snapshots in Objective-C without changes to the runtime system; and (iv) how to handle remote state after failure that is larger than the available memory.

4.4.1 State snapshots

A complete snapshot of the application state consists of a local and a remote snapshot. A local snapshot contains information needed to resume execution from a consistent, well-defined execution point. It includes (a) all active objects on the heap; (b) the state of all machine registers, obtained using in-line assembly code; (c) the current user call-stack; and (d) a table with object identifiers for all local objects. The local snapshot is written as a byte array to flash memory. Execution can resume from a local snapshot by overwriting the allocated heap, machine registers and call-stack with the data from the snapshot.

A remote snapshot is a byte array of serialised remote objects, including: (a) their object identifiers; (b) the values of all primitive fields; and (c) the object identifiers of fields pointing to other objects. Previously remote objects can be recreated locally from a remote snapshot as follows: (1) primitive data fields are restored with the values from the snapshot; (2) fields with pointers to other previously remote objects are set to the corresponding object identifiers so that the objects can be loaded on-demand (see Section 4.4.4); and (3) fields with pointers to local objects (which used to point to proxy objects on the remote node) are assigned addresses of the corresponding local objects. The mapping uses the objects identifiers from the snapshot.
### 4.4.2 Snapshotting mechanisms

Local and remote snapshots must be taken at consistent execution points. **CloudSplit** supports two snapshotting strategies with different trade-offs: in a *synchronous* strategy, **CloudSplit** creates a remote snapshot after each remote call, transfers it to the local node and stores it in flash memory. This ensures that the local node always has an up-to-date copy of all remote objects. After a remote call has failed, it can simply be repeated locally by recreating the remote objects. The synchronous strategy guarantees no loss of remote state after failure but it is less efficient due to the frequent transmission of remote snapshots after each remote call, especially when calls modify a large amount of remote state.

To address this issue, **CloudSplit** also supports an *asynchronous* strategy that decouples the transmission of snapshots from remote calls and permits an arbitrary snapshotting frequency. Since the most recent remote snapshot available locally may be inconsistent with the current local state, after failure **CloudSplit** must roll back the application to an earlier point in time. With asynchronous snapshotting, users experience better performance due to fewer snapshot transmissions but may lose the most recent application updates after failure.

#### Synchronous strategy

A remote snapshot is taken when execution returns from the remote to the local node. A challenge is that remote calls may be *nested*, i.e. a remote call may in turn execute a new call to the local node. A local node is said to have *control of execution* when it has no remote calls in progress, otherwise execution is controlled by the remote node. Nested calls do not transfer control.

Figure 4.5 gives an example of synchronous snapshotting with a nested call and failure recovery. The local node initiates a remote call $R_1$ at time $t_3$, transferring control to the remote
node. The remote node triggers a nested call \( L_1 \) at \( t_5 \). If a failure occurs while \( R_1 \) is in progress during \( t_3-t_{11} \), the nested call \( L_1 \) may have modified the local state. To revert any changes, a local snapshot is taken at \( t_2 \), before \( R_1 \) executes. When \( R_3 \) returns at \( t_{10} \), a remote snapshot of all modified remote objects is taken and transmitted together with the result of \( R_1 \).

When another remote call \( R_2 \) fails due to a loss of network connectivity, the local node can recover at \( t_{14} \): (1) it undoes any local state updates using the most recent local snapshot taken at \( t_{12} \); (2) it locally creates the remote objects on-demand (see Section 4.4.4) based on the remote snapshot received at \( t_{11} \); and (3) it repeats the failed call \( R_2 \).

After a failure, CloudSplit periodically checks if connectivity to the remote node was restored. When this is the case, it takes a snapshot of all objects that were originally part of the remote partition and sends it to the remote node. The remote node then updates pointers between objects across partitions using proxy objects. Finally, remote code offloading is enabled again.

**Asynchronous strategy**

This strategy relaxes the requirement that each transfer of control must include a remote snapshot. Instead, a background thread periodically transfers updated remote snapshots at a configurable frequency. Since the latest remote snapshot may be inconsistent with the local state, CloudSplit takes a local snapshot together with each remote snapshot and rolls back the local state to that snapshot, before rebuilding the remote state and restarting local execution.

The asynchronous strategy is shown in Figure 4.6. The local node initiates three remote calls \( R_1 - R_3 \). The background thread transfers the latest remote snapshot to the local node every \( T \) seconds. As in the synchronous strategy, remote snapshots \( RS_1 \) and \( RS_2 \) are taken at \( t_6 \) and \( t_{14} \), respectively, before control of execution returns to the local node. This ensures that they are consistent with the corresponding local snapshots taken earlier (at \( t_3 \) and \( t_9 \)).

Before the remote call \( R_1 \) returns, a remote snapshot \( RS_1 \) is taken at \( t_6 \). Note that local execution continues at \( t_8 \), before the transfer of \( RS_1 \) finishes at \( t_{13} \). The transfer of \( RS_1 \) is concurrent with the next remote call \( R_2 \) at \( t_{12} \), which returns at \( t_{16} \). Consider the case in which the next remote call \( R_3 \) fails at \( t_{18} \). The corresponding remote snapshot \( RS_2 \) was taken at \( t_{14} \) but was not yet transferred to the local node before the failure. At this point, the local node must reconstruct the remote objects based on the last available remote snapshot, which is \( RS_1 \). The local node thus reverts to the consistent corresponding local snapshot, which in our example is the local snapshot \( LS_2 \) from \( t_9 \). Any updates to the local state after \( t_9 \) are not reflected in the remote snapshot \( RS_1 \) and have to be discarded. Based on the snapshots \( RS_1 \) and \( LS_2 \), the local node has access to the entire application state, effectively reverting application execution back to \( t_{10} \).

The advantages of asynchronous snapshotting are twofold: (1) remote snapshots are trans-
mitted asynchronously to the local node, in parallel with any computation that takes place on either the local or remote node. The transmission cost is thus amortised over time, reducing its impact on application response times; (2) potentially less snapshotting data must be sent because multiple updates to the same object may be combined and only transferred once. The larger the snapshotting frequency $T$, the greater these benefits become. However, a large value of $T$ also results in potentially more application state being lost after failure.

### 4.4.3 Incremental state snapshots

To reduce the size of snapshots, CLOUDSPLIT creates incremental snapshots that only include modified objects since the last snapshot. As a result, an individual remote snapshot is no longer self-contained but depends on all previous remote snapshots when reconstructing the latest version of the remote node state during recovery.

For incremental snapshots, the Compiler uses static analysis to identify when object state is updated. For primitive data and pointer fields, updates are caused by assignments. For collection data structures, such as arrays and dictionaries, updates also include the addition or removal of elements. The Compiler synthesises code that, when a new object is allocated or its state is modified, it is added to a global list of modified remote objects, $dirtyObjs$. Object deallocations are modified to remove entries from this list. When a remote snapshot is taken, only objects in $dirtyObjs$ are included in the snapshot. For easy reference, each remote snapshot contains a hash table that maps each object identifier to the remote snapshot with the most up-to-date version of that object.

In the synchronous strategy, a remote snapshot is transferred immediately after creation and the $dirtyObjs$ list is cleared. In the asynchronous strategy, creation and transmission of snapshots are decoupled, and multiple existing incremental snapshots may have to be merged before transmission to avoid sending old state. CLOUDSPLIT uses a global $snapshotList$ to
store remote snapshot entries per object that are pending transmission to the local node. The list is updated when object state changes and new snapshots are taken.

4.4.4 Virtual memory scheme

When reverting to local execution after failure, the application state may be larger than the memory capacity of the local node. iOS does not support virtual memory and instead forcibly kills applications in low memory conditions. To address this issue, CloudSplit realises a virtual memory scheme at the user level.

Rather than restoring all objects locally from the remote snapshots before resuming with local execution, the Compiler synthesises code that loads remote objects from a snapshot on demand. Such objects are added to a loadedObjs table, indexed by their object identifiers. The Compiler statically identifies points in the code, at which methods of a remote object are invoked. Before such an invocation, it adds code to check if, based on the object identifier, the object exists in loadedObjs. When the caller is a proxy object, the object identifier is stored in the id field; if the caller is another instantiated remote object, it is derived from the object identifier stored in the caller’s field, which used to point to the callee object in the remote node’s address space. If the remote object exists in loadedObjs, the call proceeds; otherwise, the remote object is loaded from the remote snapshot and added to loadedObjs.

The total size of objects is limited by the amount of available device memory. When memory is exhausted, an object from loadedObjs is evicted from main memory and potentially written back to flash memory. In our prototype implementation, objects are evicted arbitrarily subject to two rules: (1) objects that are on the call stack must not be evicted for correctness; and (2) priority is given to evict “clean” objects, i.e. objects that have not been modified, to avoid the overhead of writing them back to flash memory. Dirty objects are identified as part of the incremental snapshotting technique from Section 4.4.3. Although more efficient eviction policies, e.g. taking temporal and spatial locality of references into account, exist, we defer their exploration to future work.

4.4.5 Related work on checkpointing and program rollback.

Rollback-recovery techniques have been used for failure recovery and debugging. Toolkits such as DMTCP [ARC07] and CLIP [CPL97] allow transparent user-level checkpointing of distributed applications, relying on separate single process checkpoints, namely MTCP [RAC06] and libckpt [PBKL95]. Both copy entire memory regions and per-thread metadata to disk, which are later used to roll back execution by restoring memory contents. libckpt supports additional optimisations, such as incremental and copy-on-write checkpointing using page protection hardware to capture only the updated state since the last checkpoint.

While coordinated checkpointing of multiple communicating processes requires suspending execution, CloudSplit’s task is easier: with only two nodes, consistent snapshots are taken
when control of execution is transferred from one to another via remote calls. Merging local and remote snapshots, however, requires semantic knowledge, making techniques that copy entire heap regions or memory pages not applicable. Since access to page protection mechanisms is unavailable under iOS, CLOUDSPLIT uses static analysis to detect updates to object state.

Rx [QTSZ05] is a rollback-recovery technique used by the Flashback OS extension [SKAZ04] to recover from software bugs. Based on checkpoints, shadow processes are forked and immediately suspended to be used as replacements after failure. Since the fork system call is not available under iOS, this approach cannot be used.

Contrary to typical checkpoint-based techniques, log-based techniques [Bar81, BBG+89, SY85] model application execution as a sequence of intervals, each starting with a non-deterministic event (e.g. user input, receiving a message from another process or a state change based on output from a random number generator). These techniques log all non-deterministic events to stable storage and also maintain consistent checkpoints to replay fewer events during recovery. As part of recovery, they replay events in the original order from the most recent consistent checkpoint to recreate the application state.

The benefit of log-based techniques is that application state is reconstructed entirely after failure. This assumes, however, that all non-deterministic events can be identified and logged, which requires an understanding of the application logic, thus contradicting CLOUDSPLIT’s goal of automated application state partitioning. For example, in the Latrunculi board game (see Section 4.5.1), the game AI’s next move is chosen at random from a set of best moves. A log-based approach would have to log this random choice, requiring knowledge of the internals of the application. Log-based approaches also incur additional overheads due to the synchronous logging of events to stable storage and the replay of a potentially large number of events during recovery.

4.5 Evaluation

We evaluate CLOUDSPLIT experimentally to investigate: (1) how application response times benefit from code offloading with application state partitioning vs. state migration; (2) the effect of large memory consumption on application functionality and performance and (3) the overheads of the synchronous and asynchronous fault tolerance strategies, as well as the user-level virtual memory scheme.

4.5.1 Experimental set-up

We deploy CLOUDSPLIT on an Apple iPhone 4 as the local node and a 2.26 Ghz Intel Core 2 Duo machine with 8 GB of RAM as the remote node. We use an IEEE 802.11g WiFi network with an average round trip time (RTT) of 40 ms and a bandwidth of 6.3 Mbps, and a 3G
mobile network with an average RTT of 523 ms and a bandwidth of 0.4 Mbps. The default network used is WiFi unless stated otherwise.

We use two Objective-C applications that represent different extremes in how they handle computation and state. Due to the lack of publicly-available, non-trivial open-source iOS applications, both applications were ported from Mac OS, which only affected UI classes:

**Latrunculi board game.** Latrunculi [MB06] is an open-source board game, in which two players alternate moving pieces until all opponent’s pieces have been captured. An AI component uses the minimax algorithm to search a tree of consecutive future moves for the best next move. The search depth is configurable and determines the strength of the game AI. Latrunculi thus maintains a modest amount of state but performs an expensive AI computation affecting the entire state.

**iSpreadsheet application.** This is a spreadsheet application that operates on CSV files and supports a range of features, including: loading/saving spreadsheets; adding/removing cells, rows and columns; sorting by row/column; and calculating complex user-defined formulas over multiple cells with operations such as sum, average and median. We consider two typical workflows: a user loads a 2 MB spreadsheet (simple workflow); and a user loads a spreadsheet, sorts it by a row and then sorts it by a column (complex workflow). In general, iSpreadsheet manages arbitrary amounts of state—depending on the size of the spreadsheet—but only carries out localised computation on sets of cells.

We apply CLOUDSPLIT to both applications, generating partitionings based on the profiled workloads. For Latrunculi, the profiling workload constitutes of a complete game play for different AI search depths. CLOUDSPLIT creates a remote partition with the classes responsible for maintaining the board and the game AI. The workload for iSpreadsheet includes loading 2–8 MB spreadsheets, sorting them, calculating formulas and saving them. The obtained partitioning places the classes that perform the loading, sorting, calculation of formulas and saving on the remote node.

The results discussed in the remainder of this section correspond to averaged values over 10 experimental runs. We assert empirically that 10 experimental runs are enough to provide an accurate representation of the performance of the system with only a negligible variance between repeated runs. We therefore do not include error bars in graphs.

### 4.5.2 Application response time

First we compare the application response times for (i) an unpartitioned application; (ii) code offloading with state migration; and (iii) with state partitioning using CLOUDSPLIT for different fault tolerance strategies and workloads. For CLOUDSPLIT, we repeat each experiment with (a) no fault tolerance (noFT); (b) our fault tolerance mechanism with synchronous snapshots (sFT); and (c) with asynchronous snapshots (aFT).

Due to the unavailability of other offloading systems for iOS, we cannot compare the absolute performance of CLOUDSPLIT directly to other systems with the same application workload.
We measure application response time as the time between when a user action is initiated and when it results in a UI update. We provide a break down of application response time in terms of the time spent on (a) remote code execution, (b) local code execution, (c) state snapshot generation, (d) RPC invocations and (e) synchronous or asynchronous transfers of remote snapshots.

Game application

Figure 4.7 shows the average time to respond to a single move with an AI depth of three. For the WiFi network in Figure 4.7(a), offloading with state migration results in a speedup
of 6.2×. CLOUDSPLIT without fault tolerance manages to improve response time by 7.9×. When using the synchronous fault tolerance strategy (sFT), this reduces to 7.2×. The speedup under asynchronous snapshotting (aFT) is 7.8×, which is comparable to sFT. The bulk of the time is spent remotely executing calls, while the time to generate snapshots is negligible.

Here, offloading provides a significant benefit due to the computational cost of the game AI. State migration and partitioning perform similarly due to the small amount of state.

Note that we use an infinite snapshot frequency $T$ for aFT, which results in no snapshot transfers during the experiment, in order to give an upper bound on the best possible performance attainable by this strategy. For the game application, aFT does not provide a benefit over sFT because the amount of modified remote state after each board move is relatively small. (The state allocated by the game AI for the tree search—which is large—is only transient and therefore not included in snapshots.)

The results for the 3G network are given in Figure 4.7(b). Here offloading with state migration suffers from the lower network bandwidth and only manages to speedup execution by 1.5×. CLOUDSPLIT without fault tolerance (noFT) manages to achieve a speedup of 2.3×.
Performance is also lower than with WiFi due to the higher latencies of RPC calls. With fault tolerance, the application is $1.9 \times$ (sFT) and $2.2 \times$ (aFT) faster, respectively. As for WiFi, the difference between the synchronous and best possible asynchronous strategy is small. In summary, state migration becomes less beneficial with a low bandwidth network.

### Spreadsheet application

Here we consider two frequencies ($T=1$ s and $T=30$ s) representing frequent and sporadic snapshots, respectively. The black bars in Figures 4.8 and 4.9 show the duration of asynchronous snapshot transfers relative to the application response times.

**Simple workflow:** For the WiFi network, Figure 4.8(a) shows the breakdown of the time required to load the spreadsheet. Offloading with state migration does not improve performance due to the amount of spreadsheet state that must be transferred with each remote call. In contrast, without fault tolerance, CloudSplit achieves a speedup of $4.6 \times$. The speedup for sFT reduces to $1.3 \times$ because remote snapshots include the entire spreadsheet data.

For aFT with frequent snapshots ($T=1$ s), we observe a higher speedup of $2.2 \times$. The asynchronous transmission enables the remote node to continue serving remote calls in parallel. For example, multiple calls related to UI updates that occur after loading a spreadsheet can be handled by the remote node before the transmission of the large snapshot has completed. With a larger snapshot interval ($T=30$ s), the speedup ($3.5 \times$) approaches noFT. Remote execution occupies less time because it is no longer concurrent with snapshot transmission, thus improving the performance of the remote node.

The corresponding results for the 3G network are shown in Figure 4.8(b). Offloading with state migration again fails to provide any benefit. Due to higher latency and lower bandwidth, the speedups of CloudSplit are more modest: without fault tolerance, the application is only $3.8 \times$ faster. With sFT, the unpartitioned version now has better performance due to the large snapshots. CloudSplit with aFT achieves speedups of $1.5 \times$ and $3.1 \times$ for $T=1$ s and $T=30$ s, respectively, because asynchronous snapshot transmission also affects application execution.

**Complex workflow:** The results for the complex workflow over WiFi are shown in Figure 4.9. Response times are $2.6 \times$ faster with state migration and $11.8 \times$ faster with state partitioning (noFT). Due to the partitioned state, many remote calls can be executed entirely using remote state. For sFT, the speedup is only $4.3 \times$ because of the large remote snapshots—each spreadsheet operation may affect a portion of the data, which in turn must be transmitted to the local node.

As before, aFT improves performance: $7 \times$ for $T=1$ s and $9.7 \times$ for $T=30$ s. For aFT with $T=1$ s, three distinct remote snapshots are taken, i.e. one for each of the load, sort-by-row, and sort-by-column operations, leading to three asynchronous transfers. With $T=30$ s, only one remote snapshot is transferred, after all three operations have completed. Additionally, overlapping updates to remote objects are combined to obtain the latest version of the state.
4.5.3 Memory consumption

We observe the memory consumption of the local and remote partitions and investigate the performance of CLOUDSPLIT as the application state increases.

Figure 4.10 shows the memory consumption of the unpartitioned version and the remote partition under CLOUDSPLIT over time for both applications. The workflow for the game application (Figure 4.10a) consists of three consecutive moves played. In the unpartitioned version, the memory consumed by the AI component to construct the tree of future moves is approximately 4 MB. Using CLOUDSPLIT, the remote partition consumes 6 MB of memory—the increase is due to the additional meta-data needed by the snapshotting mechanism (see Section 4.4.1). The local partition only consumes a negligible amount of memory because the AI state is maintained remotely.

A similar behaviour can be seen in Figure 4.10b, which plots the memory consumption when loading and sorting a 2 MB spreadsheet twice. Local memory consumption of the partitioned version is again low because the spreadsheet data is maintained remotely.

To investigate the change in performance when the application state size increases, Figure 4.11 shows the response time of remote calls for both applications as a function of the state size.
We vary the memory consumption of the game application (Figure 4.11a) by changing the AI search depth. The response time increases exponentially due to the complexity of the game AI. For a search depth above three, the unpartitioned application is terminated due to insufficient memory; in contrast, using CloudSplit, the game continues to run, exploiting the larger main memory of the remote node.

For the spreadsheet application in Figure 4.11b, we adjust the size of the spreadsheet. The unpartitioned version cannot handle an 8 MB spreadsheet due to a lack of memory—the partitioned version continues to work.

4.5.4 Post-failure performance

Next we explore the cost of our virtual memory scheme, which loads objects on-demand after failure recovery (see Section 4.4.4). We compare the response times of the spreadsheet application when sorting spreadsheets of different sizes with (i) an unpartitioned application (local execution); (ii) remote execution using CloudSplit; and (iii) post-failure execution using CloudSplit, i.e. local execution with on-demand loading of remote objects from snapshots stored in flash memory. The values for post-failure execution include recovery times, which are in the order of milliseconds. We also indicate the time required to load objects from flash memory.
Figure 4.12 shows that, for all state sizes, CloudSplit achieves a speedup of approximately 14×, compared to unpartitioned execution. For state sizes above 8 MB, the unpartitioned execution exhausts memory, while CloudSplit manages to resume local execution after a failure using the virtual memory scheme. After failure, local execution exhibits approximately a 25% degradation in performance—15% is caused by the loading of objects from flash memory, while the rest is due to the additional checks if objects are present in memory.

4.6 Discussion

CloudSplit’s snapshot-based fault tolerance mechanisms assume that only a single partition executes at any point in time. This assumption ensures that local and remote snapshots are consistent with each other because remote calls synchronise the local and remote state. This is not the case, however, for multi-threaded applications, in which different threads execute on both the local and remote nodes in parallel. This means that state updates caused by local threads and incoming remote calls cannot be distinguished, necessitating a more complex technique such as log-based recovery [Bar81]. We leave support for snapshotting with multiple threads for future work.

Our mechanisms also assume that offloaded application method calls have no externally visible side-effects, such as network communication or file I/O. Since CloudSplit does not record which operations of an offloaded call executed successfully before failure, all operations are simply repeated on the local node during recovery, which may lead to unexpected behaviour for operations with side-effects. Such operations, however, typically use dedicated iOS APIs, which CloudSplit identifies as platform native as part of the profiling step and thus only places on the local partition.

4.7 Summary

In this chapter, we presented an offloading approach that automatically partitions the state of Objective-C applications to allow them to utilise the larger memory of remote nodes while reducing application response times. The benefit of state partitioning over existing offloading approaches is twofold: (1) it permits workloads that exhaust local memory to be executed; and (2) it avoids wasteful state migration and the associated runtime overheads with every offloaded remote call.

However, state partitioning complicates failure handling since remote state becomes inaccessible after failure. We overcome this problem through a new snapshot-based fault tolerance mechanism. State changes are periodically backed up to the local device, which can thus recover remote state after a failure. Since the remote state may grow larger than the available local memory, objects are loaded from snapshots on-demand based on a user-level virtual memory scheme.
We realised this approach with a system called CloudSplit, which requires no modifications to the iOS platform or Objective-C. The evaluation of our prototype implementation revealed that state partitioning allows memory-intensive application workloads, which normally would have exhausted the local node’s main memory, to execute. In terms of response times, our approach is most beneficial for applications with repeated intensive computation on large amounts of state, where offloading with state migration incurs a larger communication cost. CloudSplit outperforms conventional state migration approaches by offering reductions of up to a factor of $15\times$ in application response times. The overhead of our fault tolerance mechanism remains low, especially with a low snapshotting frequency. This, however, comes at the price of higher data loss after failure.
Chapter 5

Network-Focused Offloading

This chapter describes a new static offloading approach that reduces mobile network traffic caused by the interaction of mobile client applications with Internet backend services. This is accomplished by intercepting application data transfers at the network edge to eliminate unnecessary data sent to the mobile device before it enters the radio access network (RAN). The approach benefits both mobile network operators and consumers by reducing traffic contention in RANs and data charges.

5.1 Overview

Mobile network operators are projected to carry the bulk of “last mile” Internet traffic in the future. According to Cisco, global mobile data traffic will grow 13-fold from 2012–2017 [CIS13]. Yet, in current mobile networks, operators struggle to keep up with the ever-increasing volume of data traffic. In particular, radio access networks (RANs) become a bottleneck due to the limits on the density of mobile base stations in urban environments and on the frequency spectrum that they can utilise [RA13]. Even the next generation of 4G/LTE networks are unlikely to meet the exponentially growing demand for mobile data capacity [CNR10].

While about half of mobile data traffic constitutes video streaming, the other half is non-multimedia traffic, with a substantial fraction caused by the large number of client/server applications on today’s smartphones [CIS13]. Mobile client applications interact with Internet backend services, which host the application’s content, through service APIs. For example, mobile clients for social networks, such as Facebook and Twitter, retrieve updates on user activity; photo sharing clients, such as Picasa and Flickr, host users’ photo collections remotely; and e-commerce clients, such as eBay, Groupon and Amazon, provide the means for online purchasing of different goods. Clients typically access content through restful HTTP APIs, such as the Twitter REST API [Twi13] and the Amazon Marketplace Web Service API [Ama13].

We make the observation, supported by empirical evidence in Section 5.3, that mobile client
applications retrieve more data from backend service APIs than is strictly necessary, thus increasing the utilisation of RANs. There are two main causes for this:

(1) Backend service APIs are designed with generality in mind, and not tailored towards the needs of specific mobile client applications. As a result, not all data returned by a backend API call is used by the client, with some of it being discarded after transmission. For example, the Twitter API response to a request for the list of recent messages includes detailed user account information, which is typically ignored by clients.

(2) In addition, client applications often aggressively prefetch binary content such as images from backend services. While this improves the application response time when the user accesses prefetched content, it increases the amount of data transmitted over the RAN. Client applications often employ simple prefetching strategies such as prefetching all objects [HFG+12], which are wasteful. For example, Twitter clients typically prefetch all user profile images associated with a list of messages, even if only a few images will be viewed by the user.

Various techniques for reducing mobile data traffic were proposed in the past. Compression (e.g. gzip for HTTP traffic) is widely used to reduce the overhead of verbose application layer protocols such as XML used by backend services; new application layer protocols such as SPDY [Pro13] and QUIC [Ros13] are designed to decrease data transmission times by eliminating unnecessary communication. Client- or network-side caches [EGH+11] and redundancy elimination (RE) proxies [SW00] avoid the repeated transmission of the same data. All of these approaches, however, cannot prevent the transmission of unused application data across the RAN.

In contrast, our network-focused approach partitions mobile client applications between the mobile device and a remote node located at the network edge, i.e. as part of the mobile network, with the goal of reducing data traffic from backend services to client applications. The remote counterparts of client applications are application-specific proxies (ASPs) that host the logic from client applications that parses response data from the backend service and stores the results as application data objects, which are in turn transmitted to the client application. Since data parsing is performed by the ASP, any data that is retrieved but not further used by the client application after parsing will not be transmitted to the mobile device.

The contributions of this approach are summarised below:

1. Filtering of backend API data: A stateless ASP is generated automatically from the source code of a client application through static code analysis. We assume that client applications are designed according to a model-view-controller (MVC) design pattern [App12], which separates data presentation from data representation in an object-oriented design. Our approach identifies classes from the application model that retrieve and parse data from a backend service. These classes are placed at the ASP, and are accessed by the client application using remote calls over the RAN. If data
returned by a backend service is never used as part of the model, it will therefore not be sent to the client application.

To reduce the number of remote calls between the local and remote nodes, we employ two optimisation techniques:

**Coalescing of remote calls.** An ASP may create many data objects during the parsing of backend responses, which would lead to many remote calls across the RAN. To improve application performance, multiple transmissions of object fields from the ASP to the client are coalesced into a single remote call. Opportunities for applying this optimisation are recognised during static analysis.

**Creation of transient data objects.** If data objects are updated by the ASP as part of the parsing process multiple times, it becomes more efficient to materialise them at the ASP and only transmit the final versions to the client. We identify opportunities to create such transient data objects at the ASP when it reduces the number of remote calls to the client application.

2. *Replacement of prefetched objects with futures:* Client applications prefetch objects such as images from the backend service to reduce application response times. Since prefetched objects are used by UI objects, they are transferred unnecessarily from the ASP to the client before being displayed. The ASP therefore replaces references to large binary objects with *futures*, which are sent to the client instead. When a client application accesses a future, it retrieves the actual binary object from the ASP.

As in the case of our compute- and memory-focused offloading approaches (see Chapters 3 and 4), this approach is designed to support applications written in unmanaged programming languages. As it builds on top of CloudSplit, we target Objective-C applications on the iOS platform. We have realised our network-focused approach with a system called EdgeReduce, which generates ASPs to reduce RAN traffic for mobile client applications without any modifications to Objective-C or the iOS platform. ASPs are deployed on the network path between the mobile device and the backend service.

We evaluate EdgeReduce on three real-world client applications for Twitter, Groupon and Yahoo! Finance on the iOS platform. We show that EdgeReduce can reduce RAN traffic by a factor of up to $8\times$, while only increasing application response time by at most 10%. We also show experimentally that EdgeReduce has the potential to speed up execution when large amounts of network data are sent to client applications.

The remainder of this chapter is organised as follows: Section 5.2 discusses existing approaches for mitigating mobile network contention. Section 5.3 discusses the potential for data traffic reduction based on unused data in mobile client applications. Section 5.4 introduces the design of EdgeReduce, and Section 5.5 explains how a partitioning is implemented using the source code of client applications written in Objective-C. In Section 5.6, we present our evaluation results to show the effectiveness and overhead of EdgeReduce. We conclude
Network-Focused Offloading

this chapter with a discussion on how EdgeReduce can be combined with CloudSplit in Section 5.7 and a summary in Section 5.8.

### 5.2 Design space

EdgeReduce is designed to extend CloudSplit with network-related objectives. Therefore, the comparison of CloudSplit with existing offloading systems (provided in Section 4.2) also applies to EdgeReduce and is summarised in Table 5.1. We note that the only difference from CloudSplit is that EdgeReduce also reduces mobile network traffic.

Although, to the best of our knowledge, no other offloading approach has focused on reducing mobile network traffic, operators have deployed or investigated a variety of solutions to address contention in mobile networks. These solutions are discussed in the remainder of this section.

**Caching.** Many solutions for caching popular content to reduce network traffic were proposed in the past (see Section 2.1.3). Forward caches use dedicated middleboxes for intercepting HTTP requests in backhaul networks [EMS94, EGH+09]. The content request is relayed to the backend service only if it cannot be satisfied from the cache. Caching thus suppresses redundant transfers of data from backend services to reduce network usage.

Client-side caches exist on mobile devices as part of client applications [SI02]. They maintain copies of previously retrieved content from a backend service in case it is requested again or required during offline operation. Maintaining large client-side caches, however, is infeasible due to the limited memory and storage resources of mobile devices.

Caches inherently require repeated requests for the same content to reduce traffic. In particular, they cannot reduce traffic from backend services that is unique and application-specific. Finally, depending on their deployment locations, they mainly focus on reducing contention in backhaul networks, ignoring bottlenecks in the RANs.

**Compression.** Compression is a widely-used approach for reducing the data transmitted from backend services to client applications [RJ91]. HTTP provides inherent support for the gzip and deflate compression methods. A client application announces its supported compression methods when issuing HTTP requests, and a backend service responds with compressed data in a supported format. For image transmission in mobile networks, transparent lossy

<table>
<thead>
<tr>
<th>System</th>
<th>Static vs. dynamic (runtime adaptation)</th>
<th>Platform and programming language</th>
<th>Code offloading goal</th>
<th>Partitions application state</th>
<th>Partitioning granularity</th>
<th>Fault tolerance (network failures)</th>
<th>Accede runtime overhead associated with</th>
<th>Resource profiling</th>
<th>Offload decision</th>
<th>State migration</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAVI</td>
<td>dynamic</td>
<td>Win. Mobile (C++)</td>
<td>energy</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ThinkAir</td>
<td>dynamic</td>
<td>Android (Java)</td>
<td>energy/time</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>COMET</td>
<td>dynamic</td>
<td>Android (Java)</td>
<td>time</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CloneCloud</td>
<td>static</td>
<td>Android (Java)</td>
<td>energy/time</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ASYWARE</td>
<td>static</td>
<td>cross-platform (C)</td>
<td>time</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CloudSplit</td>
<td>static</td>
<td>iOS (Objective-C)</td>
<td>time/memoney/network</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>EdgeReduce</td>
<td>static</td>
<td>iOS (Objective-C)</td>
<td>time/memoney/network</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 5.1: EdgeReduce vs. representative code offloading systems.
compression is sometimes used [FC96]. While it degrades image quality, it can substantially reduce images sizes.

Compression cannot eliminate unused, application-specific data traffic sent by backend services. It is, however, an orthogonal approach in that it removes redundancy from any application data transmitted across the network.

**Redundancy elimination (RE).** RE schemes are protocol-independent techniques for eliminating redundant network traffic [SW00, AGA+08, AMAR09]. For example, they may identify identical web content named by different URLs or delivered using different protocols by processing the payload of packets at the cost of additional computational overhead. Synchronised caches are then placed at both ends of a bandwidth-constrained channel, and small fingerprints of data in the cache are exchanged.

While RE can reduce redundant data traffic considerably, it suffers from the same problems as caching: it requires substantial resources at the client side, and it cannot reduce data traffic without any redundancy.

**Efficient protocols.** Another approach to reduce network contention is to deploy more efficient protocols (see Section 2.1.3). The SPDY protocol [Pro13] is added between the application and transport layers to allow concurrent interleaved streams over a single TCP connection. Though designed specifically for minimising transmission times, SPDY also compresses and in some cases eliminates request and response HTTP headers. However, given that headers today range in size from approximately 200 bytes to 2 KB, its ability to reduce overall data traffic is limited [Pro13].

QUIC [Ros13] is an experimental transport-level protocol that uses the connectionless nature of UDP to provide a multiplexed, congestion-aware transport with low latency. It is designed to avoid head-of-line blocking in switches. However, it requires client- and server-side support and lacks the maturity of established transport-layer protocols such as TCP.

In general, new network protocols are typically application-agnostic, which means that they cannot exploit opportunities for data reduction of backend service traffic that are application-specific.

### 5.3 Mobile client applications

In this section, we identify opportunities for reducing RAN traffic pertaining to the interaction of mobile client applications with backend services. We first explore the nature of the data transfers between client applications and backend service APIs (Section 5.3.1). The section finishes with a description of the baseline architecture of client applications and how it can be exploited to reduce unnecessary data transfers across RANs (Section 5.3.2).
5.3.1 Mobile client communication

Modern mobile client applications communicate with backend services through web APIs. These APIs are implemented using the HTTP protocol and follow the representational state transfer (REST) architectural style [Fie00], which provides access to resources referenced by global URL identifiers.

Backend APIs expose a programmatic interface using well-defined request-response interaction. Typically backend APIs support (i) the retrieval of lists of data items, such as products in eBay, messages and friend lists in Twitter, photos in Flickr and financial data in Yahoo! Finance; (ii) the search for specific data items given user-defined criteria, such as products by category, friends by name or images based on metadata; and (iii) the updating of content such as adding new products or social status updates. API calls provide general operations on the data maintained by the backend service without taking specifics into account of how a given client application presents the data to the users.

Applications that interact with such backend service APIs need to understand the format of the data returned. A common practice is to use human-readable text-based encoding formats, which facilitates interoperability and cross-platform support. The most commonly used formats are the extensible markup language (XML) and the JavaScript object notation (JSON). Their benefits in terms of simplicity and interoperability, however, come at the cost of substantially higher encoding overheads: e.g. XML includes metadata about the data schema as part of each data message. This leads to an increase in the transmitted data compared to application-specific binary encoding formats.

In Figure 5.1, we show part of the data returned by the Twitter REST API in response to a user request for the most recent Tweet messages. On the left-hand side, we show the representation of a single Tweet message in XML, as returned by the backend service. The right-hand side shows the corresponding fields of three data objects, NTLNMessage, NTLNUser and NTLNIconContainer, that a Twitter client application uses to model this information. Arrows indicate the pieces of network data that are actually used to initialise object fields. Items that are crossed out correspond to the encoding overhead of XML.

The NTLNMessage object stores information about a Twitter message, such as its timestamp, its content and the posting user. It also contains pointer references to the NTLNUser object, which stores information about the user responsible for the message, and the NTLNIconContainer object with a user’s profile picture.

As shown in the example, less than half of the data returned by the backend service is actually used by the client application: the original XML message has 2429 bytes, while the created data objects only occupy 842 bytes in memory. In general, there are three main sources of inefficiencies, I1–I3, when a client application interacts with a backend service:

I1. Inefficient data representation. Data items returned by the backend service API are expressed using an inefficient encoding format. In the example in Figure 5.1, a large part of the overhead is due to the repetitive nature of start and end tags, which capture the XML
allows client applications to query for user data using an SQL-style interface. This permits, for example, Facebook Query Language (FQL) [Fac13], which allows client applications to query for user data using an SQL-style interface. This permits them to retrieve precisely the required data by specifying filters on the returned data items.

To address this problem, it is usually necessary to change the backend service API. For example, Facebook offers a new API, the Facebook Query Language (FQL) [Fac13], which allows client applications to query for user data using an SQL-style interface. This permits them to retrieve precisely the required data by specifying filters on the returned data items.

This inherent redundancy in the returned data results in high compression ratios. Therefore many backend service APIs that use HTTP with XML or JSON use transparent gzip compression, as supported by web servers and clients.

I2. Unnecessarily returned data. A backend service API may return more data items or fields than necessary when compared to what is used by the client application. In the Twitter example above, the coarse granularity of the Twitter REST API does not support a fine specification of the data of interest: the Twitter client cannot express that it does not require statistics about the past user behaviour, the user’s display settings or other account information, such as its creation date, timezone or language preferences. All of this metadata is included in the response data regardless.

Figure 5.1: Data returned by the Twitter backend service API and its subsequent use in a Twitter client application.
However, such more expressive backend service APIs require the reengineering of existing client applications and add more complexity to the development of clients.

**I3. Unnecessarily prefetched data.** A client application may prefetch many data items, which will not be all used by the application. For example, the Twitter client application prefetches all images associated with retrieved Tweet messages. In Figure 5.1, when a user requests the most recent Tweet messages, the client application receives the data items that contain the URLs of images associated with a given message, which are stored in the `iconImage` field of the `NTLNIconContainer` object. When processing this data, the client retrieves all images by default. However, each image is stored locally and only displayed when a user views the message associated with the image. Depending on user behaviour, only a fraction of the images returned by the backend service are ultimately displayed on the mobile device.

Tuning the prefetching behaviour of a client application requires changes to its business logic. In general, prefetching requires the choice of a policy that strikes a balance between the amount of prefetched data and the probability that a requested object was prefetched. A wide range of prior work exists on effective prefetching strategies, e.g. in the context of web applications exploiting spatial locality, pattern mining and contextual information [HFG+12, SKDW05]. For simplicity, mobile client applications typically do not use sophisticated prefetching policies but prefetch all objects instead.

### 5.3.2 Mobile client application architecture

Mobile applications on the iOS and Android platforms are typically structured around a variation of the *model-view-controller (MVC)* design pattern [App12]. An MVC design separates the representation of information from the user’s interaction with it in an object-oriented application. As a result, applications are more easily extensible as objects become reusable and their interfaces clearly defined.

As shown in Figure 5.2, the MVC design pattern has three types of objects, *Model*, *View* and *Controller* objects, which are separated by interfaces over which they communicate with each other: View objects represent the user interface (UI) of applications; Model objects encapsulate application data and corresponding operations on that data (i.e. the business logic of the application); and Controller objects mediate input between the two by converting user actions into commands, thus keeping them separate.
Client application architecture. Based on an examination of typical mobile client applications on the iOS platform (see Section 5.6), we classify Model objects according to the role that they play in the applications, as shown in Figure 5.3. They can be divided into three different categories:

*Communicator* objects directly interface with a backend service API; *Data Parser* objects process the network data returned by backend API calls and convert them to separate data fields with semantic meaning to the application; these are stored as part of *Data Objects*, which encapsulate the data used by the application and their associated operations.

**Application workflow.** Based on the above classification, it is easy to describe the processing workflow when data is retrieved from a backend service by a client. Figure 5.3 shows the sequence of steps leading to the display of information requested by a user in a mobile client application.

When a user decides to view, for example, their latest Tweet messages, they directly interact with the appropriate View object. This request propagates to the corresponding Controller object (step 1), which in turn relays the request to a Communicator object (step 2). The latter constructs an HTTP request according to the user’s input. The request is sent to the Twitter backend service (step 3), which replies with the network data that encapsulates the information requested (step 4). This is passed to a Data Parser object to deserialise it (step 5). The deserialised data is used to initialise the corresponding Data Objects, which model the information according to the application semantics for future processing (step 6). Pointers to the Data Objects related to the user request are returned to the Controller object (steps 7–9) and are used to update the corresponding View objects (steps 10–12), thus potentially displaying the information on the mobile device.

**Opportunities and challenges.** Given the above architecture of client applications and their workflow for retrieving, processing and displaying data from a backend service, we identify opportunities for reducing RAN traffic. Our goal is to eliminate unnecessary data returned by a backend service before it traverses the RAN in order to reach the device.
A simple observation is that the Communicator and Data Parser objects only create a Data Object for content that is used subsequently by the Controller objects. In other words, unnecessarily transmitted data from the backend service will not be output by Data Parser objects. Therefore less data should be output by the Data Parser object compared to what was retrieved by the Communicator object, addressing inefficiencies I1 and I2 from Section 5.3.1.

The above observation, however, does not address the inefficiency I3 due to unnecessarily prefetched data. For prefetched data, such as images, Data Objects are created by the Data Parser and subsequently accessed by the Controller. The distinguishing feature to identify unused prefetched Data Objects is that they are never used by View objects.

### 5.4 EdgeReduce design

The goal of EdgeReduce is to address the inefficiencies I1–I3, as identified in Section 5.3.1, in relation to the response data returned by backend service API calls. EdgeReduce achieves this by extracting application-specific proxies (ASPs) from client applications to filter the data returned by backend services. ASPs can be deployed at the network edge—either directly on mobile base stations [IBM13] or within the mobile backhaul network, e.g. at radio network controllers or gateway equipment (see Section 2.1.3).

The operation of EdgeReduce involves two steps: (i) a static analyser is used to identify the application logic that needs to be included in the ASP given the source code of the client application; and (ii) a source-level compiler generates the ASP implementation and transforms the client application so that all communication with the backend service occurs via the ASP.

In Figure 5.4, we give an overview of a client application after it was transformed by the EdgeReduce approach. The generated ASP contains the Communicator and Data Parser classes of the original client application, which are responsible for retrieving and processing the data from the backend service, data_API. All other application classes remain on the
client side and are only exposed to data_app, which is the portion of data_API that is used by the client application to construct its application data models. Any additional unused data included in data_API is thus not delivered to the client application, relieving the RAN from unnecessary data transfers.

5.4.1 Filtering of backend API data

As discussed in Section 5.3.2, when a client application makes a backend API call, the Communicator object is responsible for interfacing with the backend service (step 1 in Figure 5.4b). It makes the API call and receives the data_API as the response (step 2). In step 3, data_API is passed to a Data Parser object, which deserialises the data to obtain the actual data fields, data_app, that are used by the client application. After that, data_app is used to initialise the client’s Data Objects in step 4.

Since that Communicator and Data Parser objects are now part of the ASP, the client application never receives unnecessary data returned by API calls. Only data_app is visible outside of the scope of the ASP. As in the original version of the client application, the unused data returned by the backend API call is discarded by the Data Parser objects. Since this now occurs before the data reaches the mobile device, it avoids unnecessary data transmissions over the RAN.

5.4.2 Replacing data objects with futures

A client application may prefetch large binary Data Objects such as images (see Section 5.3.1). The approach described in the previous section would transfer all such Data Objects to the client, even if they are subsequently unused by the View classes of the client.

EDGEREDUCE addresses this problem as follows: the ASP can replace large binary Data Objects with futures, which are significantly smaller in size. It therefore avoids the transmission
of these objects over the RAN until they are actually used by the client application. In this case, the client application retrieves them from the ASP, using the corresponding futures as a reference.

The optimisation for replacing Data Objects with futures is shown in Figure 5.5. When an API call returns a binary Data Object \textit{binary\_object} (step 1), the response data is passed to the Data Parser object (step 2). Before the Data Parser object instantiates the corresponding Data Object, it first creates a new association between the \textit{binary\_object} and a fresh \textit{future} (step 3). The \textit{binary\_object} is stored at the ASP and its future is given to the corresponding Data Object (step 4). When the client attempts to access the \textit{binary\_object} (step 5), it triggers a request to the ASP using the future (step 6). The ASP then returns the original object to the client (step 7). As a result, a prefetched \textit{binary\_object} that is never accessed by the client will remain at the ASP and not transferred over the RAN.

5.5 \textbf{EDGE\textsc{REDUCE} implementation}

Next we describe the process of partitioning a mobile client application given its source code in order to generate the ASP. This includes: (i) identifying the Communicator and Data Parser classes through source-level program analysis (Section 5.5.1); and (ii) transforming the client’s source code to place the Communicator and Data Parser objects as part of the ASP, which involves converting the corresponding local method calls between Controller and Communicator objects, as well as Data Parser and Data Objects, to remote calls (Section 5.5.2).

5.5.1 \textbf{Source-level program analysis}

\textsc{Edge\textsc{REDUCE}} statically analyses the source code of client applications to distinguish between the different types of application classes according to the classification discussed in Section 5.3.2. The goal is to identify Communicator and Data Parser classes to be placed at the ASP.

\textbf{Communicator} classes are selected based on the fact that they include methods that interact with objects of the \texttt{NSURLConnection} class. This class is defined in the iOS Foundation framework, which is a base layer for all primitive Objective-C classes. \texttt{NSURLConnection} objects retrieve data from a URL in a synchronous or an asynchronous fashion. Objects of this type are used to interface with the API of backend services.

\textbf{Data Parser} classes are defined as the classes that perform serialisation and deserialisation of data transmitted over the network. Usually this kind of functionality is realised by built-in iOS classes such as the \texttt{NSJSONSerialization} class or third-party libraries such as the \texttt{SBJSON} library. For \textsc{Edge\textsc{REDUCE}}, we manually compiled a list of such classes from libraries for the two most commonly used serialisation formats, namely the XML and JSON. This list is given as input to \textsc{Edge\textsc{REDUCE}’s} static analyser. For example, for XML, Data Parser
classes include the NSXMLParser class and the libxml2 C library; for JSON, they include the NSJSONSerialization class and the JSONKit, TouchJSON and SBJSON libraries. EdgeReduce can be extended to support new encoding formats, as long as it is possible to identify classes that handle data according to the given format.

As in the case of AnyWare and CloudSplit, the tools that we use to analyse statically the source code of a client application comprise simple string matching algorithms that search the source files of different application classes for calls to specific object class methods, as specified above. The focus of this work is not to realise a fully-featured static analysis tool, but rather showcase how Communicator and Data Parser objects can be inferred automatically using string matching techniques.

5.5.2 Source code transformation

EdgeReduce's source-level compiler partitions client applications written in Objective-C to generate ASPs. It uses the methodology described for CloudSplit in Section 4.3.3 to split the application source code into a local and a remote code partition. Objects interact transparently across partitions using proxy objects, which relay method calls to the actual object instances, residing on the other partition, using RPCs.

This section first describes the additional optimisations employed by EdgeReduce to reduce the RPC overhead incurred by a given partitioning. These consist of: (i) the coalescing of multiple identical remote calls into a single call (Section 5.5.2); and (ii) the generation of transient Data Objects at the ASP, which are then returned to the client in a single remote call, thus avoiding multiple calls for individual fields (Section 5.5.2). We then describe the technique used for the replacement of large binary Data Objects by futures, thus reducing the amount of prefetched content exchanged between the backend service and the mobile device over the RAN.

Coalesced remote calls

To minimise the number of remote calls, EdgeReduce modifies the source code to coalesce multiple remote method calls into a single call.

An obvious choice where this optimisation can be applied is for method calls that retrieve individual objects fields. In Objective-C, object fields have a protected scope by default. This makes them visible in methods of the class that defines them but hidden from all other code. To get and set object fields directly, accessor methods need to be provided in the object class, i.e. getter and setter methods, which read and write the corresponding instance variables. These methods are either provided by the developer manually or generated automatically by the compiler through the @property and @synthesize directives.

A typical implementation pattern is that, within a given execution block, the same object fields are retrieved multiple times in succession using a getter interface without being modified
using a setter interface. An example is shown in Figure 5.6a where an integer field is read twice for an addition (line 8) and a subtraction (line 10). If executed naively by EdgeReduce, this would result in overhead due to repeated remote calls in cases in which the code that retrieves the field and class A are placed on the ASP and the client, respectively.

EdgeReduce’s static analyser identifies cases in which it is possible to coalesce multiple identical getter calls into a single remote call, as shown in Figure 5.6b. Once identified, in line 13, the field is first stored in a temporary variable temp, which is used to replace all subsequent calls to the corresponding getter method (lines 14 and 16). This optimisation, however, can only be applied when the following two constraints hold:

**Constraint 1:** It must be possible to predict the returned value of the getter method within the scope of the execution block. This means that the setter method for the corresponding object field must not be invoked in between consecutive identical getter calls (i.e. lines 8 and 10).

Conventional getter methods in Objective-C are named after the corresponding instance variable; the naming convention for setter methods capitalises the instance variable name and prefixes it with “set”. To identify opportunities for applying this optimisation, EdgeReduce identifies locations in the source code at which reads and writes to a field occur and reasons how these are interleaved.

**Constraint 2:** The getter method is side-effect free, i.e. it is only used for the purpose of returning a given object field.

In general, implementations of getter methods are simple and rarely have side effects. However, there are two exceptions: (i) getter methods that return non-primitive object fields (i.e. pointers to another application object) may invoke the retain method on the returned object, which allows the callee to acquire ownership of the object; (ii) getter methods may make use of defensive copying techniques, e.g. when the data returned is mutable but should
not be modified by other code. Such techniques involve returning a reference to a copy of the requested object, rather than the mutable internal value.

Both cases can be identified automatically: for compiler-generated methods, the above side-effects are specified by the developer using Objective-C attributes, which alter a getter method’s default behaviour, and thus can be inferred directly; for manual implementations, getter methods with the above side-effects can be identified due to their invocations of the `retain` or `copy` methods on the returned object.

**Transient Data Objects**

As illustrated in Figure 5.7a, Data Parser objects may interact frequently with Data Objects, which would result in a high number of remote calls between the ASP and the client. A large number of these method calls consists of calls to the Data Objects’ setter methods, which initialise each of their fields separately rather than using a single call.

Figure 5.7b shows how `EdgeReduce` overcomes this problem by permitting the ASP to execute frequent calls to Data Object methods locally. When allocating and initialising these objects, the ASP uses *transient Data Objects*, which are Data Objects that are created at the ASP—instead of the client. Transient Data Objects remain valid until the ASP returns them to the client application via a remote call made by the Controller object. This requires serialising transient Data Objects and returning their actual data to the client, instead of using an object identifier. The client then initialises its permanent Data Objects using the transient Data Object for future use by the client application.

**Data Objects with futures**

With aggressive prefetching strategies, binary objects may be transferred unnecessarily to the client application. To address this limitation, the ASP replaces large binary Data Objects with smaller *futures*. The binary objects are retrieved on demand only when they are used by the client.
Our prototype implementation focuses on images as a representative example of large binary Data Objects. Images constitute the largest payload of an average web page, and, especially with high-resolution images becoming common [Bra13], the amount of image data that is received by a client application often contributes significantly to the RAN traffic.

EdgeReduce’s static analyser automatically identifies Data Object fields that represent images, i.e. pointers to the UIImage class. It also identifies locations in the source code that initialise these fields using the image data received from a backend service. At the ASP, each image is assigned a unique image identifier to serve as a future for the image. Before it is sent to the client application, the image is added to an images dictionary, indexed by its corresponding identifier.

During processing the response of a backend API call at the ASP, when an image is used to initialise a Data Object field, it is replaced by the unique identifier. The client application intercepts object methods that are used to display an image on the screen, e.g. the drawRect method of the UIImageView class—a view-based Objective-C container for displaying and animating images. Before displaying an image, the client application requests the image from the ASP using the identifier as a reference.

An implementation challenge is how to modify methods of classes for which there is no source code, such as the built-in Objective-C methods for the UIImageView class. As a solution, we use method swizzling [Coc13]: when the Objective-C runtime loads a binary, all objects have their fields and method implementations defined in memory. These object templates also include a map associating method names with implementations. The Objective-C runtime allows for modifying these mappings at runtime by either replacing the original implementation with a user-defined function, thus modifying the default method behaviour, or changing the method name for a built-in Objective-C method.

EdgeReduce’s source-level compiler is thus able to patch existing methods with the replacement methods: the original implementation of, e.g. the method drawRect is changed to originalDrawRect; and the drawRect method is mapped to a new user-defined method that first retrieves the image given its future from the ASP and then calls the originalDrawRect method.

5.6 Evaluation

In this section, we evaluate experimentally the ability of the EdgeReduce approach to reduce RAN usage for a realistic set of mobile client applications. We also quantify the impact of EdgeReduce on the response times of client applications. We show that significant savings in network usage are achieved when applying EdgeReduce, with only a modest increase in application response time.
5.6.1 Experimental set-up

For our experiments, we use an Apple iPhone 4s to host the client application and a 2.26 Ghz Intel Core 2 Duo machine with 8 GB of RAM to host the ASP. We conduct experiments with two types of network connectivity between the nodes: (i) an IEEE 802.11g WiFi network with an average round trip time (RTT) of 23 ms and an average bandwidth of 8 Mbps; and (ii) a 3G mobile network with an average RTT of 425 ms and bandwidth of 0.4 Mbps.

We apply the EdgeReduce approach to three iOS client applications for Twitter, Groupon and Yahoo! Finance:

Twitter client [Mor09] (Twitter): This client application displays friends’ Tweet messages and supports the sharing of messages and the posting of new messages. It interacts with the Twitter platform via the Twitter REST API [Twi13], which provides interfaces for accessing and manipulating data related to Twitter users, such as timelines, followers and messages. We consider a workload in which the user retrieves the most recent Tweet messages.

Groupon client [Ara11] (Groupon): This client application displays popular Groupon deals and supports different purchasing options. The application interacts with the Groupon platform via the Groupon API [Gro13], which provides interfaces for categorised access to deals, such as location-aware deals and travel deals. Our workload constitutes of retrieving the latest deals in a given geographic location.

Yahoo! Finance client [Lyo13] (Yahoo): This application produces plots of stock quotes using financial data made accessible by the Yahoo! Finance platform via their API [Fin13]. The workload for this application involves retrieving and plotting the stock quote data for a set of stock symbols.

All HTTP traffic in the experiments is compressed using gzip. As in the case of CloudSplit, the results reported below are averaged values over 10 experimental runs. Due to the low variance between repeated runs we do not include error bars in graphs.

5.6.2 Network bandwidth usage

First, we compare the network usage of the unpartitioned and EdgeReduce-transformed versions of all three applications. We show the relevant reduction in RAN data traffic with respect to the data received by the mobile device, as well as data sent from the device to the backend service and the ASP, respectively. Figure 5.8 plots the breakdown of network usage, split according to image and non-image data, for the Twitter, Groupon and Yahoo clients, before and after transformation with EdgeReduce.

For the Twitter client, Figure 5.8(a) shows that, with respect to non-image data, EdgeReduce manages to reduce traffic by 2.1× due to the elimination of the XML-encoding overhead and returning only the portion of the data that is used by the client application. Though the amount of data sent to the mobile device is comparatively low, EdgeReduce
Network-Focused Offloading

![Figure 5.8: Bandwidth usage of unpartitioned and EdgeReduce-partitioned execution.](image)

...reduces it by a factor of $1.8\times$. This is due to the fact that, with EdgeReduce, HTTP requests are constructed and sent by the ASP, as opposed to the client application. In addition, EdgeReduce reduces the amount of image data received by the client application by a factor of $14.6\times$. This is accomplished by transmitting only 7% of the total images returned by the Twitter backend service when a user requests the most recent Twitter messages. For the remaining images, futures in the form of image identifiers are returned, which are significantly smaller in size (on average, 4 bytes per future versus 2.2 KBytes per image). Of course, depending on user activity, these savings may diminish in accordance to the number of images that the user decides to browse in the client application. In total, EdgeReduce is capable of offering reductions in RAN traffic ranging from $1.5\times$ to $3.2\times$ for the Twitter client based on application usage.

For the Groupon client, the results in Figure 5.8(b) indicate that EdgeReduce reduces the amount of non-image data returned to and sent from the mobile device by $4.5\times$ and $4.3\times$, respectively. EdgeReduce also achieves a reduction of the image data by a factor of $16.6\times$—only 6% of the images returned by the Groupon backend service are sent to the mobile device. In total, the Groupon client with EdgeReduce experiences reductions in RAN data traffic ranging from $1.4\times$ to $8.2\times$ based on user activity.

Finally, Figure 5.8(c) shows the results for the Yahoo client. EdgeReduce reduces the data received by the device by a factor of $2.2\times$, and the data sent from the device by a factor of $1.5\times$. The Yahoo client does not perform image transfers and therefore does not prefetch...
Figure 5.9: Response times of unpartitioned and EdgeReduce-partitioned execution.

In summary, a significant reduction in RAN traffic is obtained for all three client applications using EdgeReduce. Both the Twitter and Groupon clients exhibit similar reductions in image data transfers by avoiding the transmission of approximately the same number of images, which are returned by the corresponding backend API calls. However, the response to the Groupon API call for the most popular deals contains approximately twice as much unused data compared to the response to the Twitter API call for recent Tweet messages. This can be observed in the relevant savings for non-image data that EdgeReduce achieves in both cases.

In addition, the data sent from all three client applications is also reduced: data transmitted by remote calls from the client application to the ASP is less than what would have been transmitted using HTTP requests by the unpartitioned versions of the applications.

5.6.3 Application performance

Next we explore the impact of EdgeReduce on application response times. We focus on the increase in response time due to the communication overhead of the additional remote calls introduced by EdgeReduce, as well as the overhead of the on-demand retrieval of binary objects.
Application response time

In Figure 5.9, we compare the response times of the unpartitioned and EDGEREDUCE-transformed versions for the three client applications. The EDGEREDUCE-transformed versions are marginally outperformed by the unpartitioned versions, which is consistent across all applications.

We observe a reduction in performance that ranges from $0.90 \times$ to $0.95 \times$, which is due to the additional remote calls between the client application and the ASP. The majority of remote calls are used to copy transient Data Objects to permanent Data Objects on the client side, as well as return the futures of binary Data Objects to the client application. Due to the optimisation of transient Data Objects, as described in Section 5.5, the overhead incurred by the additional remote calls remains low.

In all cases, the impact of EDGEREDUCE on application response times is slightly less when the mobile device and the ASP are connected over a 3G network. This is due to the relative difference between the rate at which data is transmitted over WiFi and 3G networks. Gains in response time are obtained by avoiding the transmission of unnecessary data across the substantially slower 3G network. For WiFi networks, however, the achieved absolute saving is negligible—the contribution of the transmission time of unused data over the WiFi network to the overall response time is low.

On-demand object retrieval

To evaluate the impact of EDGEREDUCE’s future mechanism for controlling the aggressive prefetching of large binary Data Objects on application response time, we also observe the user-perceived latency when retrieving an image based on its future. We measure the time between a user action that causes the Groupon client application to display an image until the image was rendered on the screen. This includes the processing time at the ASP, i.e. the lookup time for the image object using its future, and the delay for transmitting the image.

As shown in Figure 5.10, for the WiFi network, requesting and transmitting a single image from the ASP to the mobile device takes 102 ms, out of which only 33 ms are due to the processing delay. Over the 3G network, the delay increases to 509 ms due to the higher
network latency—the processing delay remains the same. As expected, there is trade-off between reducing RAN network usage and providing the lowest application response times. Retrieving binary Data Objects on demand significantly reduces network usage, but it also degrades the user experience of the client application.

**Backend data size**

We also explore the effect of larger amounts of data exchanged between the client and the backend service on the application response time over a 3G network. We conduct an experiment that uses the Groupon client to transfer variable amounts of backend data, ranging from 50 KB–1.5 MB. To emulate different backend data transfers while retaining the original behaviour of the Groupon client, we maintain the same number of calls but vary their data sizes. We further assume the same ratio of used to unused backend data as reported in Section 5.6.2, thus yielding minimum and maximum savings of $1.4\times$ and $8.2\times$, respectively.

Figure 5.11 shows the time to transmit different amounts of backend data in the unpartitioned and the EdgeReduce-transformed versions of the Groupon client, assuming minimum and maximum data reduction. As the amount of backend data increases, the relative increase in application response time is greater for the unpartitioned version, followed by the EdgeReduce transformed version with minimum savings, and the EdgeReduce-transformed version with maximum savings. This is caused by the fact that the overhead of additional remote calls between the ASP and the client is gradually amortised by the savings in data transmission time over the slow network. Eventually, these savings are enough to mask completely the remote call overhead and achieve a speedup in execution.

We conclude that, for client applications that retrieve large amounts of data from backend services over a 3G network, EdgeReduce can speed up execution by taking advantage of the richer network capability of the ASP compared to the mobile device. This is only the case when the gains of transmitting data over a higher bandwidth network between the ASP and the backend service outweigh the cost of additional remote calls between the mobile device and the ASP.
5.7 Discussion

Based on our experiences with current mobile client applications, the limited availability of compute and memory resources in mobile devices does not create performance bottlenecks. Their duties are confined to presenting information maintained by backend services to users and thus they can be characterised as computationally-inexpensive applications. We find that the most computationally-intensive functionality of client applications is performed by Data Parser objects, which deserialise API response data. As part of the operation of EdgeReduce, such objects are offloaded to the remote node by design.

Nevertheless, EdgeReduce is designed for compatibility with CloudSplit, thus allowing for also addressing the compute and memory limitations of mobile devices if required. Combining the two approaches involves: (i) using EdgeReduce to identify the objects that need to be offloaded to the remote node to reduce RAN traffic, i.e. Communicator and Data Parser objects; and (ii) using CloudSplit with minor modifications to incorporate partitioning constraints for placing Communicator and Data Parser objects on the remote partition to partition application state (Data Objects). Therefore, combining EdgeReduce with CloudSplit can result in offloading gains with respect to compute, memory and network resources for applications that combine basic client functionality with compute- and/or memory-intensive processing on the data retrieved by backend services.

5.8 Summary

In this chapter, we make the observation that mobile client applications receive unnecessary data from their Internet backend services due to a semantic mismatch of the granularity of API calls and the unnecessary prefetching of data. We presented our network-focused offloading approach for generating application-specific proxies (ASPs) for mobile client applications that blurs the boundary between mobile end-systems and networks. ASPs host the application logic that receives response data from an Internet backend service and converts it into application objects, which are then sent to the mobile device. As a result, discarded data from the backend service is never transmitted over the RAN. To reduce network usage, we described optimisations that allow ASPs to minimise the number of remote calls to the client and to retrieve large binary objects on-demand.

Our network-focused approach is realised by EdgeReduce, a code offloading system for Objective-C applications on the iOS platform. We evaluated EdgeReduce on three real-world mobile client applications and showed that it significantly reduces RAN traffic by up to 8.2×. The overheads introduced due to the additional remote calls remains less than 10% of the performance of the unpartitioned version of the application.
Considering the rate at which mobile devices are taking over the world of IT, there can be no doubt that we face a future of technological advancements ruled by mobile computing. Nevertheless, despite the evolution of mobile technologies, form factor constraints undermine the computational capabilities of mobile applications, which remain inferior to applications developed for more powerful stationary machines.

With the rise of cloud computing, new opportunities for addressing the limitations of mobile devices present themselves. Recently, code offloading systems that automatically partition applications to execute across resource-constrained mobile devices and more capable cloud-based machines have been proposed. These allow applications to offload computationally intensive code to more powerful execution environments for faster and more energy-efficient execution.

However, we identified common limitations of such approaches, which we tried to address in this thesis. First, existing approaches only focus on leveraging the faster CPUs of remote nodes, ignoring other resources that can be exploited such as larger amounts of main memory and richer networking capabilities. Second, they incur significant runtime overheads due to the way that they handle application state by keeping a copy of the entire state on the mobile device and migrating state with every offloaded operation. In their majority, they also favour adaptability of offloading policies at runtime over more efficient offloading techniques that statically partition applications across nodes. In addition, they require support from the application’s programming language and runtime environment in terms of type safety and code mobility. Such features, however, are only available on managed mobile platforms. Therefore, current approaches neglect applications on less-managed platforms and possibly low-end devices with severe hardware constraints for which cloud-assisted execution is most beneficial.

This thesis set out to take advantage of all three main resources, i.e. compute, memory and network, available on cloud-based infrastructures to address the aforementioned limitations. We managed to extend the objectives of current offloading solutions beyond just reduced application response time and energy consumption to include: (i) main memory beyond the
limited capacity of mobile devices; and (ii) reduced network traffic in congested mobile networks. Our proposed approaches incur less runtime overhead and lift assumptions regarding programming language and runtime support. They achieve these goals partly due to the fact that they employ static partitioning techniques, which do not adapt to environment changes at runtime.

In particular, we proposed three new static offloading approaches to achieve enhancements related to the limited compute, memory and network resources on mobile devices. We presented these approaches, showing how each builds on top of the other to realise a code offloading solution that is capable of enhancing mobile capabilities on all fronts described above. An overview of each approach and its contributions is provided in the thesis summary that follows.

6.1 Thesis summary

We begun this thesis with an overview of (i) current mobile environments, (ii) previous work on supporting distributed mobile application development and (iii) automatic code offloading approaches, which realise the benefits of seamless cloud-assisted execution.

We first compared different classes of computing devices with respect to CPU, memory and network capabilities to show that mobile hardware is restricted compared to desktop hardware. This motivated research work towards overcoming mobile performance limitations through cloud-assisted execution. We described the Android and iOS mobile platforms, focusing on the support to code offloading systems. We compared the two on the basis of the programming language constructs and runtime facilities that they support. We showed that existing approaches cannot be applied to applications running on iOS devices, mainly due to the lack of support for strong type safety and code mobility. We further described the current architecture of mobile networks and identified performance bottlenecks in 3G/4G networks. These affect both code offloading systems that use such networks to offload computation with associated state to remote nodes, as well as mobile client applications that retrieve data from Internet backend services. We discussed different transport- and application-layer optimisations for addressing the latency and bandwidth limitations of such networks, as well as web caching techniques to reduce data transfers over congested communication channels.

We next gave an overview of mobile middleware platforms and cyber-foraging approaches, which support distributed mobile application development. These provide specialised APIs and application frameworks to ease the task of code distribution. The former focus on distributed mobile execution in general, such as supporting reflection and context awareness concepts, efficient communication between cooperating mobile units and data sharing. The latter specialise in taking advantage of powerful machines in the vicinity of mobile devices to increase the performance of resource-constrained devices. Both types of solutions provide the mechanisms to allow for distribution to be realised easily. Defining the policies to be followed at runtime is left at the developer’s discretion. Thus, developers need to familiarise
themselves with new programming abstractions and application architectures, as well as have a good understanding of the requirements of the components of their applications.

We concluded the background chapter with a discussion of the more relevant approaches on automatic code offloading. These take mobile applications as input and partition them across a mobile device and a remote node with minimal to no developer involvement. We provided a classification of the different approaches proposed to date according to the partitioning granularity considered. We identified the weaknesses in each such approach, which include making assumptions about high-bandwidth ubiquitous network connectivity with no provisioning for network failures, assuming already modularised applications, incurring significant runtime overheads, relying on properties of managed programming languages with substantial runtime support, or any combination of the above.

The next three chapters presented our research work in the area of mobile code offloading. We described three new offloading approaches to address the limitations of the current state of the art. First, we presented our compute-focused offloading approach, which is realised by the AnyWare system. It statically partitions applications across a local and a remote node to reduce application response times. The partitioning is decided based on a fine-grained characterisation of the application’s runtime behaviour, which is obtained by means of offline dynamic profiling. It explicitly targets applications written in unmanaged programming languages with only basic runtime support.

Second, we presented our memory-focused offloading approach, which is realised by the CloudSplit system. This is a static partitioning approach that leverages both the increased memory and faster compute resources of a remote node to alleviate device memory constraints while also reducing application response times. We targeted applications written in the Objective-C programming language, running on the iOS platform with only rudimentary runtime support.

We introduced the concept of code offloading with application state partitioning that splits application state between the mobile device and a remote node: objects are placed permanently either on the local or the remote node; proxy objects are used to represent remote objects, which relay method calls using RPC. This allows applications to consume more main memory than afforded by the local node. It also reduces the amount of data that is transferred with repeated offloaded calls and eliminates the runtime overheads associated with frequent state migration. We described how a partitioning is chosen, again based on offline dynamic profiling and using an optimisation-based partitioning algorithm.
We also presented a new *snapshot-based fault tolerance* mechanism to allow the mobile device to recover remote state after network failure. It offers a choice between two strategies: a *synchronous* strategy that takes a snapshot of the application state after each offloaded call, allowing failed remote calls to re-execute locally after failure; and an *asynchronous* strategy that allows more sporadic snapshots transmitted asynchronously to the mobile device: application state is rolled back to the last complete snapshot after failure, and local execution resumes from an earlier point in time. We then presented a new *user-level virtual memory scheme* to support application state sizes that are larger than the device’s main memory after failure: remote objects are loaded into main memory from snapshots only when accessed by the application. All of the above are implemented without requiring modifications to iOS or Objective-C. Finally, we presented the results of our evaluation after applying CLOUDSPLIT to two real-world iOS applications. We showed that with CLOUDSPLIT, we are able to support application workloads that consume more memory than what is provided by the mobile device, while also outperforming existing state migration approaches by reducing application response times by a factor of 15×.

Third, we described our *network-focused* offloading approach, which is realised by the EDGE-REduce system. It reduces the radio access network (RAN) traffic caused by mobile client applications to benefit: (i) mobile users by reducing data charges and (ii) network operators by relieving contention in RANs. It offloads the code that processes the data returned from Internet backend services at the network edge.

We first motivated our approach by identifying opportunities for data traffic reduction based on unused data returned by backend service APIs. This comprises data returned by API calls that is never used by client applications, encoding overheads of inefficient network protocols and unused binary objects retrieved due to aggressive prefetching strategies. We described a baseline architecture for current client applications and discussed how it can be exploited to derive a partitioning that avoids transmitting unused response data to mobile devices. Using static code analysis, we explained how the application components that process response data can be inferred and offloaded to the remote node.

We then described two optimisations to reduce the number of remote calls after partitioning: coalescing multiple identical method calls into a single call when possible and using transient remote replicas to initialise local data objects remotely. We also introduced a methodology for tuning aggressive prefetching strategies by replacing large binary objects with smaller futures. We evaluated EDGE-REduce using three real-world iOS applications, showing that it manages to achieve reductions of up to 8.2× in RAN traffic with only a modest impact on application response times.

### 6.2 Future work

For future work, we plan address some of the limitations of our approaches stated throughout the thesis. We also want to investigate more intelligent remote execution paradigms (e.g.
parallel and speculative execution) to improve further the performance gains achieved by code offloading. Lastly, we will also explore the different challenges pertaining to the support required from service providers to facilitate code offloading. Next we discuss four different research directions along these lines.

**Support for multi-threaded applications**

The majority of current code offloading approaches only support single-threaded mobile applications. To the best of our knowledge, the only offloading system that supports multi-threading is COMET [GJM+12]. In order to achieve this, it requires modifications to the Android Dalvik VM for managing DSM operations, which limits the adoption of such an approach. Others (e.g. CloneCloud [CIM+11]) claim to support multi-threading by letting local threads continue execution until shared state is accessed. It is unclear, however, how this avoids deadlock situations when an offloaded thread waits for data from a local thread, suggesting a flaw in this type of designs.

In Sections 3.6 and 4.6, we explained why multi-threading is currently not supported by our offloading approaches. Put concisely, the problem with our compute-focused approach is the same with that described for CloneCloud. For the memory-focused approach, application state partitioning ensures that only a single (distributed) copy of the application state is active during execution. The fault tolerance mechanisms employed, however, assume that only a single partition executes at any point in time. This ensures that local and remote snapshots—taken at precise points during execution—can be combined in order to reconstruct a consistent image of the entire application state. Multi-threading, however, violates this assumption.

We plan to incorporate alternative log-based recovery techniques [Bar81] that keep track of state updates caused by different threads at runtime. By being able to distinguish between updates caused by local threads and incoming remote calls, one can effectively roll-back execution to a consistent state snapshot after failure without assuming synchronisation of local and remote state snapshots on the basis of remote call invocations.

**Support for automatic parallel remote execution**

Previous work has proposed remote execution techniques that execute multiple VM images in parallel to handle a parallelisable task, thus offering enhanced performance [KAH+12]. They, however, assume applications built with this intent in mind, i.e. programmers to incorporate parallel algorithms into their designs, or at least have a good understanding of the application logic to transform manually application code to be parallel.

Our goal is to automate this process by statically analysing applications to identify such code, and automatically rewriting it to be parallel. For example, in the case of the Latrunculi game [MB06] used for evaluating CloudSplit, the code responsible for parsing the search tree of possible future board configurations in order to choose the next move is a good candidate for parallel execution. We plan to exploit simple techniques for automatically identifying application code that abstractly exhibits a high degree of parallelism, which thus could be executed remotely in parallel. An example would be to use a combination of loop
unrolling techniques [DJ01] for unwinding loops to expose greater levels of parallelism with approaches for automatically identifying parallelisable loops [OAN00] to be optimised to run on multiple threads.

**Support for speculative remote execution**

We also envision extending our current offloading approaches by adding artificial intelligence (AI) options to the remote node for keeping track of an application’s workload history. This way, the system could anticipate user actions at runtime and employ speculative execution techniques [TFNG08] to return immediately from remote calls when they are triggered by a user action. This is a different form of parallel execution that exploits the remote node’s computational resources when idle. It assumes, however, that usage patterns can be extracted from the execution history in order to predict the application’s behaviour at runtime.

**Server-side challenges of code offloading**

While mobile code offloading is gaining traction, for it to be provided as a service by mobile providers multiple challenges need to be addressed first. We envision a future where nodes collocated with mobile base stations will assume the roles of remote nodes in the scenarios that we consider. A challenge is to allow for uninterrupted partitioned execution even when mobile users switch between different mobile cell sites during execution. We will explore how application hand-overs between adjacent base stations can seamlessly allow for cloud-assisted execution to persist user mobility, i.e. without terminating connections and losing remote application state. Such mechanisms need to be efficient so that they avoid degrading application performance when users roam frequently during execution.
Bibliography


ddings of the 2007 Conference on Applications, Technologies, Architectures, and


[FPS02] Jason Flinn, SoYoung Park, and M. Satyanarayanan. Balancing Performance,

Energy, and Quality in Pervasive Computing. In Proceedings of the 22nd Inter-


[GC04] Sachin Goyal and John Carter. A Lightweight Secure Cyber Foraging Infrastruc-
ture for Resource-Constrained Devices. In Proceedings of the Sixth IEEE Work-


[Gel85] David Gelernter. Generative Communication in Linda. In Transactions on


[GJM+12] Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and

Xu Chen. COMET: Code Offload by Migrating Execution Transparently. In Proceed-
ings of the 10th USENIX Conference on Operating Systems Design and


Mobile Computing. Technical report, Carleton University Systems and Com-


mecast/, 2013.


[GSM12a] GSMArena. Amazon Kindle Fire HD 8.9 LTE. http://www.gsmarena.com/ama-


