UNIVERSITY OF MANNHEIM

Computation Offloading for Fast and Energy-Efficient Edge Computing

Inaugural Dissertation

to Obtain the Academic Degree of a Doctor in Business Administration at the University of Mannheim

submitted by

Martin Breitbach, M.Sc.

Mannheim

Dean: Joachim Lutz
Referent: Prof. Dr. Christian Becker
Correferent: Prof. Dr. Torben Weis

Day of oral examination: February 18, 2022

Abstract

In recent years, the demand for computing power has increased considerably due to the popularity of applications that involve computationally intensive tasks such as machine learning or computer vision. At the same time, users increasingly run such applications on smartphones or wearables, which have limited computational power. The research community has proposed computation offloading to meet the demand for computing power. Resource-constrained devices offload workload to remote resource providers. These providers perform the computations and return the results via the network. Computation offloading has two major benefits. First, it accelerates the execution of computationally intensive tasks and therefore reduces waiting times. Second, it decreases the energy consumption of the offloading device, which is especially attractive for devices that run on battery. After years in which cloud servers were the primary resource providers, computation offloading in edge computing systems is currently gaining popularity. Edge-based systems leverage end-user devices such as smartphones, laptops, or desktop PCs instead of cloud servers as computational resource providers. Computation offloading in such environments leads to lower latencies, better utilization of end-user devices, and lower costs in comparison to traditional cloud computing.

In this thesis, we present a computation offloading approach for fast and energy-efficient edge computing. We build upon the Tasklet system — a middleware-based computation offloading system. The Tasklet system allows devices to offload heterogeneous tasks to heterogeneous providers. We address three challenges of computation offloading in the edge. First, many applications are data-intensive, which necessitates a time-consuming transfer of input data ahead of a remote execution. To overcome this challenge, we introduce *DataVinci* — an approach that proactively places input data on suitable devices to accelerate task execution. *DataVinci* additionally offers task placement strategies that exploit data locality. Second, modern applications are often user-facing and responsive. They require sub-second execution of computationally intensive tasks to ensure proper user experience. We design the decentralized scheduling approach *DecArt* for such applications. Third, deciding whether a local or remote execution of an upcoming task will consume less energy is non-trivial. This decision is particularly challenging as task complexity and result data size vary across executions, even if the source code is similar. We introduce the energy-aware scheduling approach *Voltaire*, which uses machine learning and device-specific energy profiles for making precise offloading decisions. We integrate *DataVinci*, *DecArt*, and *Voltaire* into the Tasklet system and evaluate the benefits in extensive experiments.

Acknowledgments

I still remember how I more or less randomly applied for the Bachelor's seminar at the chair. On the last day before the deadline, without knowing any of the supervisors, without having attended a single lecture. I never imagined that this would lead to the journey that is reaching a milestone today as I write these final words of my PhD thesis. I would like to thank all the people who supported me during this incredible time.

First, I would like to thank Prof. Dr. Christian Becker for all the support throughout the years. A lot has changed since you asked me whether I know the four deadlock criteria in 2015. Thank you for being my supervisor and for managing so many things in the background, which made it possible for us PhDs to always focus on our progress. Apart from your "professional" role as my boss for a quarter of my life, I would also like to thank you for becoming a good friend. Thank you for all the memories: For the famous wine evenings at the chair, the delicious lunches, the biplane flights, and the trips to Kyoto & Singapore. I will miss all of this very much.

I would like to thank Prof. Dr. Torben Weis for immediately agreeing to be my second examiner. It is a pleasure for me to discuss my work with "one of the most intelligent people I know" (quote C. Becker). I would also like to thank Prof. Dr. Markus Strohmaier for joining the board of examiners.

A big thank you to all my colleagues at the chair. You made these last years special for me! Thank you to Prof. Dr. Patricia Arias Cabarcos, Prof. Dr. Janick Edinger, Melanie Feist, Kerstin Goldner, Melanie Heck, Benedikt Kirpes, Dr. Sonja Klingert, Prof. Dr. Christian Krupitzer, Markus Latz, Michael Matthé, Dr. Jens Naber, Dr. Martin Pfannemüller, Dr. Felix Maximilian Roth, Dr. Dominik Schäfer, and Anton Wachner. Thank you to the "old generation" (Patricia, Jens, and Max) for making the transition from student assistant to PhD student easy for me. And thank you to Melli and Michael for bringing a breath of fresh air to the chair. Thank you Kerstin for helping me through all the struggles with paperwork. Thank you Melanie for proofreading the important parts of this thesis and thank you for showing the world that management students are the better computer scientists. It's amazing how you write entire letters of recommendation faster than I write my signature. Thank you Sonja for always brightening the atmosphere at the chair with your smile. Thank you Markus for being my ally in my fights with the hp support. A big thank you to Martin for the amazing REACT side project and — more importantly — for being a close friend. You were missed a lot at the chair after your graduation.

It would have been impossible to get this far without my three “mentors”. I would like to thank Pitzi (aka Prof. Dr. Christian Krupitzer) for all the cooperation from the first day as a student assistant to today. Thank you for hiring me, for believing in me from the start, and for all the amazing opportunities, even after you started having your own group. That meant the world to me. I would like to thank Dominik (Dr. Schäfer) for being like an older brother for me. Thank you for introducing me to the Tasklet project, for the amazing writing phase when we easily met the PerCom’19 deadline, and for good advice on whatever crossed my mind. You will always be an idol for me. A special thank you to Janick (Prof. Edinger). I couldn’t have done this without you. Your energy and your passion for research are unmatched. Thank you for always motivating me, for all the night shifts, for discussing every idea with me — no matter how crazy — and of course for your friendship. I am so proud of what we have achieved together.

Thank you to my co-authors, also from other groups and universities. Thank you Dr. Samy El-Tawab, Niklas Gabrisch, Prof. Dr. Renato Lo Cigno, Siim Kaupmees, Prof. Dr. Samuel Kounev, Veronika Lesch, Prof. Dr. Amr Rizk, Johannes Saal, Dr. Bradley Schmerl, Prof. Dr. Andy Schürr, Dr. Michele Segata, Prof. Dr. Gregor Schiele, Prof. Dr. Heiner Stuckenschmidt, Dr. Timo Sztyler, Prof. Dr. Sven Tomforde, Heiko Trötsch, Dr. Sebastian VanSyckel, and Dr. Markus Weckesser. This work was supported by the German Research Foundation (DFG) under grant BE 2498/10-1 “Tasklets: Ein Ansatz für Best-Effort Computing”. Thank you to all students who supported the Tasklet project while writing their thesis with me or working as a student assistant. They are: Jan-Nicklas Adler, Maximilian Barth, Niklas Bauer, Maurice Bürkle, Philipp Drayß, Yves Ekspenszid, Liam Goodman, Siim Kaupmees, Mika Koalick, Susanne Koch, Martin Koller, Vladislav Kozhevnikov, Jonathan Lersch, Luca Mohme, Alicia Rose, Paul Rößling, Sarah Schmitt, Melissa Speer, Quirin Stahuber, and Heiko Trötsch.

Thank you to my family and friends who have always been there for me. Thank you to my parents Monika and Thomas for the unconditional love. You always have my back and you are the reason I can pursue my dreams without any worries. I love you so much. Thank you to my amazing brother Christian. You are the most funny person I know. Sharing (almost) all my passions with you is the greatest thing, even when we’re both upset that our team once again didn’t manage to get promoted. You will be my best friend forever. I would also like to thank my grandparents Karola, Margot, and Paul. You are all the reason why I love coming back to Nickenich whenever possible. Thank you to all my friends, especially the how-soon gang (Cousienchen, Domme, Felix, and Khaki), Freddy, Kathrin, and Vio. Thank you for all the laughs, for the trips, and for showing me that there is a life outside of the chair. Finally, thank you to my wonderful girlfriend Lily. It’s difficult to express on paper how much you mean to me. Your support helped me a lot while writing this thesis. I already share so many beautiful memories with you and I’m looking forward to a future together. I love you.

Contents

List of Figures

List of Tables

List of Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
AR	Augmented Reality
CAN	Content-Addressable Network
CPU	Central Processing Unit
GPU	Graphics Processing Unit
IaaS	Infrastructure as a Service
IoT	Internet of Things
JVM	Java Virtual Machine
LRU	Least Recently Used
NIST	National Institute of Standards and Technology
OS	Operating System
PaaS	Platform as a Service
PDA	Personal Digital Assistant
QoC	Quality of Computation
QR	Quick Response
RGB	Red, Green, Blue
RPC	Remote Procedure Call
SaaS	Software as a Service
TCP	Transmission Control Protocol
TVM	Tasklet Virtual Machine
UI	User Interface
VM	Virtual Machine
VR	Virtual Reality

1. Introduction

Mark Weiser's vision of a world where computers unobtrusively surround and serve us [1] has in parts become reality. Nowadays, we interact intuitively with multiple computers per day. In addition, the individual devices have become more powerful. Modern smartphones have more computing power than most computers had when Weiser formulated his thoughts in 1991. At the same time, the demand for computing power of applications has grown analogously. Applications in the areas of machine learning, Augmented Reality (AR), or computer vision, for instance, exceed the capabilities of many devices [2]. Users face waiting times and quick battery drain while running these applications, especially on mobile devices such as smartphones. Thus, in such situations computing power remains a scarce resource. This is a paradox: although today's computing landscape — where dozens to hundreds of mostly idle devices are connected in distributed systems — offers more than sufficient computing power, the execution of computationally intensive applications is limited to the processor cores of the user device only.

Inspired by the above observations, the research community has introduced computation offloading [3–5]. Devices that run computationally intensive applications — the so-called consumers — offload tasks to other devices that act as remote resource providers. The providers perform the computation for the consumers and return the results via the network. This makes the cumulative computing power of a whole distributed system available to every device. Computation offloading has three main benefits. First, it accelerates the execution of computationally intensive tasks if the provider has more computing power than the consumer or if multiple providers work on the same task simultaneously. In this way, computation offloading reduces waiting times and improves the user experience. Second, computation offloading decreases the energy consumption of the consumer device if transferring a task and receiving the results consumes less energy than executing the task locally. Making applications more energy-efficient is especially attractive for mobile devices such as smartphones that run on battery power.

Third, computation offloading increases device utilization as idle computers can perform computations for others. This sharing of resources increases sustainability. In the future, it may be sufficient to own fewer and less powerful devices.

Computation offloading has a long history (cf. [6, 7]). Volunteer computing projects such as *SETI@home* [8–10], which searches for extraterrestrial life, and *Folding@home* [11], which drew large media attention during the COVID-19 pandemic, are two popular example projects. The advent of cloud computing [12–14] made computation offloading accessible to a broad public. While cloud computing certainly transformed the way businesses and private persons use computers, it suffers from several drawbacks. Cloud resources are typically (i) comparably expensive, (ii) controlled by a central organization, and (iii) topologically far away from the users, which introduces considerable latencies. In recent years, the edge computing paradigm [15–17] has therefore emerged as a complement to cloud computing. Edge computing moves computation from the "core" (e.g., from cloud data centers) to the "edge" (e.g., desktop PCs) of the network. Thus, end-user devices such as PCs or laptops are typical resource providers in edge computing systems. Performing computation offloading in such environments leads to lower latencies and higher utilization of end-user devices.

1.1. Problem Definition

Implementing computation offloading in edge computing environments is challenging. In contrast to cloud computing, the pool of resources that act as providers is highly heterogeneous in terms of hardware, Operating System (OS), software, and network connection [18]. Various device types including servers, desktop PCs, laptops, smartphones, and even wearables participate in a single resource sharing system. As a majority of these devices is user-controlled, fluctuation is common. Devices join and leave the system spontaneously when, for instance, users shut off devices or require the whole computing power for their own purposes. Resources in edge computing systems are therefore unreliable. The diversity of today's applications adds further complexity. Modern applications are written in different programming languages and with varying requirements for the execution of offloadable application parts. While some applications require a reliable remote execution, others are particularly time-critical, or operate on large data sets.

The demand for computing power changes continuously depending on the user behavior. Thus, a proper allocation of tasks to remote resource providers in the edge is non-trivial.

In this thesis, we design a computation offloading approach with a strong focus on edge computing. Despite the above challenges, consumers shall be able to seamlessly exchange tasks and results with heterogeneous resource providers. We focus on the design of task placement strategies that are capable of achieving application-specific requirements such as fast execution or energy efficiency under the special conditions in the edge.

1.2. Research Questions

We derive four research questions from the above problem statement. The objective of this thesis is to design a computation offloading approach for fast and energy-efficient edge computing. To achieve this, developing a system that is able to offload tasks originating from heterogeneous applications to heterogeneous devices is the first step, which leads to the following research question:

Research Question 1: *How to design a computation offloading system that enables heterogeneous applications to execute tasks on heterogeneous providers?*

As far as task completion times are concerned, computation offloading is in general attractive for computationally intensive tasks with small amounts of input data. Today, many tasks such as classification or face recognition are, however, data-intensive. In such cases, the task completion time is dominated by the input data transfer from consumer to provider, which has to take place before starting the remote execution. Nonetheless, computation offloading would — neglecting the data transfer — be beneficial for these applications, too, since they require considerable computing power. As we aim to design a computation offloading system that is beneficial for all applications, we derive the research question:

Research Question 2: *How can computation offloading in the edge reduce the completion times of data-intensive tasks?*

Many applications such as speech recognition or games are user-facing and, thus, have to be responsive. Users generally perceive waiting times of up to one second as acceptable. While computation offloading can help to meet relevant deadlines thanks to the choice of fast providers or parallelization, achieving sub-second task completion times in the edge remains a challenge. Sending a message to a remote provider takes dozens of milliseconds, which has a comparably high influence on short tasks. In addition, task abortions due to provider churn or excessive queuing before execution are not compensable within the deadlines. Motivated by these observations, we formulate the third research question:

Research Question 3: *How can computation offloading in the edge reduce the completion times of tasks with sub-second deadlines?*

Apart from lowering task completion times, computation offloading can also reduce the energy consumption of the consumer device. The decision whether a certain task should be executed locally or remotely to minimize the battery drain is, however, non-trivial in edge computing systems. A variety of context factors such as task complexity, input and result data size, available bandwidth, and hardware of the consumer device have an influence on whether a local or remote execution consumes less energy. Accordingly, we state the final research question:

Research Question 4: *How can computation offloading in the edge minimize the energy consumption of the consumer device?*

1.3. Contributions

This thesis contributes to the state of the art with the design, implementation, and evaluation of a computation offloading approach for the edge. The basic artifact is the *Tasklet system* — a middleware-based computation offloading system. Consumers offload workload in form of Tasklets. Tasklets are self-contained units of computation that comprise everything required for a remote execution including source code and input data. Providers run Tasklet Virtual Machines (TVMs) that abstract from the local hardware and allow the execution of Tasklets on a wide range of edge devices. With the help of a well-defined Application Programming Interface (API), application programmers can launch Tasklets and receive results from several programming languages. They can specify Quality of Computation (QoC) goals, which are task-specific requirements such as reliable, fast, or energy-efficient execution. The Tasklet Middleware enforces these QoC goals transparently with multiple QoC mechanisms. The design of the Tasklet system answers research question 1.

The main contribution of this thesis are three QoC mechanisms tailored to edge computing environments. The first QoC mechanism is *DataVinci* — a proactive data placement approach for data-intensive tasks. *DataVinci* transfers input data to one or multiple providers in advance. In this way, it prevents time-consuming ad-hoc data transfers and prepares providers for an immediate start of the Tasklet execution. *DataVinci* therefore answers research question 2. *DecArt* is the second QoC mechanism. It is a decentralized scheduling approach for tasks with sub-second deadlines. *DecArt* allows consumer to make independent task placement decisions without prior coordination with other devices or a central resource manager. This considerably accelerates Tasklet completion times — especially for sub-second tasks — and answers research question 3. The third QoC mechanism is *Voltaire*. It minimizes the energy consumption of the consumer device and therefore answers research question 4. *Voltaire* predicts the complexity and the result data size of an upcoming task. In addition, it monitors other context dimensions such as the current bandwidth to decide whether a local or remote execution of a Tasklet consumes less energy.

1.4. Structure

This thesis consists of ten chapters. Following the introduction in this chapter, Chapter 2 gives an overview of distributed computing, computation offloading, and context-awareness. Chapter 3 derives seven functional and five non-functional requirements for computation offloading in edge computing systems. In Chapter 4, we review related work and identify a research gap since no existing approach fulfills all requirements. To close this research gap, we present the Tasklet system for computation offloading in Chapter 5. This includes a brief presentation of the prototypical implementation. Chapters 6 to 8 contain the core contributions of this thesis. These chapters introduce and evaluate the approaches *DataVinci* (Chapter 6), *DecArt* (Chapter 7), and *Voltaire* (Chapter 8) that serve as QoC mechanisms in the Tasklet system. In Chapter 9, we discuss whether the Tasklet system fulfills the requirements. We conclude the thesis in Chapter 10.

2. Fundamentals

This thesis introduces a novel computation offloading approach for distributed computing systems with a special focus on edge computing environments. In this chapter, we present the necessary fundamentals for this. First, we briefly summarize the most relevant distributed computing paradigms in Section 2.1. This illustrates the variety of environments in which computation offloading is applied. Based on this knowledge, we discuss the concept of computation offloading in more detail in Section 2.2. We concentrate on the decision making perspective and present the typical design alternatives for offloading approaches. In addition, we show that modern offloading systems profit from context-awareness. This leads us to Section 2.3 in which we provide theoretical background about context-aware and (self-)adaptive systems, also from adjacent research domains.

2.1. Distributed Computing

Modern computers form *distributed systems.* Van Steen and Tanenbaum define a distributed system as "*a collection of autonomous computing elements that appears to its users as a single coherent system*" [19, p. 968]. This definition highlights two characteristics of such systems that are essential for this thesis. First, a distributed system consists of independent computers which may be built for different purposes, equipped with different software, and owned by different parties. Second, this potentially complex and heterogeneous system ideally provides value to users as a transparent unit. Hiding the inherent complexity of a distributed system is also a major design goal of this thesis.

The purpose of a distributed system varies in practice, from sharing storage to communicating with geographically dispersed users. In this thesis, however, we focus on distributed systems designed for *distributed computing.* Distributed computing allows devices to harvest computing power from other devices in the system. In this way, it is possible to augment the computational capabilities of a device. As the alternative is to invest in more powerful hardware, distributed computing is common in today's computing landscape. Some applications — for instance in the scientific domain where petabytes of data have to be analyzed [20] — even require more computing power than the strongest supercomputer has to offer. Distributed computing is the only option to cope with such use cases.

In a distributed computing system, devices participate either as *consumers*, as *providers*, or as a combination of both [21]. A consumer runs one or multiple computationally intensive applications and desires to harvest computing power. The providers perform the computation for the consumer and return the results via the network. This high-level organization is the basis of every distributed computing system. Some approaches additionally require devices that act as *brokers*. These central nodes have special responsibilities such as resource management or task placement, depending on the respective approach. Over the decades, several distributed computing paradigms with different interpretations of this basic organization have been trending. In the following, we briefly summarize these paradigms that shape our understanding of distributed computing.

2.1.1. Cluster Computing

Cluster computing is a well-established paradigm in which tightly coupled off-the-shelf computers form a powerful distributed computing system [22]. Usually, the nodes of a cluster are rather homogeneous and connected via a high-speed network. In contrast to devices in grid or edge computing, which are typically owned by end-users, cluster nodes are dedicated resources with the sole purpose to be a provider in the cluster, often managed in large data centers. The size of a cluster varies from several computers, e.g., owned by a university department, to medium-sized clusters such as the *bwHPC* cluster[1] that was used for the simulations in this thesis to large-scale clusters at Microsoft [23] or Google [24].

While energy-awareness certainly is an influential topic in cluster computing research [25], the main focus of the community is the reduction of response times and load balancing. From a software perspective, Google's *MapReduce* [26, 27], which is also available under the name *Hadoop* as an open-source implementation, is highly influential. *MapReduce* is an easy-to-use parallel programming model

[1]`https://www.bwhpc.de/cluster.php`, accessed 2021-10-12

that is frequently applied in domains such as machine learning [28] or DNA analysis [29]. Researchers have presented a variety of cluster schedulers that have been implemented successfully in practice, including *Spark* [30], *Borg* [24], and *Dryad* [31]. More recent efforts such as *Firmament* [32] prove the ongoing relevance of cluster computing. Our decentralized scheduling approach *DecArt* is largely inspired by cluster scheduling research (cf. Chapter 7).

2.1.2. Grid Computing

The term *grid computing* was coined in the 1990s [33]. Similar to the electrical power grid, the idea of grid computing is to supply computing power to everyone whenever required. The computing power is offered by heterogeneous, loosely coupled computers from different organizations. This view on distributed computing is diametrically opposed to cluster computing where resources are homogeneous, tightly coupled, and usually controlled by a single organization. Devices participating in grid computing join *virtual organizations*. To harvest computing power, the members of a virtual organization perform resource sharing. Grid computing therefore bases on the reciprocal agreement that consumers will eventually serve as providers. There is no clear distinction between consumers and providers unlike in cluster computing.

One of grid computing's major challenges is to overcome the prevailing heterogeneity. Thus, members of a virtual organization have to agree upon software and protocols [12]. Among the most popular grid computing software for this purpose is *HTCondor* [6, 34], which is still regularly updated and extended today[2]. *HTCondor* offers resource management and scheduling for the grid. It is compatible with the *Globus* toolkit, another grid computing project with a long history [35, 36] that is still commonly used today[3]. *Globus* offers lower-level services for grids such as remote data access or multicast communication services that implementations like *HTCondor* can build upon. Other prominent grid computing systems from the literature are *Nimrod/G* [37] and *AppLeS* [38]. Grid computing is relevant for this thesis as (i) grid scheduling approaches usually cope with heterogeneous resources (cf. surveys in [39, 40]) and (ii) data-intensive

[2]`https://research.cs.wisc.edu/htcondor/`, accessed 2021-10-12
[3]`https://www.globus.org/`, accessed 2021-10-12

applications are challenging in such environments. Transferring large amounts of input data in a grid is time-consuming since resources may be spread across organizations or even continents, unlike in cluster computing. In Chapter 6, we propose the data-aware scheduling approach *DataVinci* that is heavily influenced by data and task placement approaches for grids such as [41, 42].

Volunteer computing is a paradigm where providers voluntarily offer their resources for scientific projects. Grid and volunteer computing share the concept that providers across organizations contribute to a powerful pool of resources that appears to the consumers as a transparent unit. A major difference between both paradigms is the motivation of providers to participate. In grid computing, providers expect to harvest computing power as consumers in the future, i.e., to profit from the grid themselves. In volunteer computing, providers “donate” their resources for a good cause. Thus, the group of consumers (typically the members of a scientific project) is clearly defined from the start. In addition, volunteer computing resources originate from private end-users while grid resources often belong to a certain organization, e.g., a university. Hence, as private end-users participate as providers, research on volunteer computing has a strong focus on security, fault-tolerance, usability, and incentive mechanisms. Today, the *BOINC* software [43, 44] is the de-facto standard for volunteer computing. It is an open-source middleware that is used by the most popular volunteer computing projects such as *SETI@home* [8–10] and *Folding@home* [11].

2.1.3. Cloud Computing

Cloud computing is the paradigm that fulfills the promise grid computing had made: a transparent access to scalable computing power. While grid computing is often limited to scientific use cases, cloud computing is a disruptive technology that has achieved high market penetration and transformed the way we use computers. Commercial cloud computing platforms such as *Amazon Web Services*[4] and *Microsoft Azure*[5] are highly popular. According to the National Institute of Standards and Technology (NIST), “*cloud computing is a model for* [...] *convenient, on-demand network access to a shared pool of configurable computing resources*

[4]`https://aws.amazon.com/`, accessed 2021-10-13
[5]`https://azure.microsoft.com/`, accessed 2021-10-13

[...] *that can be rapidly provisioned and released with minimal management effort* [...].” [45, p. 2]. Grid and cloud computing therefore share the same goal but achieve it in different ways. Foster *et al.* compare the two paradigms extensively in [12]. Armbrust *et al.* [14] observe three differences between grid and cloud. First, the cloud provides scalable, almost limitless computing resources on demand. Second, no prior commitment of the consumers (e.g., by submitting jobs or reserving providers) is required. Third, consumers pay for the resources in a “pay-as-you-go” fashion instead of participating in a resource sharing system as in grid computing. Grossman [13] adds that deploying applications in the cloud is easier for the users. From an architectural perspective, cloud computing is a rather centralized distributed computing paradigm. Instead of devices owned by end-users or university departments that typically serve as providers in a grid, mainly (commercial) data centers provide cloud resources [46]. Cloud providers offer their resources for economic benefits, not for profiting from the same system later as consumers.

Cloud computing approaches usually offer one or multiple of the following service models: Software as a Service (SaaS), Platform as a Service (PaaS), or Infrastructure as a Service (IaaS) [45]. In SaaS, applications run on the provider devices, i.e., on the cloud infrastructure. Hence, the users transparently access an application that is running in the cloud via a thin client interface. In PaaS, consumers have the flexibility to run their own custom applications on the provider devices. This, however, adds some complexity since users have to tailor the application to the respective cloud infrastructure, e.g., with regards to OS or hardware. In IaaS, consumers are allowed to run arbitrary software in the cloud. This includes, for instance, the installation of a particular OS, which is not possible in PaaS. We observe that — independent from the service model — the underlying hardware and network are entirely transparent to the user.

Mobile cloud computing [47–49] is a variation of cloud computing that allows mobile devices such as smartphones to access the scalable resources in the cloud. In this context, the characteristics of mobile devices lead to new research challenges. For instance, mobile devices may frequently change their location and are not connected to a constant power supply.

2.1.4. Edge Computing

Albeit being a huge success, cloud computing has several disadvantages. First, it typically requires the setup of a cloud data center consisting of dedicated devices. Second, it leads to a central collection of data, which is problematic for privacy-sensitive users. Third, it is comparably expensive for consumers. For instance, running an Amazon EC2 instance of type `a1.2xlarge` with 8 cores and 16 GiB RAM costs 0.23 dollars per hour[6]. Fourth, it causes considerable communication latencies between the consumer devices and the cloud infrastructure. Recent trends such as the Internet of Things (IoT) [50], Virtual Reality (VR), AR, and machine learning increased the number of applications that require fast processing of large amounts of data created at the consumers. Analyzing this data in the cloud does not meet the response time requirements of these applications due to the high latency [51]. The same applies for highly interactive applications or games that require fast responses [52]. The *edge computing* paradigm [15–17, 53–55] promises to overcome these hurdles by moving the computation from the "core" (e.g., from cloud data centers) to the "edge" (e.g., desktop PCs) of the network. Typical providers in edge computing environments are therefore PCs, laptops, smartphones, or edge servers — servers attached to cellular base stations — that are topologically closer to the consumer. In addition to low latencies, edge computing leads to a better utilization of existing hardware (e.g., unused office PCs) [56, 57]. Reference [15] claims that edge computing improves the users' trust thanks to the decentralized analysis of potentially sensitive data. Both consumers and providers may benefit from edge computing economically [57–59]. While consumers have lower costs, providers may contribute their computing power in return for a monetary compensation.

The precise definition of "edge computing" is the subject of an ongoing debate in the research community [55]. Similar to Garcia Lopez *et al.*, we interpret edge computing as the aforementioned shift of computing power from the core to the edge of the network. From our point of view, typical providers in edge environments encompass end-user devices such as smartphones or desktop PCs. This facet of edge computing is sometimes referred to as *(mobile) ad-hoc computing* [60–62]. A different form of edge computing is *mobile edge computing* [4, 63]. Here, only

[6]`https://aws.amazon.com/de/ec2/pricing/on-demand/`, accessed 2021-10-19

dedicated servers at cellular base stations act as providers. These servers are reachable for mobile devices with a single hop, which is especially attractive in 5G networks [64]. In contrast to some researchers, we do not limit our definition of edge computing to such architectures.

Edge computing is closely related to *fog computing* [65]. Fog computing has a strong focus on the integration of edge resources, the cloud, and the devices that are topologically located in between edge and cloud. This leads to fog computing systems having three layers. The first layer consists of edge devices that produce data. This data is communicated to the second layer — a set of local fog nodes that offer computing power and act as a bridge to the cloud. The third layer represents the cloud that is responsible for higher-level data analysis on a global (i.e., system-wide) scale. In our perception, edge computing focuses more on end-user devices while fog computing emphasizes the integration of the cloud.

2.2. Computation Offloading

The distributed computing paradigms introduced in the previous section are able to offer multiple services to consumers. For instance, cloud or grid infrastructures can store user data and therefore act as backup servers. In this thesis, however, we focus entirely on *computation offloading* in such systems, i.e., the transfer of computation to providers. Computation offloading has evolved considerably since its origin as *cyber foraging* [7] in 2001. Today, many applications in the domains of Artificial Intelligence (AI), AR, or computer vision require more computing power than available on the consumer device, especially on “weak” devices such as smartphones or wearables [2]. In addition to the reduction of response times, computation offloading may improve the energy consumption of (i) the consumer device and/or (ii) the whole distributed system. Especially in edge computing systems, the benefits of computation offloading in terms of response time and energy consumption are context-dependent. Offloading becomes more attractive if, for instance, a powerful provider device (e.g., a server) is currently idle or the bandwidth increases. This example shows that an effective computation offloading system must make sophisticated, context-aware decisions for every unit of computation that is potentially offloadable. In the following, we shed light on the different facets of this decision. We structure this section in the sense of Flores *et al.* [66] who state that the offloading decision encompasses *what*, *when*, *where*, and *how* to offload. Figure 2.1 shows an overview of all design considerations that are discussed in the remainder of this section.

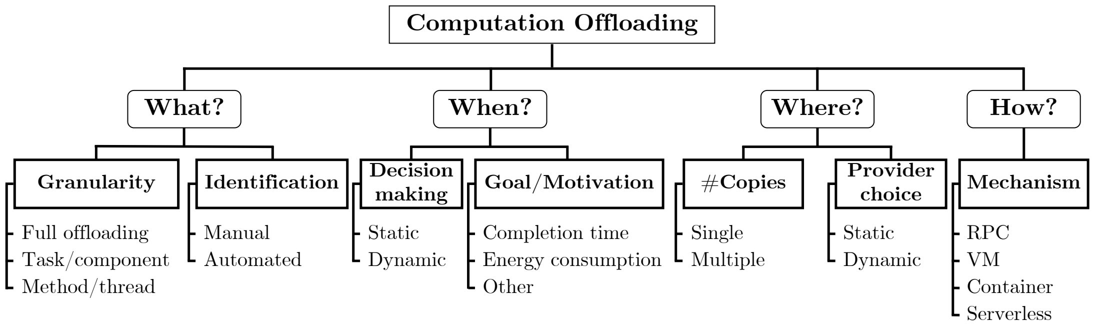


Figure 2.1.: Design considerations in computation offloading.
(RPC = Remote Procedure Call, VM = Virtual Machine)

What to offload? This dimension of the computation offloading decision — often referred to as *partitioning* — selects the parts of an application that will be offloaded. It determines the *granularity* of offloading. Lin *et al.* [5] mention three options: *full offloading*, *task/component*, and *method/thread*. In full offloading, the entire application is migrated to the provider. *XtremWeb* [67], for instance, offloads application binaries from the consumers to the providers. In [68], the authors present an architecture where mobile devices are augmented with a Virtual Machine (VM) in the cloud that acts as an offloading target for whole applications. Offloading on the level of tasks/components differentiates between parts of an application and decides for each part whether offloading is beneficial. This approach is more flexible but requires the careful partitioning of an application into several tasks or components. Offloading based on methods or threads (cf. [69, 70]) is the most fine-granular of the three partitioning options. As we mainly focus on task-based offloading in this thesis, we henceforth use the term “task” instead of the broader term “offloadable application (part)”.

Identifying the tasks that qualify for offloading is another challenge. Some tasks that are, e.g., responsible for the creation of the User Interface (UI), are per se not offloadable. The first option is to rely on manual identification by the application programmer, e.g., with annotations in the source code [71]. Application programmers should have a good intuition about which tasks are suitable for offloading. As an alternative, several approaches offer an automated identification [70, 72] based on code analysis or history traces [66]. While such approaches are convenient for programmers, they are potentially error-prone.

When to offload? The question whether a task that was identified in the partitioning process benefits from a remote execution is non-trivial. Two high-level options exist: *static* and *dynamic* decision making. Static approaches always choose either a local or a remote execution for a task, independent from the context. Strictly offloading a task, for instance, works sufficiently well if the task — independent from the parameters or the input data — is always exceptionally computation-heavy. Dynamic approaches decide whether a task should be offloaded based on the context. Even the same task may be sometimes executed locally and sometimes remotely. Dynamic decision making is more complicated but helps to unleash the full potential of offloading.

The design of an offloading system depends on the *goal*. In the literature, the most prominent goals are the minimization of task completion times (e.g., [70,73,74]) and the reduction of the energy consumption on the consumer device (e.g., [71, 75, 76]). Offloading reduces the task completion time if

$$\frac{data\_size}{network\_throughput} + \frac{task\_complexity}{remote\_throughput} < \frac{task\_complexity}{local\_throughput} \tag{2.1}$$

where the *data size* encompasses all data that must be transmitted before a remote execution (e.g., code and input data) and after (e.g., a result file). Offloading becomes more attractive if the *network throughput* increases. The *task complexity* is an abstract metric for the workload that has to be processed either locally or remotely. *Local throughput* and *remote throughput* determine the workload that the local device and the chosen remote provider, respectively, are able to process per time unit. How to quantify these metrics depends on the particular offloading system. As far as the energy consumption of the consumer device is concerned, offloading saves energy if

$$power_{transmit} * \frac{data\_size}{network\_throughput} < power_{compute} * \frac{task\_complexity}{local\_throughput} \tag{2.2}$$

where $power_{transmit}$ and $power_{compute}$ describe the power for transmitting the data and the power for computing the task locally. In addition to reducing the energy consumption of the consumer device, optimizing the cumulative energy consumption of a whole distributed computing system is also feasible with offloading. We omit a detailed discussion of this case here as the remainder of the thesis concentrates on the energy consumption of the consumer device only. Please note that the above equations are simplified as, e.g., task abortions may happen. We present more detailed models of the task completion times and the energy consumption for the Tasklet system in Section 7.2 and Section 8.2.2, respectively. Several other offloading goals aside from the reduction of task completion times and energy consumption are mentioned in the literature. They usually play a subordinate role. Examples are cost savings by selecting the cheapest provider [76] or higher precision [77], e.g., by running more simulations thanks to the available additional computing power.

Where to offload? The placement of a task is crucial in computation offloading systems. After deciding that a remote execution is beneficial, it is required to select a *single* or *multiple* providers for execution. The number of providers depends on the level of parallelism and the desired redundancy. If a task consists of multiple subtasks that can be computed in parallel, it is possible to offload these subtasks to different providers. The consumer eventually merges the results. A task that applies a filter to a video of multiple frames, for instance, can be divided into a large number of subtasks, which each apply the filter to one distinct frame of the video. This rather extreme example shows that the level of parallelism is often application-dependent. In addition, redundancy as used in, e.g. *Sparrow* [78], requires to choose several providers as execution targets. Especially in edge environments, where providers leave the system spontaneously, it can be beneficial to offload the same (sub-)task to multiple providers to ensure proper task completion times. We observe that the choice set of available providers also has a strong influence on the question when to offload that we discussed above.

The choice of a provider happens either *statically* or *dynamically.* In a purely cloud-based distributed computing system, the decision is static as all tasks are offloaded to the cloud. The same applies to systems where each provider serves fixed consumers such as [68]. Modern edge environments require dynamic decision making as they contain a variety of potential offloading targets including the cloud, edge servers, desktop PCs, laptops, or even smartphones. Such environments necessitate a context-aware choice of the provider [5].

How to offload? Remote execution on a provider should yield the same results as local execution. Technically, multiple alternatives for task execution are available. We briefly summarize four prominent technologies based on [5]. First, early approaches such as *Spectra* [77] or *Chroma* [79] apply Remote Procedure Calls (RPCs). The advantage of an RPC is — in addition to its simplicity — that the execution on the remote device is starting quickly once the server-side implementation of the code has been deployed. The RPC mechanism, however, requires two versions of the same code on consumer and provider, which is cumbersome in heterogeneous systems. Second, offloading systems may use virtualization with VMs. A VM abstracts from the underlying hardware and makes it possible to run the same code on different architectures. This is a key feature for edge computing environments. VM-based offloading is prominently used in the literature, e.g, in *Cuckoo* [69], *CloneCloud* [70], and *COMET* [73], which run Java code in either Java Virtual Machines (JVMs) or Android's (former) VM *Dalvik*. Third, container-based virtualization with software such as *Docker*[7] is a lightweight alternative to VMs. While each VM runs a guest OS on top of the provider device's OS, containers share components of the device's OS. In this way, containers are typically faster and more flexible than VMs. *Rattrap* [80] is an example for a cloud-based offloading system that uses containers. Fourth, *serverless computing* is a comparably recent trend in computation offloading. The idea is that small code snippets that are written in the form of stateless functions are executed remotely [81]. The concept has similarities with the PaaS service model in cloud computing (cf. Section 2.1.3). The strength of serverless computing — and the distinction to PaaS — is transparency. Application developers are able to focus entirely on the logic of the program, written in a language supported by the respective serverless computing platform. The execution of the code is handled transparently by the platform. In [82], Baresi *et al.* present an offloading system that applies serverless computing in mobile edge computing environments. Cicconetti *et al.* [83, 84] also use serverless computing but focus more on pervasive environments and the IoT. In addition, several commercial serverless computing platforms such as *AWS Lambda*[8] or *Microsoft Azure Functions*[9] are popular.

[7]`https://www.docker.com/`, accessed 2021-10-18
[8]`https://aws.amazon.com/lambda/`, accessed 2021-10-18
[9]`https://azure.microsoft.com/en-us/services/functions/`, accessed 2021-10-18

2.3. Context-Awareness

The previous section demonstrated that today's distributed computing environments require increasingly dynamic offloading decisions. Therefore, they are typically *context-aware*. Dey and Abowd define context as "*any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves*" [85]. Context-aware applications gather context information about the surrounding computing environment and are therefore able to react to changes [86]. This makes context-awareness a fundamental property of self-adaptive software [87], i.e., software that is able to adjust itself autonomously after context changes [88]. Computation offloading approaches ideally follow the principles of self-adaptive systems to make well-informed task scheduling decisions.

As far as context acquisition is concerned, we distinguish between context that (i) is static and therefore known a priori such as the data size of a task, (ii) can be measured such as the current bandwidth, and (iii) must be predicted such as the complexity of a task. A context-aware scheduling approach can consider a multitude of context dimensions such as provider performance [74, 89], provider reliability [90, 91], system load [26, 78], task complexity [70, 77], task deadlines [37, 92], data size [83, 93], data locality [41, 94], or bandwidth [38, 76]. This list is not exhaustive. Thus, the challenge is to identify relevant context dimensions, quantify them, and weight their importance.

3. Requirements Analysis

In this chapter, we present requirements for effective computation offloading, driven by the four research questions from Section 1.2. Section 3.1 describes an ideal scenario where consumers and providers share resources in an edge computing system. The goal of this thesis is to design a fast and energy-efficient computation offloading system that realizes this ideal scenario. In Section 3.2, we introduce three major challenges of the scenario that we address in this thesis. Section 3.3 and Section 3.4 describe functional requirements and non-functional requirements derived from the scenario and the challenges.

3.1. Scenario

Modern edge computing systems lead to a paradox. On the one hand, the cumulative computing power of such systems is ever-increasing and probably sufficient for all contemporary applications. On the other hand, users still face waiting times while running computation-intensive applications from domains such as machine learning, VR, or video processing. The applications typically do not exploit the computing power of the whole distributed system. Instead, they are restricted to local processing on their end-user device. As a solution, resource sharing with computation offloading can unleash the full potential of a distributed computing system. In a computation offloading system, consumer devices transfer computationally intensive tasks to provider devices, which return the results via the network. This process is handled by a middleware that runs on consumer and provider devices. We identify three essential stakeholder groups: application users, resource providers, and application programmers.

The *application users* run an application in a business or private context. They act as resource consumers in a computation offloading system for two reasons. First, they face waiting times while using the application. Harvesting computing power from remote resources can accelerate task execution and, hence, reduce such waiting times. Second, they run the application on mobile devices such as laptops or smartphones that lack a constant power supply. The application users' devices can offload computation to resource providers to reduce battery drain. Application users do not have to be experts in information technology. The offloading process is therefore embedded in the application and invisible for the application user. An application that leverages the computation offloading middleware has the same usability as a similar application that performs local processing only. It is transparent whether a task was executed locally or remotely. In addition, the middleware selects a suitable offloading target transparently in case of a remote execution. Application users use a multitude of device types, including desktop PCs, laptops, smartphones, and even wearables. All devices — independent from hardware, OS, and software — are therefore able to run the middleware and participate in the computation offloading system.

Resource providers control devices that offer computing power for the consumers. The provider's incentives to participate may be, e.g., (i) to contribute to research for a good cause, (ii) to help family, colleagues, and friends, (iii) to receive a (monetary) compensation, or (iv) to profit from the excess computing power of others in the future. Since providers share their resources voluntarily, computation offloading should not create any disadvantages for them. Thus, the middleware is secure and prohibits the execution of malicious tasks. The state of the provider device remains unchanged from task executions as consumer tasks run in a sandbox environment with restricted access to, e.g., input/output and storage. To ensure that the device remains usable by the owner while being a provider, processes launched by the provider device itself have priority over incoming consumer tasks. The middleware runs in the background and does not negatively influence the experience while using the device for other purposes. It executes consumer tasks only if there are currently excess capacities on the provider device. Similar to the application users, resource providers do not necessarily have to be computer experts. Installing the middleware and participating as a provider is therefore possible without prior knowledge. Providers run the middleware on a variety of device types. Desktop PCs, laptops, smartphones, wearables, edge servers, and cloud servers are among the potential offloading targets.

The *application programmers* write the application code that leverages the middleware for computation offloading. Their main motivation to apply computation offloading is to improve the user experience by reducing waiting times and/or battery drain. Therefore, the middleware's performance in terms of task completion times and energy consumption is crucial. Instead of manually implementing the offloading process for each application, application programmers rely on the middleware's features. With an easy-to-use API, they embed computation offloading into an application with little effort. Various applications from diverse domains potentially benefit from computation offloading since they either contain computationally intensive tasks or drain the consumer device's battery. The middleware and the corresponding API therefore provide a generic approach for integrating computation offloading, independent from the use case.

3.2. Challenges

Computation offloading — as described in the above scenario — is in general beneficial for all involved stakeholder groups. Nonetheless, we identify three challenges that must be overcome to realize the full potential of computation offloading in edge-based distributed computing systems. Designing approaches that overcome these challenges is the core contribution of this thesis. In this way, computation offloading unleashes its full potential in terms of task acceleration and energy efficiency. In the following, we derive three challenges C_1 to C_3. Challenges C_1 and C_2 relate to the reduction of task completion times. Challenge C_3 influences the energy efficiency of computation offloading.

Challenge C_1 — Large input data: Many contemporary applications are data-intensive. In the emerging domains AI and machine learning, it is common to perform computations on large data sets. The machine learning models applied in such use cases are often large as well. Processing and editing images and videos with software such as *Adobe Photoshop*[1] requires the transmission of the respective images or videos when applying computation offloading. In the context of the IoT, many applications analyze sensor data, which can also amount to considerable data sizes. As these exemplary applications require complex computations, application programmers can use computation offloading to improve the experience of the application users. However, materializing the benefits is a challenge as the input data must be present on the provider before

[1]`https://www.adobe.com/products/photoshop.html`, accessed 2021-11-09

starting the task execution. In comparison to, for instance, cluster nodes, which are connected with high-speed links, devices in edge computing systems face considerable latencies when transferring data. The data transfer time therefore contributes largely to the overall task completion time of a remote execution. As data-intensive applications would in general — ignoring the data transfer times — often benefit from computation offloading, it is an essential challenge for an effective computation offloading middleware to mitigate the negative influence of data transfers on task completion times.

Challenge C_2 — Sub-second deadlines: Traditionally, computation offloading is used for applications with long-running, computationally intensive tasks. The scientific volunteer computing projects *SETI@home* [8–10] and *Folding@home* [11] that rely on the *BOINC* middleware [43, 44] are typical examples. Today, however, many applications such as face detection and recognition [71, 95] or game AI [71, 95, 96] are interactive and user-facing. Application users generally accept delays of less than one second but consider longer waiting times as unresponsive [97–99]. Since the advent of mobile devices such as smartphones, users run such interactive applications on resource-constrained devices. Computation offloading helps application programmers to achieve the desired response times even on devices with low computational power. Although the tasks are comparably short, they benefit from computation offloading as (i) parallelization can accelerate the execution considerably, (ii) providers have a higher throughput than the consumer device, or (iii) the consumer device may be busy running other tasks.

Meeting sub-second deadlines with computation offloading in the edge introduces new challenges. First, communication latencies contribute notably to the task completion time. Even if the input data is small, sending a Transmission Control Protocol (TCP)-based message to a remote provider takes dozens to hundreds of milliseconds, which has a high influence if task deadlines are short. Second, task abortions, e.g., due to a provider leaving the system, lead to costly re-transmissions of the task from the consumer to another provider. Detecting the failure and re-transmitting the task amounts to a large percentage of the time before a sub-second deadline. Third, if a provider is unable to execute a task immediately, e.g., due to task queuing, deadlines are presumably violated. Queuing is especially harmful if the queue contains one or multiple long-running tasks that block the resource for several seconds or even minutes. We therefore conclude that sub-second tasks require particular attention when making task placement decisions.

Challenge C_3 — Energy consumption: Reducing the energy consumption of the consumer device is a major motivation for computation offloading [71, 75, 76]. The consumer device benefits from a remote execution if the energy consumption of sending task and input data as well as receiving the results is lower than the energy consumption of local processing (cf. Equation 2.2). Whether this is the case depends on the characteristics of each individual task. Complex tasks that require less input data should typically be executed on a provider device; less complex tasks with comparably large input data on the consumer device.

In edge computing systems, deciding whether a particular task should be executed locally or remotely is challenging as the right decision depends on the context. Several context dimensions related to (i) the task such as task complexity, input data size, and result data size, (ii) the consumer device such as energy efficiency or computational power, and (iii) the network connection such as bandwidth influence whether a task benefits from a remote execution (cf. Section 2.2). Some of the aforementioned context dimensions (input data size, computational power of the consumer) are known a priori. Others, however, need to be measured (bandwidth) or predicted (task complexity, result data size). Thus, even for tasks that originate from the same application, a situation-dependent choice of local or remote execution based on accurate measurements and predictions is required.

3.3. Functional Requirements

We formulate seven functional requirements. Requirements R_F1 to R_F4 are derived directly from the ideal scenario in Section 3.1. Requirements R_F5 to R_F7 address challenges C_1 to C_3 from Section 3.2.

R_F1 — Computation offloading: The fundamental requirement is that the system will perform computation offloading. This encompasses the execution of tasks on remote providers and the reception of the results via the network. Application programmers shall be able to offload computationally intensive tasks from their preferred programming language with low programming effort. Whether tasks are executed locally or remotely should be transparent for both application programmer and user. The system shall make task placement decisions that realize the benefits of computation offloading such as accelerated task execution or energy efficiency.

***R_F*2 — Task-specific requirements:** The computation offloading system shall allow application programmers to formulate specific requirements for each task. This is crucial due to the heterogeneous pool of applications that benefit from computation offloading. Exemplary requirements are reliability, low task completion time, or energy efficiency. Application programmers shall be able to select these requirements programmatically and, hence, context-dependent for each task separately. The system shall make fine-granular task placement decisions based on the respective requirements and the current context.

***R_F*3 — Heterogeneity support:** All types of devices including laptops, desktop PCs, smartphones, edge servers, and cloud servers will serve as consumer or provider devices. The computation offloading system therefore shall be executable on all of these devices, independent from hardware, OS, software, or network connection. In addition, heterogeneous applications written in different languages shall be able to offload tasks that are itself heterogeneous in terms of data intensity, task complexity, or completion time requirements. The computation offloading system should provide a uniform abstraction that overcomes all the aforementioned facets of heterogeneity in the edge.

***R_F*4 — Context-awareness:** The computation offloading system shall measure context dimensions about the tasks, the devices, and the network. It should use this knowledge for implementing a well-informed task placement that fulfills task-specific requirements under various environmental conditions.

***R_F*5 — Data placement:** The computation offloading system shall place input data proactively on providers to achieve low completion times when offloading data-intensive tasks. Proactive data placement ensures that task execution can start immediately as the time-consuming data transfer, which may otherwise dominate the overall task completion time, has happened in advance. Depending on the context, this will include the creation and distribution of multiple copies (i.e., replicas) of an input data file to make the concurrent execution of tasks that require the same input data possible. The system shall determine the number of replicas individually for each data file to properly manage the tradeoff between task completion time and data transfer overhead. In addition, it should select the provider that will store a new replica carefully based on the context. This requirement overcomes challenge C_1 and makes computation offloading beneficial even for data-intensive applications.

$\boldsymbol{R_F}$**6 — Decentralized scheduling:** The computation offloading system shall allow consumers to make independent, decentralized task placement decisions in certain cases. Decentralized scheduling eliminates the need for coordination (e.g., with a central resource manager or other peers) before offloading a task. Thus, it accelerates the offloading process. Especially for tasks with sub-second deadlines, decentralized scheduling is essential as the time overhead for coordination in edge computing systems — where nodes may face latencies of more than 100 ms — would amount to a large percentage of the remaining time before the deadline. The decentralized scheduling approach integrated into the computation offloading system shall place tasks on reliable, fast, and idle providers. It shall avoid situations where many consumers offload tasks to the same providers as this leads to either task abortions or excessive task queuing. This requirement overcomes challenge C_2 and makes computation offloading attractive for responsive applications.

$\boldsymbol{R_F}$**7 — Energy-awareness:** When the application programmer requests an energy-efficient execution of a task, the system shall decide whether a local or a remote execution consumes less energy and place the task accordingly. This encompasses the monitoring of relevant context dimensions and the accurate prediction of task complexity and result data size. The computation offloading system shall make this decision for each task individually based on the context. This requirement overcomes challenge C_3 and maximizes energy efficiency.

3.4. Non-Functional Requirements

We identify five non-functional requirements for the computation offloading system.

$\boldsymbol{R_{NF}}$**1 — Performance:** Multiple metrics can describe the performance of a computation offloading system. The two major benefits of computation offloading are shorter task completion times and lower energy consumption. The degree to which a certain system achieves these benefits is therefore the essential quality metric that we apply in this thesis. Alternative performance metrics would be, e.g., the number of successfully executed tasks, the communication overhead, or the memory footprint.

$\boldsymbol{R_{NF}}$**2 — Scalability:** Scalability assesses whether the computation offloading system works properly when increasing the system load. The load depends on multiple factors such as the number of participating devices, the number of tasks, and the network conditions. The simplest computation offloading system consists of one consumer and one provider device. In practice, however, systems may connect hundreds to thousands of devices that frequently exchange tasks. The system shall therefore cope with high loads and achieve satisfactory performance under real-world system sizes.

$\boldsymbol{R_{NF}}$**3 — Robustness:** Computation offloading in the edge is error-prone due to fluctuation, communication link failures, malicious behavior, and user control over devices. Although it is impossible to eliminate such errors entirely, the computation offloading system should continue working properly after an error. In addition, it shall recover from errors transparently, i.e., without noticeable interruptions for the application users.

$\boldsymbol{R_{NF}}$**4 — Usability:** We identified three stakeholder groups: application users, resource providers, and application programmers. For all groups, using the system shall be convenient and as self-explanatory as possible. Applications that leverage the computation offloading approach should be as usable as applications that rely on local processing only. Providers shall profit from an easy installation process. Task execution on the provider device shall not interfere with local processes nor decrease the user experience of the device owner in any way. The application programmer shall require only minimal training before programming applications that leverage computation offloading. The usability requirement defines the extent to which the system meets these goals.

$\boldsymbol{R_{NF}}$**5 — Extensibility:** Distributed computing is subject to constant change. While the computation offloading system may have a suitable design for current edge computing systems, it may be unusable in a future distributed computing environment. The system shall be extensible and therefore adjustable to change. This includes, for instance, the integration of new devices, OS, or network technologies. Moreover, adding new features such as additional task placement algorithms shall be easy. This necessitates a modular design of the system.

4. Related Work

The previous chapter discusses requirements for computation offloading in the edge. This chapter assesses related work in terms of these requirements and, hence, identifies the research gap. We focus on full-fledged computation offloading systems. Approaches that consider, for instance, only application partitioning or transfer of input data are thus out of this overview's scope. We further limit our discussion to general purpose approaches, i.e., computation offloading systems that are usable with a wide range of applications. Approaches that are designed for a particular use case like *Kahawai* [100] — a Graphics Processing Unit (GPU) offloading approach for game rendering — are therefore excluded. In total, we review 30 computation offloading approaches.

We structure this chapter along the seven functional requirements gathered in Section 3.3. Section 4.1 gives an overview of how related approaches implement computation offloading (requirement R_F1). It considers the granularity of offloading, the mechanism, and the provider types. Section 4.2 analyzes which task-specific requirements related approaches are able to enforce (R_F2). Section 4.3 describes whether the approaches overcome heterogeneity in terms of hardware, OS, programming language, and network access (R_F3). As effective computation offloading requires the acquisition and exploitation of context (R_F4), Section 4.4 reviews whether existing approaches are context-aware. Section 4.5 covers data placement (R_F5), decentralized scheduling (R_F6), and energy-awareness (R_F7), which are specific to edge computing environments. The main contributions of this thesis — *DataVinci*, *DecArt*, and *Voltaire* — address requirements R_F5, R_F6, and R_F7. In Chapters 6 to 8, in which we present *DataVinci*, *DecArt*, and *Voltaire*, we provide an additional in-depth review of related work for the respective approach. These overviews cover more than full-fledged computation offloading systems as they also discuss approaches from adjacent research areas such as replication or low-latency scheduling, which are out of the scope of this chapter. Table 4.1 summarizes our findings.

Author/*System*	**Year**	**Granul.**			**Mechan.**					**Provider**				**TSR**			**HS**				**CA**			**Edge**		
		Full offloading	**Task/component**	**Method/thread**	**RPC**	**Binaries**	**VM**	**Container**	**Serverless**	**Mobile device**	**Static device**	**Edge server**	**Cloud**	**Reliability**	**Completion time**	**Energy**	**Hardware**	**OS**	**Language**	**Network access**	**Device**	**Task**	**Network**	**Data placement**	**Decentralized scheduling**	**Energy-awareness**
HTCondor [6,34]	1987*	●	●	●			●				●				●		●	●	●	●	●	●		○		
XtremWeb [67]	2001	●				●				○	●						●	●	●		●					
Spectra [77]	2002		●		●						●				●	●					●	●	●	○		●
Chroma [79]	2003		●		●						●				●	●					●	●	●			○
Entropia [101,102]	2003*	●					●			○	●				●		●		●		●	●				
BOINC [43,44]	2004*		●			●	●	●		●	●				●		●	●	●	●	●	●				
OurGrid [103]	2006		●				●				●						●	●			●			○	●	
Cloudlets [2]	2009	●					●					●					●	●	●		●					
MAUI [71]	2010			●	●						●				●		●	●		●	●	●	●			●
Cuckoo [69]	2010		●				●			●	●	●	●		●	●	●			●	●		●			
CloneCloud [70]	2011			●			●			●	●	●	●				●			●						●
MACS [104]	2012		●				●				●	●	●		●	●	●	●		●	●	●	●			●
COMET [73]	2012			●			●			●	●	●	●				●	○	○	●						
COCA [105]	2012			●			●						●				●			●	○	○	○			
Serendipity [106]	2012		●				●			●							●				●	●	●			○
ThinkAir [76]	2012			●			●						●		●	●	●			●	●	●	●			●
Aneka [107,108]	2012		●	●				●			●	●	●		●		●	●			●	●				
Clone2Clone [68]	2013	●					●				●	●	●				●			●						
CMcloud [109]	2014			●			●						●		●		●			●	●	●				
Jade [110,111]	2014*		●				○			●					●		●			●	●	●	●			●
Sapphire [112]	2014		●		●					●	●	●	●				●	●	●	●	●	●	●	○		
COSMOS [113]	2014		●				●						●		●		●			●	●	●	●		○	○
FemtoClouds [114]	2015		●							●							●				●	●	●			
Avatar [115]	2015		●				●						●				●	○	○	●						
Rattrap [80]	2017		●					●					●				●			●						○
Nebula [74]	2017		●				●				●	●	●				●	●		●	●	●	●	●		
CloudAware [116]	2018		●				●			●	●	●	●				●	●		●	●		●			○
EMCO [75]	2018		●				●						●				●			●	●	●	●			●
Echo [117]	2019			●			●					●	●				●	●		●	●	●	●	○		
Cicconetti [83,84]	2019*			●					●	●	●	●					●	●	●	●	●	●	●	●	●	

Table 4.1.: Overview of related computation offloading approaches. For all approaches marked with *, papers published in later years were considered as well. The year in the table refers to the publication date of the respective first paper considered for this overview.
(Granul. = Granularity, Mechan. = Mechanism, TSR = Task-specific requirements, HS = Heterogeneity support, CA = Context-awareness)
● fulfilled ○ partially fulfilled

4.1. Computation Offloading

The approaches that we examine in this chapter all allow consumers to offload workload to providers. Thus, they all fulfill requirement R_F1. This section reviews how related approaches meet the requirement. It includes information about the *granularity* of offloading, the *mechanism*, and the *provider types*. In Table 4.1, the columns `Granul.`, `Mechan.`, and `Provider` cover this information.

4.1.1. Granularity

We categorize the literature into approaches that perform (i) *full offloading* — running a whole application on a provider —, (ii) offloading on the *task/component* level, and (iii) offloading on the *method/thread* level as described in Section 2.2.

Only five of the 30 reviewed approaches perform full offloading. We briefly present *Clone2Clone* [68] as a representative of these approaches. *Clone2Clone* is a platform that allows Android devices to run whole applications in the cloud. Each mobile device is associated with an individual "clone" — a VM in the cloud that also runs Android. This clone can execute arbitrary applications for the consumer. Whenever required by the application, the clone communicates updates (i.e., results) back to the consumer. As all clones are hosted in the cloud, they can easily communicate with each other, which can be a valuable alternative to direct and unreliable connections between the mobile devices. *Clone2Clone* requires applications to be aware of this special architecture. The application programmer is therefore responsible for designing the applications such that they are executable on the clone, communicate with other clones, and transfer results to the consumer.

Performing offloading on the task/component level is the most common design (18 out of 30 approaches). Either the application programmer or a special software — integrated into the offloading approach or developed separately — partitions the application into several tasks or components, which are offloaded separately. *Chroma* [79] is an example for a grid computing approach that applies offloading with this granularity. It offers a domain-specific language that allows application programmers to formulate so-called *tactics*. A tactic encompasses a meaningful series of sequential and/or parallel application components. Thus, each tactic describes a successful execution of the application. An application typically has multiple tactics. *Chroma* automatically chooses the tactic that maximizes the application programmer's goal such as fast execution at runtime depending on the context. For each component of the chosen tactic, it furthermore decides whether a local or a remote execution is more beneficial.

The most fine-granular offloading approaches work on the method/thread level (implemented by ten of 30 approaches). *CloneCloud* [70], for instance, is able to migrate a thread from an Android device to an arbitrary remote provider. Later, the thread may migrate back to the consumer and continue execution there. On the provider, *CloneCloud* creates a VM-based execution environment that is similar to the one on the consumer device. *CloneCloud* automatically scans the application's source code for points where a thread can potentially migrate to or from the provider. Thus, manual annotations are not required. To gather a sufficient data basis for decision making, *CloneCloud* runs the application several times on consumer and provider to measure completion times and energy consumption between two subsequent migration points. Based on this data, it optimizes future executions by deciding at each migration point whether or not the thread should migrate to or from the consumer. *CloneCloud*'s approach is closely related to *COMET* [73], which also uses thread-level migration in Android. The main unique characteristic of *COMET* is that it relies on distributed shared memory. *COMET* synchronizes the VM-based execution environments on the consumer and the provider such that a thread migration is possible at any time with low overhead. In contrast to *CloneCloud*, which concentrates on application partitioning, *COMET* focuses on designing an effective offloading mechanism with low communication overhead, i.e., less frequent and less heavy state synchronization.

4.1.2. Mechanism

A large majority of the reviewed approaches gives an implicit or explicit description of how computation is offloaded from consumer to provider. Only the *FemtoClouds* project [114] identifies the choice of the exact mechanism as an ongoing challenge in a recent publication [118]. We observe offloading based on (i) *RPCs*, (ii) *execution of binaries*, (iii) *VMs*, (iv) *containers*, and (v) *serverless computing*. This categorization is largely inspired by Lin *et al.* [5] (cf. Section 2.2).

Spectra [77], *Chroma* [79], *MAUI* [71], and *Sapphire* [112] rely on traditional RPCs. The advantage of an RPC is that it is both simple and fast once the provider-side implementation of the code has been deployed. Only the volunteer computing approaches *XtremWeb* [67] and *BOINC* [44] transfer binaries that are compiled for the specific target provider. In *XtremWeb* [67], for instance, providers execute such pre-compiled binaries. While this mechanism — similar to RPCs — minimizes the level of required virtualization, it necessitates the creation of several binaries of the same application. This introduces considerable overhead to the offloading process, especially in highly heterogeneous environments. In *XtremWeb*, the broker stores the binaries from all consumers centrally. To offload an application, it transfers the binary that matches the respective execution environment to the provider.

Offloading to VMs it the most common design, which 21 of 30 approaches follow. Using a VM-based design is attractive as it creates the same execution environment on both consumer and provider, which makes it possible to execute tasks on heterogeneous providers without large modifications to the application code. Being a commercial grid computing solution, *Entropia* [101] must include an execution environment on the provider that is secure and unobtrusive. The *Entropia* VM [102] is able to run arbitrary Windows applications without modifications. It provides a sandbox environment, which guarantees that the application is not able to modify the local disk or to make malicious system calls. In addition, it monitors the system load on the provider and suspends task execution if the device owner requires the computing power for other purposes. To ensure security for the consumer, *Entropia* stores input data encrypted on the provider. In *Avatar* [115], every mobile device user has access to a VM in the cloud. This design is similar to *Clone2Clone* [68] presented in the previous section. In *Avatar*, however, every VM is associated with one user — who may control several devices — instead of one particular device as in *Clone2Clone*. VM-based computation offloading would in practice considerably increase the workload that the cloud infrastructure has to process. *Avatar*'s authors explicitly discuss this disadvantage of heavyweight VMs in [115]. A cloud infrastructure that, for instance, provides one VM for each device has to utilize the available resources in the data center such as storage capacity efficiently to cope with the large number of VMs running in parallel.

Instead of optimizing the management of the VMs, Wu *et al.* propose to replace VMs with lightweight containers [80]. They develop the computation offloading system *Rattrap*, which relies on containers that provide a mobile OS environment on top of the cloud servers' OS. *Rattrap*'s prototypical container only encompasses the subset of the Android OS that is required for remote execution. This excludes, for instance, camera or sensor drivers. Additionally, the authors optimize, e.g., boot process and storage usage.

The approach by Cicconetti *et al.* [83, 84] is the only reviewed approach that uses serverless computing. It is able to offload stateless methods to edge devices or servers. The approach shows that it is occasionally difficult to assign the offloading mechanisms in the literature to distinct categories as the approach's serverless computing architecture builds upon RPCs and VMs/containers. The consumers execute an RPC on a broker, which selects the provider for task execution. The broker relays the RPC to the provider that executes the stateless method in a VM/container. While serverless computing approaches typically offer additional services such as scaling of containers or fault tolerance, they may internally rely on, e.g., RPCs or VMs as this example illustrates.

The approaches presented so far in this section have in common that they rely on one particular offloading mechanism. Of all 30 reviewed approaches, only *BOINC* [44] includes multiple alternatives. *BOINC* is a middleware for volunteer computing. Since its origins in 2002, it has become the de-facto standard for volunteer computing and one of the biggest success stories in distributed computing in general. Application programmers — mainly from the scientific domain — can publish projects that rely on the *BOINC* middleware. The providers install the *BOINC* client and select the projects that they prefer to donate computing power to. Whenever excess capacities on the provider device exist, the *BOINC* client requests workload from the central *BOINC* server. *BOINC* offers three offloading mechanisms. First, it is able to offload binaries to devices with the respective OS and hardware. Second, it can offload applications that run in *VirtualBox*[1] VMs. Third, it can handle container-based applications that use *Docker*. This versatility eases the application programmers' development process as they can choose their preferred offloading mechanism.

[1]`https://www.virtualbox.org/`, accessed 2021-11-26

4.1.3. Provider Types

Various device types can act as providers. We assess whether related approaches offload workload to providers from the following categories: (i) *mobile devices* such as smartphones or laptops, (ii) *static devices* such as desktop PCs or privately owned servers, (iii) *edge servers* in the consumer's proximity, e.g., attached to cellular base stations, and (iv) *cloud servers*.

Twelve of the 30 reviewed approaches consider offloading to mobile devices. The potential of hand-held mobile devices as providers was already recognized by the *XtremWeb* [67] project in 2001. The authors anticipated major future developments and planned to use mobile devices such as Personal Digital Assistants (PDAs) as providers. After the advent of powerful smartphones, multiple projects such as *Cuckoo* [69] and *CloneCloud* [70] have considered such devices as providers. *Serendipity* [106] is a computation offloading approach that explicitly considers only mobile devices as providers. The system is designed for situations where such devices move and, hence, frequently lose connection. *Serendipity*'s focus is thus to offer task placement strategies that enable effective computation offloading in such "hostile" environments. The prototypical implementation is able to offload Java application parts from and to Android smartphones. Similarly, *Jade* [110, 111] also performs computation offloading to mobile devices that run Android. This system is motivated by the observation that many people own multiple devices such as smartphones and tablets, which are often located in close proximity. *Jade* connects these devices in a peer-to-peer fashion via Wifi or Bluetooth. In contrast to *Serendipity*, *Jade* does not assume frequent connection losses as both consumer and provider are controlled by the same person.

Offloading to static devices (18 approaches) or the cloud (17 approaches) is even more prominent in the literature than offloading to surrounding mobile devices. When we analyze our findings in Table 4.1, we observe a shift from static devices to cloud servers over time as described in Section 2.1. In the early decades of computation offloading research, grid computing and, hence, offloading to static devices was the dominating paradigm. Popular approaches for the grid include *HTCondor* [6,34] and *Entropia* [101,102]. Over the years, cloud computing became well-established. *COCA* [105] is an approach that relies on cloud computing. The system offloads Java methods from Android devices to cloud servers. First, *COCA* identifies all pure functions — methods that only access input variables, output variables, and variables declared inside the method body — as offloadable methods. It then measures the completion times and the memory footprints of these methods on the mobile device. In addition, *COCA* estimates these metrics for a remote execution in the cloud based on an emulation with the values measured on the mobile device and the bandwidth given as parameters. In the next step, the application programmer uses the measurements and the estimates to manually select the methods that will be executed in the cloud in the future. Eventually, *COCA* creates two application versions for mobile device and cloud. At runtime, the *COCA* software in the cloud waits for offloading requests and runs the respective methods accordingly.

While offloading to the cloud promises unlimited access to computing power, it has two major disadvantages that are discussed in related work. First, using public cloud resources leads to monetary costs for the consumers. Second, cloud resources are topologically far away from the consumer devices, which induces considerable latencies. *CMcloud* [109] is a cloud-based computation offloading approach that addresses the first disadvantage. It pursues two goals. It minimizes the server costs of the cloud provider (i.e., it maximizes the utilization of the servers) while simultaneously ensuring the execution of offloaded tasks before an application-specific deadline. Initially, *CMcloud* places a task on the cloud server that (i) is able to meet the deadline and (ii) has the highest utilization. In addition, it continuously monitors the execution on the server and migrates the task to a less loaded server if timely execution before the deadline is at risk.

Aneka [107, 108] is another approach that reduces the monetary costs of using cloud resources. To achieve this, *Aneka* combines cloud computing with grid computing. One of the main contributions of *Aneka* is a container that runs on heterogeneous hardware with heterogeneous OS. This abstraction allows the approach to include both grid and cloud resources in one system. To reduce costs, *Aneka* places tasks on grid resources whenever possible. Only if a task's resource demand — defined by a task-specific deadline — exceeds the capabilities of the grid environment, *Aneka* offloads it to the public cloud. Thus, *Aneka* provides scalable computing power while minimizing the monetary costs, similar to *CMcloud*.

The main motivation of approaches that rely on edge servers (twelve out of 30 approaches) is to reduce latencies between consumers and providers — the second disadvantage of cloud computing mentioned above. Satyanarayanan *et al.* introduce *Cloudlets* [2]. Cloudlets are small data centers that are reachable for the consumers with one fast hop. The consumers connect to the nearest cloudlet similar to Wifi access points and offload applications to VMs on this cloudlet. As this idea was already published in 2009, cloudlets are a forerunner of the now popular edge computing paradigm. More recent publications [119, 120] integrate cloudlets into hybrid architectures that run some services in the more stable and powerful cloud environment and some services on cloudlets in the edge.

The *FemtoClouds* [114] approach builds upon the idea of cloudlets. Similar to *Serendipity* or *Jade*, a *FemtoCloud* consists of mobile devices that are located in proximity to each other. These devices perform resource sharing. The offloading process is orchestrated by the cloudlet that is closest to the consumer. The cloudlet is responsible for, e.g., making task placement decisions. Thus, the cloudlet only hosts the broker software instead of executing tasks itself. This approach paves the way for new business models. Owners of cafés or public transport operators can install a cloudlet infrastructure with less powerful and, hence, cheaper hardware. The devices in proximity mutually share computing resources, which leads to low-cost access to scalable computing power.

4.2. Task-Specific Requirements

We analyze whether existing approaches allow application programmers to select task-specific requirements with regards to (i) *reliability*, (ii) *task completion time*, and (iii) *energy consumption* (R_F2). In Table 4.1, column `TSR` shows our findings. We consider requirement R_F2 to be fulfilled by an approach if the application programmer can explicitly set a requirement, goal, or deadline for *each* task. Thus, computation offloading approaches that always optimize a metric such as energy consumption for all tasks do not fulfill this requirement. The *Echo* system [117] is a good representative for such approaches. *Echo* either performs local execution or offloads computation to edge or cloud servers. Reducing the task completion time is the only goal in the *Echo* system. Thus, it does not allow application programmers to specify task-specific requirements but treats all tasks equally.

We observe that no existing approach considers task-specific requirements with regards to reliability. The reliable execution of tasks itself is considered by some approaches. However, no approach allows the application programmer to choose whether the overhead of such a reliable execution is even necessary. For instance, Nebula [74] — a computation offloading system for the edge — *always* executes a re-transmission mechanism after task abortions to ensure a reliable execution.

Setting task-specific requirements related to completion time and energy consumption is offered by 13 and five of the 30 approaches, respectively. As far as task completion times are concerned, several approaches such as *MAUI* [71] allow application programmers to set a deadline. Only the providers that are expected to meet this deadline are considered as offloading targets. Among these options, *MAUI* chooses the most energy-efficient. Although *MAUI* is a seminal energy-aware computation offloading approach, it does not offer any options to quantify an energy consumption goal or to deactivate energy-aware scheduling for a particular task. In addition to deadlines, there are other task-specific requirements that are often only used in a single approach. The *COSMOS* [113] system, for instance, allows application programmers to set a risk value for their application. *COSMOS* [113] offloads workload from Android devices to the cloud. The consumer may lose connection to the cloud provider due to the mobility of Android devices such as smartphones. Thus, remote execution in the cloud is less predictable and, hence, riskier than local execution with regard to task completion times. The system adapts its task placement strategy to the respective risk value. If the application programmer accepts a high risk to realize the full potential of frequent offloading, *COSMOS* is more likely to execute a task remotely.

In contrast to *MAUI* and *COSMOS* — where application programmers can only specify requirements of a certain pre-defined type —, the *HTCondor* project [6, 34] includes a flexible approach to formulate custom task-specific requirements. Providers advertise their resources with so-called *Machine ClassAds*, which specify the characteristics of the device including, e.g., OS and storage capacity. The consumers analogously formulate the task requirements in a *Job ClassAd*. There are no restrictions on the content of a *ClassAd*, which is an advantage compared to having a list of distinct task-specific requirements that the computation offloading system offers. *HTCondor*'s matchmaker then compares the *Job ClassAd* and all available *Machine ClassAds* to find the best match. While this approach offers high flexibility as *ClassAds* can potentially contain arbitrary information, the approach also has its limitations. It focuses rather on the provider characteristics instead of the task execution's characteristics. Requesting an energy-efficient execution, for instance, is difficult to achieve with *ClassAds* as the energy consumption of a task does not solely depend on static characteristics of a provider but also on dynamic context of the network or even the task itself. Incorporating this into a *ClassAd* would be inconvenient for the application programmer and would require frequent updates of the advertisements to react to context changes.

Approaches that allow multiple task-specific requirements have to determine how these requirements are weighted and how conflicts can be resolved. *HTCondor*'s *Job ClassAds* can additionally contain a function to rank providers based on their characteristics, which can be interpreted as a custom utility function specifiable for each task. *Spectra* [77] and *Chroma* [79] follow a similar concept. *Spectra* considers the three metrics task completion time, energy consumption, and fidelity. Fidelity describes the quality of a task execution, e.g., the precision of a simulation. It is, therefore, an application-specific value. The application programmer can specify a utility function that weights the three metrics for an upcoming task. *Spectra* then selects the provider for execution that is expected to maximize utility. An almost identical approach is used in *Chroma* [79]. The fine-granular weighting of task-specific requirements is both a blessing and a curse. On the one hand, it offers a sophisticated approach for application programmers to express requirements. On the other hand, the effort may exceed the benefits as the application programmer must provide a function that describes the desirability of each fidelity value and a function for weighting the three objectives. This might be a cumbersome procedure in practice, especially since expressing preferences in percentage values may be unintuitive for many application programmers.

4.3. Heterogeneity Support

An effective computation offloading approach needs to overcome the inherent heterogeneity of today's distributed systems (R_F3). To meet requirement R_F3, an approach must overcome heterogeneity in terms of (i) *hardware*, (ii) *OS*, (iii) *programming language*, and (iv) *network access*. Column `HS` in Table 4.1 lists our findings. A large majority of the reviewed approaches (28 out of 30) considers hardware heterogeneity. Devices with different hardware are able to participate as consumers and/or providers in the same system thanks to, e.g., virtualization with VMs. Several approaches (15 out of 30) overcome OS heterogeneity but only a few (nine out of 30) are not restricted to a certain programming language for the application. A common design is computation offloading from Android devices to VMs in the cloud via Wifi or cellular. Exemplary approaches for this design are *COSMOS* [113], *COCA* [105], and *CMcloud* [109]. Such approaches are usable with varying underlying hardware and at least two types of network connections. Consumer devices, however, are required to run the Android OS. In addition, these approaches can typically only offload tasks from Java applications.

Overcoming heterogeneity is a major focus of *Sapphire* [112]. *Sapphire* includes an object-oriented programming model that allows application programmers to declare objects as *Sapphire objects*. The location of these objects is managed transparently by *Sapphire*. The objects may be invoked locally or at an arbitrary remote provider, which is invisible to the application. *Sapphire* encompasses a *deployment kernel*, which runs on all devices, including mobile devices and servers. The deployment kernel is responsible for overcoming heterogeneity. It offers lower-level services such as an RPC mechanism and *Sapphire* object location. On top of this deployment kernel, application programmers can select at most one *deployment manager* for each *Sapphire* object. Deployment managers exploit the low-level services provided by the deployment kernel to achieve the behavior desired by the application programmer. *Sapphire* includes a library of standard deployment managers for, e.g., computation offloading, replication, load balancing, or fault tolerance. In addition, application programmers can create custom deployment managers with the API offered by *Sapphire*'s deployment kernel. Thus, *Sapphire* is a distributed computing platform in a wider sense, with computation offloading being only one of several use cases.

Cuckoo [69] is a computation offloading approach that overcomes network access heterogeneity. With *Cuckoo*, Android smartphones can offload workload to heterogeneous providers such as laptops, desktop PCs, and cloud servers. As smartphones offer several communication channels, *Cuckoo* supports Wifi, cellular, and Bluetooth. *Cuckoo* exploits the *Ibis* communication middleware [121] for this purpose. Application programmers are able to work with *Cuckoo*'s API while *Ibis* manages the underlying network communication transparently. The matchmaking between consumers and providers works with Quick Response (QR) codes. Each provider that runs the *Cuckoo* software can display a unique QR code. The consumer smartphones scan the code, which establishes a consumer-provider connection between the devices.

With the *Jade* system [110, 111], offloading to devices in proximity is possible via direct Wifi or Bluetooth connections. *Jade* therefore also considers network access heterogeneity. The system includes a comparably simple mechanism to decide which connection type reduces the consumer's energy consumption. Before transfer, *Jade* puts all data in a buffer. *Jade* uses Wifi if the buffer size exceeds a threshold because in this case either the amount of data or the frequency with which data is added to the buffer is high. Wifi is more energy-efficient under these circumstances. Analogously, *Jade* uses Bluetooth if the buffer size is below a second threshold. In this case, Bluetooth consumes less energy as only small amounts of data have to be transferred at a low frequency. If the buffer size is between the two thresholds, *Jade* continues to use the current connection type.

4.4. Context-Awareness

Requirement R_F4 defines that computation offloading systems should be context-aware. Due to changing environmental conditions, they should monitor the context and behave accordingly, which is especially important for effective task placement decisions. As shown in column `CA` of Table 4.1, several approaches fulfill requirement R_F4 and consider context from all three categories that we cover in this chapter: (i) *device context* such as a provider's throughput, (ii) *task context* such as input data size, and (iii) *network context* such as bandwidth. The *MACS* middleware [104], for instance, uses context information from all three categories to make task placement decisions. It is able to minimize task completion times, energy consumption, or memory usage. To decide whether a local or a remote execution of a specific application part is more beneficial, *MACS* gathers information about the (i) consumer device (energy profile, available memory, and local throughput), (ii) provider device (remote throughput), (iii) task (memory consumption, code size, input data size, and result data size), and (iv) network (bandwidth). The number of instructions — a crucial influence on both task completion time and energy consumption — is assumed to be proportional to the code size. In Chapter 8, we demonstrate that this assumption is overly simplistic in many use cases, which may lead to inaccurate decision making.

ThinkAir [76] is another context-aware approach. It offers VM-based computation offloading from Android smartphones to the cloud. The application programmer can select multiple goals such as low task completion time, low energy consumption, or low monetary cost for the cloud resources. *ThinkAir* decides for each method invocation at runtime whether a remote execution is suitable. This approach differs from related work such as *COCA* [105] where the placement decision is made statically for each method at design time. Unlike most other approaches, *ThinkAir* considers a large variety of context dimensions to make the offloading decision. First, it monitors the Central Processing Unit (CPU) state, screen brightness, and network connection (Wifi vs. cellular) on the consumer device. Second, it stores data about past executions of a method including completion time, number of instructions, or garbage collector invocation count. Third, it profiles the network including uplink data rate and uplink channel rate for both Wifi and cellular. Especially network monitoring is comparably simple in *ThinkAir*'s system model as all consumers offload to the cloud and do not share resources with each other, which would necessitate monitoring the connection from each consumer to various instead of only one static provider.

The *CloudAware* [116] system applies context prediction for minimizing task completion times. *CloudAware* considers context information about the consumer device that is not used in any other of the reviewed approaches such as location, calendar events, call history, and application usage. Based on this data, *CloudAware* predicts, for instance, the task completion time on a provider, the available bandwidth, and the probability that a task will be executed successfully on a provider. *CloudAware*'s prediction approach is extensible and offers generic interfaces for both machine learning classification and regression.

4.5. Edge Support

Edge computing systems are the major focus of this thesis. Computation offloading in the edge has to fulfill novel requirements including (i) *data placement* for data-intensive applications (R_F5), (ii) *decentralized scheduling* for responsive sub-second tasks (R_F6), and (iii) *energy-awareness* for reducing the consumer device's energy consumption (R_F7). Column `Edge` in Table 4.1 summarizes whether the reviewed approaches fulfill requirements R_F5 to R_F7. As meeting these requirements is the major contribution of this thesis, we review further related work that goes beyond general purpose offloading systems when we introduce our approaches *DataVinci*, *DecArt*, and *Voltaire* in Chapters 6 to 8.

We observe that only *Nebula* [74] completely satisfies requirement R_F5. *Nebula* distinguishes explicitly between devices that provide computing power and devices that offer data storage. The authors develop several data placement strategies that, for instance, minimize the duration of data transfers from storage nodes to providers. *Nebula* additionally applies data replication for fault tolerance and performance reasons. The system's centralized task placement strategies reduce task completion times while considering the current location of the input data.

Requirement R_F6 demands that consumers can autonomously select the exact provider for each task. Several approaches such as *MAUI* [71] or *ThinkAir* [76] allow consumers to choose between local and remote execution. The exact choice of the provider is either fixed, e.g., in cloud-based approaches or made by a central instance. Only *OurGrid* [103] and the approach by Cicconetti *et al.* [84] realize decentralized scheduling in the sense of requirement R_F6. We exemplary discuss *OurGrid* here. *OurGrid* [103] is a former open source grid computing approach that relies on decentralized scheduling. Typically, providers participate in a grid if they (i) are part of the same organization as the consumer or (ii) profit financially. Incentivizing providers with the first option is only possible in small grids that do not span across multiple organizations. The second option — a monetary incentive system — requires the design of a fair, secure, and reliable accounting system, which is challenging. Motivated by the disadvantages of the two options, *OurGrid* introduces a third option. It includes an incentive mechanism that relies on resource sharing with a so-called *network of favors*, where each peer is both consumer and provider at the same time. Every peer keeps one individual favor score for each other peer in the system. Whenever a peer executes a task, the consumer peer increases its local favor score of the respective provider peer, i.e., the consumer now owes the provider a favor. If multiple consumers request to execute tasks on the same provider, the provider prioritizes consumers to which it owes favors. This decision making happens in a decentralized peer-to-peer system without a central resource manager. All favor scores are local values as they represent a particular one-to-one relationship between peers. While *OurGrid*'s approach to use favor scores as the central metric for decision making is simple and scalable, it potentially leads to non-optimal task completion times.

The final requirement R_F7 considers energy-awareness. Several projects such as *Serendipity* [106], *CloudAware* [116], and *Rattrap* [80] mention energy efficiency as a benefit of computation offloading and evaluate their approaches with respect to energy consumption. These approaches, however, do not include any mechanisms to improve energy efficiency explicitly. They rather discuss energy efficiency as a side effect of minimizing task completion times. Nonetheless, a few approaches (seven out of 30) satisfy requirement R_F7. We exemplarily describe *EMCO* [75] in the following. *EMCO* applies crowdsourcing to gather information about past executions of an application. Based on this data, which includes task completion time, energy consumption, and context of past executions, *EMCO* learns a model that describes the circumstances under which offloading is beneficial. *EMCO*'s prototypical implementation, for instance, trains decision trees which classify the current context into situations where a local execution should be chosen and situations where a remote execution is expected to perform better, e.g., to reduce the energy consumption. In contrast to energy-aware offloading approaches such as *MAUI* — where the offloading decision is mainly made by the consumer device — *EMCO* trains the decision model centrally with the crowdsourced data. *EMCO* transfers the trained model to the consumers, which solely need to apply it for classifying the context.

4.6. Discussion and Summary

The literature review provides three key insights. First, we observe an insufficient consideration of task-specific requirements. Fewer than half of the reviewed approaches allow application programmers to select task-specific requirements at all. Moreover, only five of those approaches offer a specification of requirements that address both task completion time and energy consumption. Second, merely a few approaches (four out of 30) overcome heterogeneity in terms of hardware, OS, application programming language, and network access. Especially the independence from a certain programming language is a challenge that typically remains unresolved. We address these two observations by using the Tasklet system — an effective VM-based computation offloading system. The Tasklet system allows application programmers to offload fine-granular tasks from multiple programming languages to heterogeneous providers. In addition, application programmers can specify QoC goals such as reliability, fast execution, or energy-awareness. The Tasklet system enforces these goals with various QoC mechanisms. We describe the design and prototypical implementation of the system in the next chapter.

As a third key insight, we observe that the three challenges of edge computing systems that we identify in Section 3.2 are barely addressed in the literature. No existing approach properly supports data-intensive applications, responsive applications with sub-second deadlines, and energy-aware applications in the edge. In this thesis, we close this research gap. We extend the Tasklet system with sophisticated QoC mechanisms that achieve fast and energy-efficient edge computing even for the aforementioned challenging applications. We contribute to the state of the art by introducing three QoC mechanisms — *DataVinci*, *DecArt*, and *Voltaire*. *DataVinci* performs proactive data placement for data-intensive applications in the edge. *DecArt* is a decentralized scheduling approach for responsive applications with sub-second deadlines. *Voltaire* makes context-aware task placement decisions that reduce the energy consumption of the consumer device. We integrate *DataVinci*, *DecArt*, and *Voltaire* into the Tasklet system and present them in Chapters 6 to 8.

5. The Tasklet System

The Tasklet system is the basic artifact of this thesis. Its design answers research question 1. The Tasklet system is a computation offloading approach for heterogeneous distributed computing environments. It integrates the three scheduling approaches *DataVinci*, *DecArt*, and *Voltaire*, which are the main contributions of this thesis and which answer research questions 2 to 4. While this chapter presents the basic design of the Tasklet system, the subsequent Chapters 6 to 8 introduce *DataVinci*, *DecArt*, and *Voltaire* in detail. First, we describe the design of the Tasklet system in Section 5.1. Second, we introduce the implementation that is used as the primary prototype for evaluation throughout this thesis in Section 5.2. This implementation encompasses a computation offloading system for real-world testbeds — the *Tasklet Core System* — and the *Tasklet Simulator*, which is a realistic representation of the Tasklet system for large-scale simulations. This chapter partly bases on [122].

5.1. Design

The vision of the Tasklet system is that computing power is available for everyone whenever needed. Thus, there are no limitations anymore for a participating device. Instead of being limited to the computing power of the built-in CPU, every device can harvest computing power from the surrounding distributed computing environment. The computing resources are offered by the providers in the Tasklet system on demand and in a scalable fashion. While this sounds similar to the goal of grid computing, the Tasklet system is able to achieve this goal in edge computing environments. The Tasklet system is flexible. It is not constrained to a specific type of application, which is the case for many grid computing systems. In addition, it imposes no restrictions on the participating devices. A Tasklet-based distributed computing environment can contain cloud servers, edge servers, end-user devices, and even wearables in one system as we demonstrate, e.g., in [123][1]. These devices can act as consumers, providers, or both. With Tasklets, a consumer performs computation offloading with the same effort as local execution. Edge computing environments lead to many obstacles on the path towards this vision. Such environments typically contain a multitude of devices with heterogeneous hardware, software, and network connection [18]. Since many of these devices are user-controlled, fluctuation (i.e., devices joining and leaving the system) is common. In this section, we show how the Tasklet system overcomes these hurdles. The main idea is to bridge the gap between the set of heterogeneous applications and the pool of heterogeneous resources with an abstraction: the Tasklet. Tasklets are self-contained units of computation that include everything required for either a local or a remote execution. Before presenting the idea of Tasklets in more detail, we now proceed with a description of the underlying system model.

5.1.1. System Model

The Tasklet system is a general purpose computation offloading system. It can be deployed in stable cloud computing environments, in unreliable edge computing environments, and even in hybrid combinations of cloud and edge [123]. This flexibility is reflected in the system model, which does not imply constraints on the participating devices and only minimal constraints on the applications.

Applications offload workload in form of Tasklets. We introduce the concept of a Tasklet in detail in Section 5.1.2. All participating devices in the distributed computing environment run the *Tasklet Middleware*. The Tasklet Middleware is a software that is responsible for creating Tasklets, offloading them, and receiving the results. In addition, it provides an abstraction from the local hardware. To create a homogeneous execution environment, the Tasklet system uses virtualization by VMs. The TVM runs on every device that offers computing resources, i.e., on every provider. A provider may run several TVMs in parallel to offer multiple concurrent Tasklet executions. Thanks to the TVM concept, the heterogeneity of the devices is irrelevant from the perspective of a scheduler. Each device that offers one or multiple TVMs is a potential offloading target. We provide an in-depth presentation of the Tasklet Middleware in Section 5.1.3.

[1]Reference [123] is joint work with D. Schäfer, J. Edinger, J. Eckrich, and C. Becker.

The Tasklet system consists of *consumers*, *providers*, and *brokers*. Consumers run applications that use Tasklets for computation offloading. Providers offer their computing power in form of TVMs. The brokers perform resource management and Tasklet scheduling. One device can fulfill multiple roles. For instance, a common combination is that one device acts as a consumer and a provider at the same time. While parts of the workload of an application are offloaded to remote providers, the device's local TVMs also compute Tasklets. Figure 5.1 illustrates the three entities.

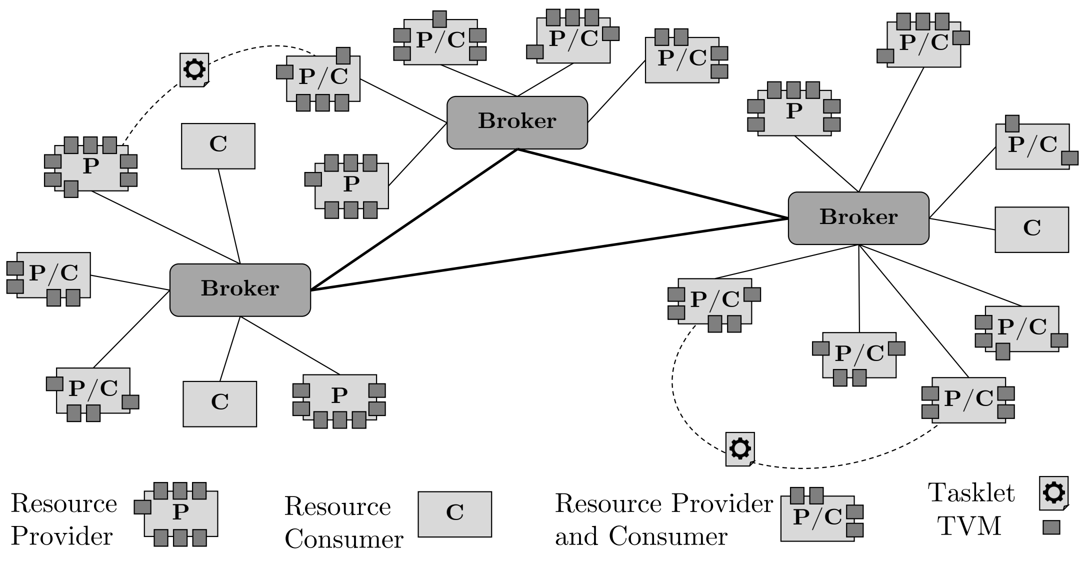


Figure 5.1.: The Tasklet system consists of consumers (C), providers (P), and brokers. Consumers offload workload in form of Tasklets to providers. Providers offer one or multiple TVMs for Tasklet execution. Brokers perform resource management and matchmaking between consumers and providers. Figure taken from [124].

Before offloading a Tasklet, a consumer sends an execution request to a broker. The broker performs the task scheduling, selects a provider for execution, and communicates this task placement decision to the consumer. The consumer sends the Tasklet directly to the chosen provider in a peer-to-peer fashion. After the execution on the TVM, the provider sends the results of the Tasklet back to the consumer, which concludes the offloading process. As, for instance, task abortions may occur, the lifecycle of a Tasklet can be more complex in practice.

The Tasklet system is a hybrid peer-to-peer system. While offloading the Tasklet and receiving the results happen directly between consumer and provider, the brokers act as central nodes with special responsibilities for matchmaking. The central role of the broker reduces the communication overhead in comparison to a purely decentralized peer-to-peer system. In large systems, several brokers can co-exist. They are each responsible for a subset of the participating devices. Brokers can share information about the providers and, hence, perform load balancing. There are — in theory — no restrictions on the devices that act as brokers. Ideally, they should be reliable and capable of handling the overhead of running the broker software. Therefore, it is possible to launch new brokers on demand. This procedure as well as systems with multiple brokers in general are, however, out of the scope of this thesis.

The Tasklet system works in a best-effort fashion. Thus, there are no guarantees for the execution of Tasklets. Tasklets may be aborted at any time, e.g., due to a provider leaving the system. In addition, task placement happens randomly. This simple model may satisfy the requirements of some applications. For all applications that require additional guarantees, we introduce the concept of QoC. Application programmers can specify QoC goals for particular Tasklets. These goals are realized transparently by the Tasklet Middleware. For instance, application programmers are able to request an energy-aware execution. As the programmers know that their application is mostly used on smartphones with a mobile data connection, they want to minimize the energy consumption of the consumer. The Tasklet Middleware incorporates this information into the execution request and the broker performs task placement accordingly. We introduce the concept of QoC in more detail in Section 5.1.4. The design of QoC mechanisms for challenging applications in edge computing environments is the main contribution of this thesis.

5.1.2. Tasklets

A Tasklet is a fine-grained and self-contained unit of computation. The core idea of the Tasklet system is that all devices are able to create Tasklets and to execute them, independent from the device type, OS, or network connection. A Tasklet is an encapsulated unit that contains everything required for execution. First, it includes the source code, compiled as bytecode that is executable on TVMs. Second, it includes (optional) parameters of the source code. For instance, if a

Tasklet calculates whether a specific number is prime, this number is passed as a parameter. Third, a Tasklet includes the input data. While the prime number calculation would not require input data, many Tasklets, e.g., for image processing or machine learning contain input data such as images, videos, or text. Fourth, a Tasklet contains metadata such as a unique identifier and the QoC goals that were chosen by the application programmer. Thanks to the design of a Tasklet, any provider is able to execute arbitrary Tasklets without any preparation.

Tasklets are independent. Different Tasklets do not interact with each other, neither by sharing memory, nor by passing messages. Therefore, the execution of a Tasklet always yields the same result even when other Tasklets are executed in parallel on the same device. This design choice allows the broker to make task placement decisions without dependencies between Tasklets. Nonetheless, many applications implicitly lead to a specific workflow. For instance, tasks should be executed in parallel or after each other with the input of the second task depending on the result of the first task. Such workflows are easily realizable with Tasklets. Parallel tasks can be offloaded with several Tasklets that are launched simultaneously. A sequential execution is achieved by launching a new Tasklet after the result of the first Tasklet has arrived at the consumer.

A Tasklet can contain a small code snippet, a method, or multiple methods that call each other. In Section 2.2, we categorize partitioning alternatives from the literature into *full offloading*, *task/component*, or *method/thread* [5]. Since Tasklets always represent a part of an application and not a whole application, the Tasklet system is usable for offloading on the level of task/component and of method/thread. The execution time of a Tasklet may therefore range from milliseconds to several hours. Tasklets are embedded into an application. Application programmers are able to write this so-called host application in their preferred language, which acts as the host language. With an easy-to-use API, programmers can launch Tasklets from the host application. The offloading process and the reception of the results is entirely transparent. Handling the offloading process and, hence, achieving this transparency is the responsibility of the Tasklet Middleware, which we describe in the next section.

5.1.3. Tasklet Middleware

The Tasklet Middleware is the software that runs on consumer and provider devices. It handles the offloading process. The Tasklet Middleware consists of three layers: *construction layer*, *distribution layer*, and *execution layer*. The construction layer of the consumer is responsible for the communication with the application that offloads Tasklets. It creates Tasklets upon request and forwards them to the distribution layer. The distribution layer handles the offloading itself. It communicates with the broker and the distribution layer of the provider. The execution layer is able to execute Tasklets on TVMs. The results are forwarded to the application via the distribution and the construction layer. Figure 5.2 depicts the layered architecture of the Tasklet Middleware. We observe that a pure consumer device without local TVMs does not require an execution layer. Analogously, a device that solely acts as a provider does not run the construction layer. If a device acts as a consumer and a provider, it hosts all three layers.

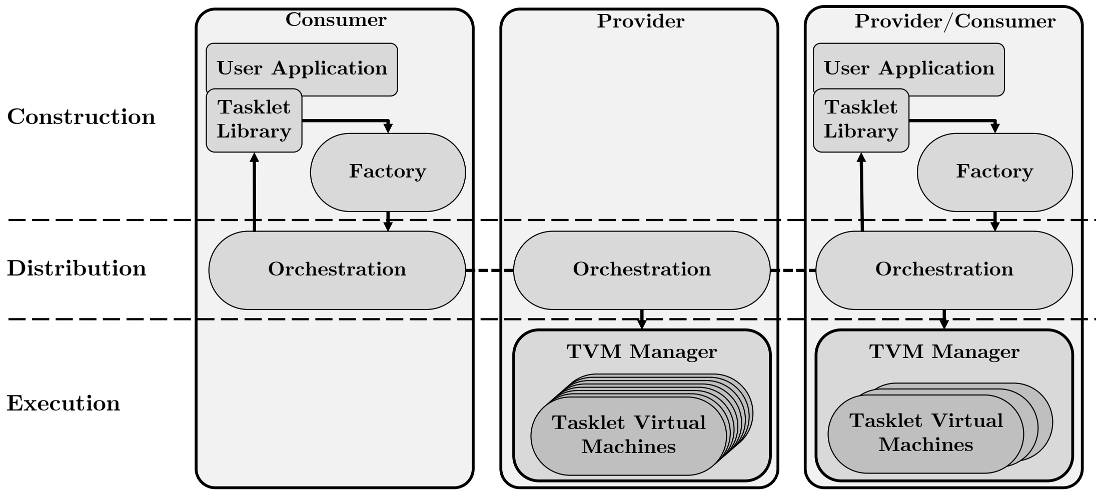


Figure 5.2.: The Tasklet system consists of three layers. The execution layer communicates with the application that offloads Tasklets. It creates the Tasklet and forwards it to the distribution layer, which handles the communication with other devices in the system. This layer orchestrates the offloading process. The execution layer is responsible for executing Tasklets on TVMs. A pure consumer device does not require an execution layer. A pure provider device does not use the construction layer.

Construction Layer

The construction layer is only present on a consumer device. It is responsible for the communication with the user application and, hence, the creation of Tasklets. The application itself can be written in the programmer's favorite language. Components such as the UI and the communication with the user via mouse or keyboard, as well as other types of input/output always have to run on the local device. These parts do not qualify for offloading. It is reasonable that programmers use their favorite programming language for such application parts. Only the computation-intensive parts, which potentially benefit from offloading, have to be written in the Tasklet language C--, which was explicitly designed for this purpose. This separation allows for an easy integration of Tasklets into legacy systems. Large parts of the existing application code remain untouched.

The Tasklet language C-- is a procedural programming language that is designed for computation offloading. It has a comparably small set of features including primitive data types, loops, conditions, and methods. The language allows programmers to add parameters to a Tasklet code and to specify an arbitrary number of results that will be transferred back to the consumer.

The programmer starts Tasklets from the host language with the help of the *Tasklet Library* — an easy-to-use API. The library connects the application code to the Tasklet code. It offers developers convenient methods in their favorite language to create Tasklets and to add parameters, QoC goals, and input data. The Tasklet Library transforms this information into a language-independent representation that is handed over to the *Tasklet Factory*. Moreover, the Tasklet Library offers methods to launch Tasklets and to receive results. In theory, application programmers could also create the language-independent representation of a Tasklet without the help of the library. While this makes the Tasklet system usable with all programming languages, it is rather cumbersome to do in practice.

The Tasklet Factory is the part of the Tasklet Middleware that receives the information from the Tasklet Library. It compiles the source code into bytecode. Together with the input data and metadata such as the QoC goals, the bytecode is transformed into the final representation of a Tasklet. The Tasklet Factory forwards this representation to the distribution layer.

Distribution Layer

The *Orchestration* component in the distribution layer is responsible for the placement of the Tasklet in the system. First, it decides whether a remote execution is possible for a Tasklet. Certain QoC goals, for instance, prevent a remote execution from the beginning. The Orchestration forwards the Tasklet to the local execution layer if a remote execution is not an option. Otherwise, the Orchestration requests resources at the broker.

The broker is the central authority for resource management and task placement. All providers register at the broker while joining the system. They announce the number of TVMs that they offer and their computing power based on a benchmark computation. Providers periodically send heartbeat messages to the broker. If the broker does not receive heartbeats from a provider for a specific time interval, it considers this provider to be unavailable. Whenever the broker allocates a Tasklet, it decrements the provider's number of idle TVMs. After a provider finished a Tasklet execution, it informs the broker that an additional idle TVM is available now. Thus, the broker has an up-to-date view of the current system. It is able to make well-informed task placement decisions based on this view. The task placement depends on the QoC goals of the Tasklet. Without any QoC goals, a Tasklet is randomly scheduled to a provider.

After making the task placement decision, the broker communicates the provider choice to the consumer's distribution layer. Subsequently, the Orchestration offloads the Tasklet to the chosen provider in a peer-to-peer fashion. The provider's distribution layer receives the Tasklet and forwards it to the execution layer.

Execution Layer

The execution layer contains the *TVM Manager* and one or multiple TVMs. The TVM Manager administrates the TVMs of one device. Per default, it launches one TVM per CPU core. The TVM Manager receives Tasklets from the distribution layer and schedules them on idle TVMs. The TVM is able to interpret the bytecode that was created by the Tasklet Factory. It is a lightweight VM with a small memory footprint. Therefore, the TVM is executable on many device types, which is beneficial in today's computing landscape. A TVM always executes

Tasklets sequentially without preemption. The Tasklet language C-- does not offer system calls, which ensures that the provider's state remains untouched during Tasklet execution. TVMs can be started and stopped spontaneously by the user of the provider device, e.g., if other applications require more computing power. After Tasklet execution, the results are forwarded to the distribution layer, which transfers them to the consumer. This concludes the lifecycle of a Tasklet.

5.1.4. Quality of Computation

The Tasklet system works in a best-effort fashion without any execution guarantees. Tasklets, however, may be aborted if a provider leaves the system or may be scheduled on slow providers. Such a behavior is not acceptable for many applications. In such cases, the Tasklet system therefore allows the application programmer to tailor the Tasklet execution to the application requirements by setting QoC goals. Exemplary QoC goals are reliable execution, fast execution, or energy-efficient execution. A single Tasklet can have multiple QoC goals. With the Tasklet Library, the application developer can easily add or remove QoC goals programmatically. Thus, the Tasklets launched by one application can have different QoC goals.

QoC Design

Thanks to the QoC concept, the Tasklet system is able to cope with the highly heterogeneous pool of user applications. These heterogeneous applications, however, are applied in a variety of distributed computing environments. The same application, for instance, may be deployed in a stable cloud environment or in an unreliable edge computing environment that only consists of mobile devices. Enforcing a specific QoC goal such as a reliable execution in these two exemplary computing environments is a challenge. Whereas the best-effort approach of the Tasklet system may already lead to a reliable execution in the cloud environment, achieving reliability in the edge environment may require additional measures such as a redundant offloading of two copies of a Tasklet code. At design time, the application programmer is not aware of the exact environments in which the application will potentially run. The Tasklet approach therefore separates QoC goals and QoC mechanisms. The programmer sets the goals. The Tasklet

Middleware observes the current environment and applies the suitable mechanism to achieve the QoC goal under the particular circumstances. This decoupling of QoC goals and QoC mechanisms additionally leads to extensibility. It is possible to implement new mechanisms in the Tasklet Middleware or the broker software for existing QoC goals. This happens transparently for the application programmer. Even legacy Tasklets would be able to profit from the new mechanism, without any changes in the user application or Tasklet code. Analogously, new QoC goals could be enforced by existing mechanisms or a new combination of QoC goal and mechanism could be added with low effort. In the following, we summarize the essential QoC goals and QoC mechanisms in the Tasklet system. A more detailed perspective on the QoC concept is given in [125].

QoC Goals

Extensibility is a major strength of the QoC concept. Thus, the list of QoC goals will grow further in the future. In this section, we present four QoC goals that are essential for this thesis. The Tasklet Middleware applies various mechanisms to achieve the QoC goals depending on the context. We introduce the mechanisms in the next section.

Reliability: Especially in unreliable edge computing environments, offloading a Tasklet, executing it, or transmitting the results may fail. With this QoC goal, developers request a guaranteed execution of a Tasklet. This also encompasses the successful transmission of the results to the consumer.

Speed: Setting the QoC goal *Speed* requests a low Tasklet completion time. As the completion time largely depends on the distributed computing system, i.e., the resources that are available, the Tasklet Middleware does not provide any guarantees to meet a deadline. Instead, it schedules the Tasklet to achieve the lowest possible completion time in the current environment.

Sub-second deadline: Some user-facing applications such as face detection, game AI, or speech recognition require sub-second response times for a satisfactory user experience. Thanks to parallelization and the choice of powerful providers, offloading is beneficial in these cases although the communication latencies may be as high as the task execution times. The QoC goals *Speed* and *Sub-second deadline* are similar since they both lead to low completion times. Sub-second response times, however, require different QoC mechanisms (cf. Chapter 7). As the complexity of a task is difficult to predict in advance, we therefore offer a separate QoC goal *Sub-second deadline* to be able to select the proper QoC mechanisms for responsive applications.

Energy: Reducing the energy consumption of the consumer device is one of the main motivations of computation offloading. The QoC goal *Energy* ensures that the Tasklet Middleware chooses either local or remote execution, depending on the estimated energy consumption of both options.

QoC Mechanisms

The Tasklet Middleware offers various QoC mechanisms to achieve the QoC goals. In the following, we present a selection of these mechanism that is relevant for implementing the four aforementioned QoC goals.

Redundancy: The Tasklet Middleware creates multiple copies of the same Tasklet. These copies are identical in source code, input data, and parameters. The broker schedules the copies independently, e.g., on different providers. With every copy, the probability that at least one Tasklet completes successfully increases. This mechanism therefore helps to achieve the *Reliability* QoC goal. In addition, it contributes towards low task completion times. After receiving the results of the first Tasklet, the consumer can continue with the application logic and discard the results of the other copies.

Retransmission: If a Tasklet fails, e.g, due to a provider leaving the system, the Tasklet Middleware detects the failure and starts a new copy of the same Tasklet. Thus, this mechanism is important for the reliable execution of Tasklets. To enable retransmission, the provider establishes a heartbeat channel to the consumer after receiving a Tasklet. The consumer is able to detect a task abortion when it does not receive any heartbeats for a specific period of time.

Fault-avoidance: The previous mechanism is able to cope with task abortions. Ideally, such cases do not occur in the first place. In edge computing systems, it is unfeasible to determine exactly whether a provider will leave the system soon or whether a communication link will fail. Nonetheless, the Tasklet Middleware is able to predict whether a provider will remain reliable based on its past behavior.

With this prediction, the Tasklet Middleware can allocate Tasklets to the most reliable providers. This mechanism for fault-avoidance is published in [90].

Migration: In the worst case, the provider that is currently executing a Tasklet leaves the system just before finishing the computation. All the progress is lost and the Tasklet might be restarted by the retransmission mechanism. We introduce Tasklet migration in [126][2] to preserve as much computing progress as possible in case of task abortions. This includes *reactive migration* and *proactive migration*. Reactive migration creates a snapshot of the TVM state before a provider performs an explicit leave. This snapshot is sent to the consumer, which migrates the state to another provider. Thus, reactive migration prevents the loss of Tasklet progress at the cost of communication delays during the migration process. To cover cases where an explicit leave is not possible, e.g, when users turn off their devices, we introduce proactive migration. Proactive migration creates snapshots of the TVM state periodically and sends them to the consumer. In this way, only a small part of the progress is lost.

Speed filter: The throughput of the provider is an essential influence on Tasklet completion times. This mechanism defines a threshold value for the processing speed. Only providers that have a higher throughput than the threshold are considered for the task placement decision.

Workload partitioning: Many applications launch multiple Tasklets that belong to the same job concurrently. A photo filter application, for instance, may launch four Tasklets that apply the same filter to a quarter of the image each. Such a parallelization reduces the job completion time — the time to apply the filter to the whole image — by up to 75 %. In edge computing environments, the throughput of providers differs considerably. Static devices such as edge servers or desktop PCs are expected to be faster than, e.g., smartphones. With workload partitioning, the Tasklet Middleware autonomously adapts the workload of a provider to its throughput such that all Tasklets of one job return their results approximately at the same time. We present workload partitioning with Tasklets in more detail in [126].

Data and task placement with *DataVinci*: Data-intensive applications are challenging for computation offloading systems as input data has to be transferred

[2]Reference [126] is joint work with D. Schäfer, J. Edinger, and C. Becker.

to the providers. This prolongs task completion times. *DataVinci* is a scheduling approach that proactively distributes input data on providers. The Tasklet execution can then happen without delays for ad-hoc data transfers. *DataVinci* — presented in Chapter 6 — therefore constitutes a valuable QoC mechanism for data-intensive applications.

Decentralized scheduling with *DecArt*: Similar to *DataVinci*, *DecArt* is a sophisticated QoC mechanism for particular use cases. If the Tasklet Middleware activates *DecArt* for a Tasklet, it uses decentralized scheduling without contacting the broker for task placement. This is particularly attractive for user-facing, responsive applications. Thus, *DecArt* is a QoC mechanism that achieves the *Sub-second deadline* QoC goal. We introduce *DecArt* in Chapter 7.

Energy-aware scheduling with *Voltaire*: *Voltaire* is a QoC mechanism that runs on the broker. Based on machine learning, *Voltaire* predicts the complexity of an upcoming Tasklet. *Voltaire* is able to approximate the energy consumption of a local or a remote execution with the expected Tasklet complexity and device-specific energy profiles. *Voltaire* therefore implements the *Energy* QoC goal. We describe *Voltaire* in detail in Chapter 8.

5.2. Implementation

We use a prototypical implementation — the *Tasklet Core System* — for the experiments in this thesis. In Section 5.2.1, we introduce the main features of this prototype. Section 5.2.2 describes the implementation of the Tasklet Library that allows programmers to launch Tasklets from a host language. For the experiments, we create several exemplary applications that apply computation offloading with Tasklets. We give a brief overview of these use cases in Section 5.2.3. In addition to the Tasklet Core System, we implement a simulator that is an accurate representation of the Tasklet system. We present this helpful addition for large-scale experiments in Section 5.2.4.

5.2.1. Tasklet Core System

The Tasklet Core System is a prototypical implementation of the design in Section 5.1. It consists of the Tasklet Middleware and the broker software, both implemented in C. The prototype of the Tasklet Middleware includes all three layers of the architecture in Figure 5.2. This encompasses an implementation of the TVM that is able to interpret Tasklet bytecode and, hence, to execute Tasklets. Interested readers may refer to [127] for more information about the implementation. The Tasklet Core System is a fully-fledged and operational computation offloading system. In a recent experiment, whose results are out of the scope of this thesis, we deployed the Tasklet Core System on more than 100 devices including cloud servers, desktop PCs, laptops, and smartphones. The Tasklet Core System is executable on Windows, macOS, Linux, Android, and iOS.

Application programmers implement Tasklet code with the programming language C-- in files with the ending `.cmm`. We describe the application development process with Tasklets in [128][3] and [129][4]. Listing 5.1 shows an exemplary C-- code. The Tasklet returns all prime numbers from `low` to `high`, with `low` and `high` being input parameters. Line 1 contains the declaration of global variables. Lines 3 to 11 encompass a method that checks whether variable `a` is a prime number. After all methods (or one method in this case), the main part of the Tasklet starts. Thus, line 13 is the entry point where the Tasklet execution begins. The `>>` operator connects the C-- code to the parameters that are added with the Tasklet Library from the host language. Here, lines 13 and 14 read the parameters `low` and `high`. The `<<` operator analogously determines the variables or values that are returned to the host application as a result. In the example, all prime numbers are declared as results of the Tasklet in line 19.

5.2.2. Tasklet Library

The Tasklet Library provides the link between the user application written in the host language and the Tasklet code written in C--. Currently, implementations of the Tasklet Library in Java, C#, and Dart exist. Listing 5.2 is an exemplary Java code snippet that uses the Tasklet Library to launch a Tasklet and to receive the results. Line 1 creates a new Tasklet with the C-- code stored in `primes.cmm`. Lines 2 to 4 add two integer parameters to the Tasklet. These have to match the variables that follow the `>>` operator in the `.cmm` file. Lines 5 and 6 add

[3]Reference [128] is joint work with D. Schäfer, J. Edinger, and C. Becker.
[4]Reference [129] is joint work with J. Edinger, D. Schäfer, and C. Becker.

the *Reliability* QoC goal to this Tasklet, which will be enforced by the Tasklet Middleware. Line 7 launches the Tasklet. Line 9 is a blocking call that waits for the Tasklet results. We observe that the offloading process is completely transparent for the application programmer. It is, for instance, not visible whether a local TVM or a remote provider executes the Tasklet or whether retransmissions occur. Line 10 converts the `TaskletResult` object into a Java `ArrayList`. Now, the host application continues with the application logic and processes the results.

```
int low,high,result;

procedure int checkprime (int a){
   int c;
   c:=2;
   while(c<=(a-1)){
      if((a%c)=0){return 0;}
      c:=c+1;
   }
   if(c=a){return a;}
}

>>low;
>>high;


while(low<high){
   result:=checkprime(low);
   if(result # 0){<<result;}
   low:=low+1;
}
```

Listing 5.1: An exemplary prime number calculation in C--.

```
1  Tasklet tasklet = Tasklet.fromFile("primes.cmm");
2  TaskletParameterList parameters = tasklet.getParameterList();
3  parameters.addInt("low", 2);
4  parameters.addInt("high", 10000);
5  QoCList qoc = tasklet.getQoCList();
6  qoc.setReliable();
7  tasklet.start();
8
9  TaskletResult results = tasklet.waitForResult();
10 ArrayList<Object> resultList = results.getResultItems();
```

Listing 5.2: An exemplary Java code snippet that uses the Tasklet Library to launch a Tasklet and to receive the results.

5.2.3. Use Cases

We implement a variety of real-world applications that perform computation offloading with Tasklets. We run these applications in the experiments or model their behavior in the Tasklet Simulator.

Mandelbrot set: This application creates a visualization of the famous Mandelbrot set. Every pixel of the resulting image represents a complex number. The color of the pixel is determined by applying a mathematical algorithm. The application is interactive as users are able to scroll and zoom in the image, which leads to new calculations.

Option pricing: The user enters different parameters about a financial option such as strike price and volatility. The application then applies Monte Carlo simulation to approximate the fair value of the option. As the accuracy of the simulation increases with more simulation runs, computation offloading is attractive for this use case. Similar to the Mandelbrot set, the option pricing application benefits considerably from a parallel execution of many Tasklets.

Ray tracing: The ray tracing technology is commonly used for the generation of digital images, e.g., in computer games. The user is able to define a set of spheres in terms of size, color, transparency, reflectiveness, and location in the image. The application uses this information to render the image. Being a computation-heavy rendering technology, ray tracing profits considerably from computation offloading.

Photo filter: The grayscale photo filter may be used in photo editing or social media apps. Users choose Red, Green, Blue (RGB) color ranges (e.g., via a range slider) and the filter application converts all pixels of an image that differ from the specified color range to grayscale.

K-means clustering: This unsupervised machine learning method clusters a data set such that members of a cluster are more similar to each other than to data points outside of the cluster. Users can choose input data sets that vary in dimensionality and size. With the help of computation offloading with Tasklets, k-means clustering assigns all data points to a user-defined number of clusters.

Decision tree classifier: The decision tree classifier — the second typical machine learning application — is useful in, e.g., pervasive healthcare use cases. For instance, it may perform classification on physiological data collected by wearables or smartphones to determine the health condition of a user. This application offloads a trained decision tree model and a data set to a provider.

Speech detection: The speech detection application identifies the periods of an audio file where people talk, similar to, e.g., Matlab's `voiceActivityDetector`. This application may be the basis for many pervasive applications such as speech enhancement for accessibility, speech coding, and speech recognition.

Further applications: The aforementioned applications are commonly used in the experiments described in this thesis. Throughout the years, we implemented additional applications from the domains of mathematics (prime number finder, π approximation, matrix multiplication), game AI (Connect Four, nine men's morris), chess problems (eight queens puzzle, knight's tour), face recognition, and ant colony optimization.

5.2.4. Tasklet Simulator

While the Tasklet Core System is suitable for real-world experiments, the Tasklet Simulator is useful for large-scale studies with dozens of applications, hundreds of providers, and millions of Tasklets. It is a representation of the Tasklet Core System in the OMNeT++[5] discrete event simulator. We validated the Tasklet Simulator to ensure that it behaves like the Tasklet Core System. Both instances use the same communication protocol. We monitored the behavior (online time, CPU, and memory utilization) of multiple office and personal computers over several weeks and developed models based on these measurements, which we fed into the simulation environment. Devices enter and leave the system based on a normal distribution modeled for each device type. We let the actual performance of each single device fluctuate to account for varying utilization (CPU, memory) based on the observed usage patterns. We measured latencies in real-world networks including the time to establish a TCP connection to model transmission times. We measured the throughput of different device types for actual computing tasks to account for heterogeneity in the system. We performed basic measurements on both, the Tasklet Simulator and the Tasklet Core System to compare execution and transmission times. Eventually, we are confident that the simulation environment reliably reflects the behavior of the Tasklet Core System.

[5]`https://omnetpp.org/`, accessed 2021-10-26

6. DataVinci

In the previous chapter, we introduced the Tasklet system. The middleware allows devices to exchange Tasklets — self-contained computational units consisting of code, parameters, and input data. Offloading of such closures leads to satisfactory completion times when tasks require no or only small amounts of input data. The more data is required for the execution, the longer it takes to offload the workload. This leads to unacceptable delays and a worse user experience. In this chapter, we present the *DataVinci* data and task placement approach. *DataVinci* is a QoC mechanism in the Tasklet system to achieve the *Speed* QoC goal for data-intensive tasks. To achieve fast response times even for such tasks, we slightly alter the abstraction of a Tasklet. Now, the input data is not shipped together with the task, but is independently exchanged in the system. Thus, *DataVinci* is able to proactively distribute the required input data to avoid latency caused by ad-hoc data transfers, i.e., data files that are sent along with the task.

Scheduling for data-intensive applications has been extensively researched in grid and cloud environments where resources are homogenous, reliable, and monitored by an omniscient controller. Results show that careful data and task placement can lead to reduced task completion times [130, 131]. Scheduling of data-intensive tasks in edge computing, however, has received little attention in research so far. In contrast to a cloud infrastructure, an individual device is not scalable but has limited storage and computing resources. Therefore, it is neither an option to send all data files to all devices, nor to send all tasks to a single computer. Instead, a performant scheduler needs to make data and task placement decisions that keep task completion time as well as data transfer at a minimum.

The complexity of scheduling decisions in edge computing is driven by multiple factors. First, there are heterogeneous resource providers that do not only vary in their computational performance but may also enter and leave the system at any time. Therefore, the scheduler must on the one hand frequently adjust the data placement and on the other hand select the best provider for a task depending on the current context. Second, users expect a timely task execution despite a varying load in the system. Thus, the scheduler should be robust to load changes in the system and adaptively choose suitable providers for data and task placement. Finally, numerous applications might simultaneously offload tasks with different data requirements that may change over time.

Data-intensive applications in edge computing such as machine learning or face recognition typically require a combination of up-to-date "private" data created by the consumer and "public" data, which is shared among multiple users and often stored in the cloud. For instance, offloading a machine learning task requires the resource provider to store both a potentially large classifier as well as the input data to be classified. On the example of image detection, classifiers easily reach a size of several hundreds of megabytes such as a pre-trained model of the convolutional neural network architecture VGG16 (490 MB). This model is publicly available and can be downloaded, e.g., from the cloud. In contrast, private data which needs to be classified differs for each user but might also add up to tens or hundreds of megabytes. Public and private data may be reused for multiple tasks when different parameter settings are evaluated for the classification. We argue that proactive data placement can reduce latencies that otherwise occur when data has to be transferred ad-hoc, i.e., when a task is started.

DataVinci is a QoC mechanism in the Tasklet system that addresses the aforementioned challenges. It carefully adjusts the number of copies and maintains sufficient replicas in the system to achieve timely task completion. We present *DataVinci* in this chapter. In Section 6.1, we review related work. In Section 6.2, we first summarize the underlying system model. The major part of the section is, however, dedicated to *DataVinci*'s design, which consists of a *data placement* level and a *task placement* level. In Section 6.3 and Section 6.4, we describe *DataVinci*'s strategies for data placement and task placement, respectively. Section 6.5 extensively evaluates *DataVinci*. We conduct a pilot study in a real-world experiment and assess *DataVinci*'s effectiveness on a larger scale in a simulation. Section 6.6 concludes the chapter with a brief summary. This chapter bases on [124][1] and [132][2]. It answers research question 2.

[1]Reference [124] is joint work with D. Schäfer, J. Edinger, and C. Becker.
[2]Reference [132] is joint work with J. Edinger, D. Schäfer, and C. Becker.

6.1. Related Work

Across all distributed computing paradigms, the question of how to place input data in the system as effectively as possible always remains a major focus in computation offloading research. In this section, we review data and task placement approaches in grid, cluster, cloud, and edge environments. We concentrate entirely on the *decision making* perspective, i.e., the strategic planning of where to place data and task. Therefore, we omit approaches such as *HTCondor*'s [133] *Stork* [34] scheduler, Reference [134], or Reference [135] that are designed to realize a desired pre-defined data distribution in a system efficiently. In addition, we omit architectures with a fixed data flow such as [136]. Table 6.1 gives an overview of related work.

In general, we categorize the literature into approaches that consider (i) data placement only, (ii) task placement only, or (iii) a combination of data and task placement similar to *DataVinci*. As far as data placement is concerned, approaches either rely on (i) caching, (ii) static replication, (iii) reactive replication, or (iv) proactive replication. When applying *caching*, providers explicitly store input data (or results) that were required by a past task execution. Thus, the data can potentially be reused in the future. Such approaches are often enriched with algorithms that decide which data file to keep, e.g., based on popularity [94, 149] or a Least Recently Used (LRU) heuristic [140]. *Static replication* encompasses strategies that distribute data in the system once. This is sufficient in scenarios where the whole future workflow is known a priori (e.g., [143] or [155]). *Reactive replication* strategies adjust the data placement in response to a trigger event. Potential triggers include peaks in the offloading demand [140, 144, 148], provider joins and leaves [74], or bottlenecks (cf. dynamic replication in Section 6.5.2). All reactive strategies have in common that they observe the system state for a certain time period before becoming active. In contrast, *proactive replication* as used in *DataVinci* adjusts the data placement *before* the event that would potentially trigger a reactive approach even happens. This comes at the cost of data transfer and monitoring overhead. Independent from the replication strategy, it is beneficial to observe the history of a certain data file to make well-informed data placement decisions. This is especially relevant for new versions of data files as described extensively in this chapter. While some approaches analyze the

		Data					**Task**		**Context**						**Ev.**	
Author/*System*	**Year**	**Caching**	**Static**	**Reactive**	**Proactive**	**Versions/History**	**Data-aware**	**Performance-aware**	**Data size**	**Storage**	**Fluctuation**	**App. requirements**	**Performance**	**Provider load**	**Simulation**	**Real-world testbed**
Alhusaini [137]	1999						●	●	○				○	○	●	
APST [138,139]	2000						●	●	●				●		●	●
Braun [89]	2001						●	●	○				●	●	●	
Ranganathan [140]	2002	●		●		○	●							●	●	
He [141]	2003						○	●				○	●	●	●	
Cameron [142]	2004	●				○	●	○		●				●	●	
Blythe [42]	2005						●	●	●			○	●	●	●	
Desprez [143]	2005		●				●	●	●	●			●		●	
Tang [144]	2006	●	●	●		○	●	●	●	●			●	●	●	
Chang [145]	2006	●					●	○	○	●				●	●	●
Venugopal [146]	2006						●	●	●				●	●		
DIANA [41]	2007						●	●	●			○	●	●		●
Chervenak [147]	2007		●				○	○	○			●	○	○		●
Ramakrishnan [130]	2007						●	●	●	●			●		●	
DistReSS [148]	2008			●		○	●	●	●	●			●	●	●	
Nukarapu [149]	2011	●	●						●	●					●	
Liu [150]	2012						●	●	●			●	●		●	
Van Den Bossche [151]	2013						●	●	●			○	●	○	●	
JDS-BC [152]	2013		●				●	●	●	●		●	●	●	●	
Choudhury [153]	2015	○					●	●	●	●		○				●
Li [154]	2016						●		●					●	●	
Nebula [74]	2017		●	●			●	●	●	●	●	●	●	●		●
BaRRS [155]	2017		●				●	○	●	●		●	●	●		●
Elbamby [94]*	2017	●				○	●	●	●	●		○	●	●	●	
iFogStor [156]	2017			●					●	●					●	
ECS [157]	2018		●							●					●	
D-ReP [158]	2018			●		○			●						●	
Cicconetti [83]	2019						●	●	●	○			●	●	●	●
Braud [93]	2020						●	●	●		●	●	●	●	●	●
DataVinci			●	●	●	●	●	●	●	●	●	●	●	●	●	●

Table 6.1.: Overview of related data and task placement approaches.
* Although the authors describe the approach in [94] as proactive, it does not match the definition of proactive replication applied in this thesis. The approach caches results of tasks to have the data available for the next request. It is therefore categorized as a caching approach. (App. = Application, Ev. = Evaluation)
● fulfilled ○ partially fulfilled

past usage of a data file to infer its popularity (e.g., [158]), none of the reviewed approaches explicitly consider different versions of data files.

In terms of task placement, we distinguish between data-aware and performance-aware strategies. *Data-aware* task placement strategies consider the location of data in the scheduling process; *performance-aware* strategies the computational capabilities (or throughput) of a provider. We observe that most approaches that offer task placement consider both data locality and provider performance in their decision making. An intuitive way to combine both in a single decision function is to model and estimate the task completion time at each provider and choose the one with the lowest estimate (e.g., [42, 83, 93]). Only two of the approaches [140, 154] schedule tasks independent from the throughput of the providers, which is only reasonable in purely homogeneous systems.

Edge computing systems consist of heterogeneous devices that run a large variety of applications. In addition, devices join and leave such systems frequently and without restrictions. Therefore, we argue that proper data and task placement in such environments considers the context for adaptive decision making. We review the literature with regards to context-awareness and assess whether the approaches consider the data size, storage capacity, fluctuation (i.e., provider stability), application requirements, provider performance, and provider load. We observe that a majority monitors the data size and also incorporates the context of the provider (performance and/or load) in the decision making. Many approaches also consider the storage capacity of the devices. This context dimension, however, is usually interpreted as a binary variable — (i) no storage left or (ii) sufficient storage left — instead of a continuous value to establish a "fair" data placement among providers. The treatment of application requirements including degree of parallelism [42, 93, 147, 150, 152, 153, 155], task complexity [41, 42, 93, 94, 150–152, 155], deadlines [93, 151], or QoC constraints such as security, bandwidth, or reliability [74, 141, 150] is rather diverse in the literature. An exemplary approach with a strong focus on application requirements is [93]. In this paper, Braud *et al.* explicitly exploit application characteristics for an AR offloading approach that uses other mobile devices, edge servers, and cloud servers as providers. We observe that only *Nebula* [74] and the approach by Braud *et al.* [93] consider fluctuation. Remarkably, these are recent approaches designed for edge computing environments. As such systems are highly dynamic, we argue that it is vital to monitor fluctuation and consider it in the data and task placement decisions.

Except for [146], all reviewed approaches have been evaluated in a simulation or a real-world testbed. While we acknowledge the effort that is required to assess a strategy under real-world conditions, we still argue that such experiments provide essential insights on the way towards an implementation in practice. Similar to the evaluation of *DataVinci* later in this chapter, Cicconetti *et al.* first apply their approach for serverless computing [83] in a real-world mobile edge testbed before scaling the experiments up in a simulation. Casanova *et al.* use a different methodology. They first verify *APST* in a simulation [139] before deploying it in a real-world scenario in [138].

We conclude from our literature review that data-aware task placement is well-studied in comparison to data placement. In this chapter, we propose *DataVinci* — a scheduler with a strong focus on data placement. In contrast to related work, *DataVinci* applies proactive replication. *DataVinci* is one of only a few context-aware approaches that consider a broad variety of context dimensions. We evaluate *DataVinci* in a real-world testbed and in a simulation.

6.2. Design

In this section, we first the underlying system model and *DataVinci*'s fundamental design, which consists of a data placement and a task placement level.

6.2.1. System Model

Applications in the system consist of two parts. First, the application logic contains executable, computationally intensive parts of the application that can be executed on the TVM as Tasklets. Second, applications have data dependencies which determine the data files that are required to exist on the machine where the task is to be executed. Data requirements from applications can be divided into two categories: public and private data files. Public data files are shared by all users of this application and are downloadable for the providers from well-connected data clouds. Private data files are unique for each user and are exchanged directly between consumers and providers. Applications can require either public or private data, or both. A provider is able to start the execution of a task as soon as it stores all required data. Once a data file has been sent, it is stored until the provider leaves the system. Multiple identical copies of the same data files (so-called *replicas*) may exist on different devices. Both public and private data can be reused for multiple tasks, e.g., for object detection tasks with different parameters. The data files for applications might change over time, leading to different data file versions. Once the data file is updated, applications require the new version for the execution of their tasks. Public and private data files might get updated independently from each other at any time. Outdated versions of the data files are not required anymore and might be deleted.

6.2.2. Data and Task Placement

DataVinci is an integrated scheduling approach that manages both, data placement and task placement. This integration into one scheduler leads to better optimization of the system state and less overhead for negotiation between two independent schedulers. In addition, the integrated design avoids oscillation effects, that might occur if separated data and task schedulers repeatedly counteract each other's decisions. *DataVinci* does not assume knowledge about the future workflow or dependencies between data files and certain tasks. This uncertainty necessitates a separation of data and task placement in the scheduler.

As far as data placement is concerned, *DataVinci* uses *initial replication*, which is triggered as soon as new data files or new versions of existing data files enter the system, such as when a new pre-trained classifier becomes available. Here, *DataVinci* differentiates between (i) an entirely new data file and (ii) a new version of an existing data file. For new data files, *DataVinci* offers *context-aware replication*. This strategy monitors several context dimensions such as data size or application requirements to approximate the suitable number of replicas. As far as new versions of an existing data file are concerned, *DataVinci* is able to exploit knowledge about the previous versions to make even better decisions. The strategies for such cases analyze usage patterns of old versions to determine a suitable number of replicas for the new version. We have coined the term *history-based replication* for these strategies.

In addition to *initial replication* with its two facets, *DataVinci* further uses *continuous replication*, which permanently adapts the data distribution to the current system state. Continuous replication copes with bottlenecks that result from an insufficient data distribution. Due to varying system load or device fluctuation, it is necessary to create new replicas of certain data files over time in order to maintain timely task execution, even if initial replication works properly. *Static* replication is a continuous replication strategy that maintains a stable number of replicas. It distributes new replicas if devices that stored certain data files leave the system. *Reactive continuous replication* distributes new copies if it detects bottlenecks that occur when data files are sent along with the task. *Proactive continuous replication* avoids bottlenecks in the first place. For each data file, the strategy constantly ensures a certain amount of idle resources to prevent ad-hoc data transfers.

Ideally, task placement exploits the current data distribution to minimize task completion times. *DataVinci* offers four task placement strategies. In contrast to *random* task placement, a *performance-aware* strategy allocates tasks to the fastest idle provider. In data-intensive scenarios, performance differences among providers may be less severe than latencies of ad-hoc data transfers. Thus, *data-aware* task placement schedules tasks on providers that already store the required input data files. *DataVinci* further introduces *hybrid* task placement, which combines data-aware and performance-aware scheduling. Figure 6.1 summarizes *DataVinci*'s strategies, which we discuss in greater detail in Section 6.3 (data placement) and Section 6.4 (task placement).

6.3. Data Placement Strategies

DataVinci's data placement level optimizes the distribution of new data or new data versions in the system by applying replication, i.e., the management of various copies of the same data distributed on different computers [159]. *DataVinci* uses replication to proactively distribute data files on multiple providers. These providers are then suitable offloading targets for future executions of tasks that require these data files. Hence, *DataVinci* improves task completion times of data-intensive applications by avoiding ad-hoc data transfers.

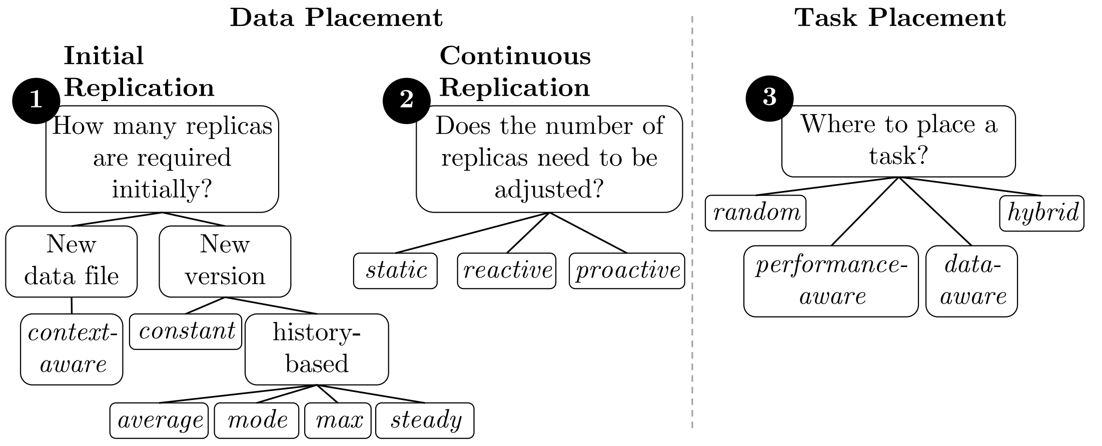


Figure 6.1.: *DataVinci* makes three types of decisions. 1 When new data files or new versions of existing data files enter the system, the scheduler decides how many replicas need to be placed on remote devices. 2 When the context changes (e.g., because devices leave the system or the system load increases), *DataVinci* might adjust the number of replicas. This adjustment can either keep the number of replicas stable (static), react to bottlenecks in the system (reactive), or avoid bottlenecks (proactive). 3 For each task, *DataVinci* selects a provider either randomly or based on the computational performance of the providers, the data distribution, or both.

The design of a replication-based scheduling strategy needs to consider the tradeoff between low task completion times and data transfer overhead. Two extreme strategies exist: *no replication* where no data files are distributed proactively and *full replication* where all data files are sent to all devices. However, between those two extremes, strategies that balance data transfer overhead and task completion times exist. The design of such as strategy has to address two questions in particular: (i) how many replicas should the environment store in total and (ii) which providers are most suitable to store these replicas?

DataVinci applies replication in three cases: (i) when a new data file becomes available, (ii) when a new version of an existing data file becomes available, and (iii) when the current context requires an adjustment of the data distribution. When a new data file or a new version becomes available, *DataVinci* performs *initial replication.* We introduce several strategies for choosing an adequate number of replicas for initial replication in Section 6.3.1 (new data file) and in Section 6.3.2 (new version). As it is necessary to continuously adjust the number of replicas according to the current context, *DataVinci* performs *continuous replication* using one of the strategies presented in Section 6.3.3.

6.3.1. Initial Replication of New Data Files

Designing a proper data placement strategy for an entirely new data file is challenging as the information that is available for decision making is comparably scarce. It is, for instance, uncertain how many consumers will offload tasks that require the new file or how long the execution times of these tasks will be. Nonetheless, *DataVinci* includes a *context-aware replication* strategy that considers several context dimensions to approximate the optimal number and placement of replicas. Context-aware replication encompasses two decisions for each new data file d. First, it needs to determine the number of replicas n. Second, the strategy chooses the providers that will store the replicas. A separation of these two steps ensures a reasonable number of replicas during execution without relying on device monitoring or assuming certain device characteristics. Concerning the first decision, four context variables play a major role. The appropriate number of replicas depends on the data size, the remaining storage capacity of the system, the current fluctuation, and the application. *DataVinci* models the current state of these four variables with normalized coefficients ranging from 0 to 1. The coefficient C_{data} describes the relative data size by comparing the absolute data size s_d to the maximum allowed data size in the system s_{max}. In a system with k providers $p_1, ..., p_k$, the coefficient for the remaining storage capacity, C_{cap}, is the sum of the free storage c_f of all devices divided by the sum of the total storage c_t of each device.

The influence of the current fluctuation on the desired number of replicas is modeled in the coefficient C_{flu}. To calculate this coefficient, *DataVinci* applies a sliding window approach and determines the mean residence times for each provider that has been part of the system in this time window. The average of these mean residence times quantifies the current provider stability $Stab_{prov}$. There is, however, no linear relation between provider stability and the appropriate number of replicas. A high number of replicas is beneficial in systems with moderate stability values. In unstable systems with high fluctuation, fewer replicas should be chosen since replicas are likely to leave the system before any task execution

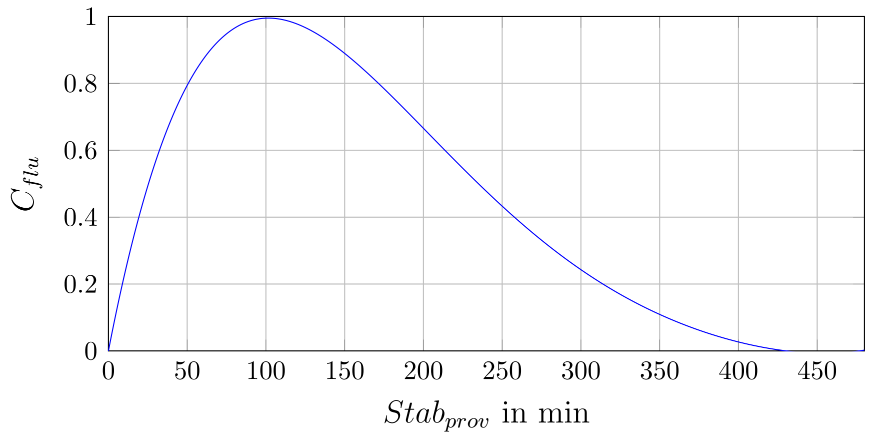


Figure 6.2.: Function to determine C_{flu} based on the provider stability value.

can happen on the device that stores the replica. Additionally, systems with high stability values also require less replicas. Devices barely leave the system which makes having another copy inefficient. To model this non-linear relation, we apply a polynomial function as shown in Figure 6.2. This function represents an example and can be exchanged depending on system or application characteristics.

Moreover, the application characteristics influence replication. To incorporate these application-specific characteristics into the replication decision, the coefficient C_{app} represents whether the application might require a higher number of replicas. *DataVinci* calculates this coefficient as the average of the data availability factor f_{ava} and the parallelism factor f_{par} of the application that entered the data into the system. Applications with a high data availability factor benefit from having multiple replicas since they require to continuously have an available copy on a provider to ensure fast execution. A high parallelism factor models that an application runs parallel tasks on the same data, which also requires multiple replicas. These two factors, ranging from 0 to 1, may be transmitted to *DataVinci* by the application itself or observed at runtime. The equations for calculating the three coefficients C_{data}, C_{cap}, and C_{app} are as follows:

$$C_{data} = \frac{s_d}{s_{max}}; C_{cap} = \frac{\sum_{i=1}^{k} c_{f_i}}{\sum_{i=1}^{k} c_{t_i}}; C_{app} = \frac{f_{ava} + f_{par}}{2} \tag{6.1}$$

The relative importance of the four context coefficients can vary, depending on system and application state. Therefore, we multiply weights α_1 to α_4 to the coefficients. The sum of these weights further determines the maximum number of replicas in the system. In edge environments, k — the number of devices in the system — changes and does not necessarily equal the norm size of the system that was used during design time (k_0). Hence, we scale the resulting number of replicas depending on the relative size of the current system compared to the norm size. The final equation to calculate the number of replicas n then looks as follows:

$$n = (\alpha_1 * C_{data} + \alpha_2 * C_{cap} + \alpha_3 * C_{flu} + \alpha_4 * C_{app}) * \frac{k}{k_0} \tag{6.2}$$

The weights allow system designers to adjust the replication strategy to the needs of their system. For instance, in systems with large bandwidths that focus on fast execution of time-critical tasks, α_1 might be positive. With increasing data size, the number of replicas also increases as migrating aborted tasks becomes particularly costly. Systems with low bandwidth or limited storage capacities may use a negative α_1 to decrease the data transfer overhead. A proper choice of the weights is crucial for the effectiveness of the replication strategy. Unsuitable weights may lead to unsatisfactory results such as a constant shortage of replicas. Using a control structure that adapts the weights dynamically at runtime can reduce the risk of choosing unsuitable weights in the first place.

So far, we have considered context dimensions of the data file and the system itself to determine an adequate number of replicas. For the decision on where to store the replicas, *DataVinci* additionally takes the characteristics of the providers into account. If possible, replicas should be stored on devices that will not leave the system until the data transfer was worthwhile. Thus, the *stability* of the providers is an important context variable. We characterize the stability by considering mean μ_p and variance σ_p^2 of the devices' residence times. The *mean residence time* is a relevant context dimension as it helps to predict whether a provider remains connected for a longer time on average. However, some devices may be connected for a long time on average but still leave the system after a short while in some cases. To consider this behavior in the decision, we add the *variance of the residence time* as a context variable to distinguish stable resources from unpredictable resources. *DataVinci* estimates the values for μ_p and σ_p^2 based on the past residence times of the providers in the system. Further, it monitors the current *residence time* t_p.

In addition, the *storage load* c_p of the providers is a relevant context variable. If a device faces a high storage load, replicating additional data on this resource may not be beneficial. Further, *DataVinci* also takes the *data queue sizes* q_p of the providers into account. Devices that store a large amount of replicas will likely execute a larger number of tasks in the future. New replicas should be stored on devices that store less data to avoid task queues and to allow a timely execution of the associated task for this replica. Finally, the *relative performance index* RPI of a device determines its computational performance compared to the average performance of the current environment based on a benchmark measurement. A provider with a high RPI stores more replicas due to the processing performance. Combining all of these context variables into an equation to calculate a provider's utility U_p leads to the following function:

$$U_p = \beta_1 * (\mu_p - t_p) + \beta_2 * \sigma_p^2 + \beta_3 * c_p - \beta_4 * q_p + \beta_5 * RPI \tag{6.3}$$

Context-aware replication now places the new replicas on the n providers with the highest utility value. Similar to Equation 6.2, the context variables are attached with weights β_1 to β_5 to allow a customization of the function. Since the choice of relevant context dimensions does not claim to be exhaustive for all use cases, Equations 6.2 and 6.3 both allow to integrate further context dimensions as addends if required.

6.3.2. Initial Replication of New Data Versions

The previous section shows how *DataVinci* approximates the optimal number of replicas for a new data file. If a new version of a data file becomes available, *DataVinci*'s decision making has more information available. The scheduler is able to analyze the data distribution of the previous version in the system over time and use this information for the current decision. *DataVinci* includes two basic strategies to select the suitable number of replicas for new versions: *constant replication* and *history-based replication*. These strategies differ in how they determine the number of replicas. For the choice of the providers that store these replicas, they reuse the utility-based approach introduced in the previous section. The *constant replication* strategy creates a fixed number of replicas n, where n is a parameter adjustable by the system administrator or the value determined by context-aware replication. This strategy does not consider the usage of the prior version for decision making. *DataVinci* sends the application data along with the task if n is set to 0 and continuous replication is deactivated. Thus, setting n to 0 optimizes initial data transfer overhead. However, it leads to high task execution times as each execution will always include the latency for the data transmission.

We propose *history-based replication*, a data placement strategy that analyzes the data distribution of the previous version in the system over time. In many cases, the number of providers that store a certain data file is not stable during the lifespan of a version. The number may change after the initial placement due to provider fluctuation or peak demand, which might necessitate placing additional copies of the data file on other providers. As history-based strategies, we introduce *average-based*, *mode-based*, *maximum-based*, and *steady* replication. In average- and mode-based replication, *DataVinci* monitors the current number of replicas in the system at each time step and aggregates this information into an average or mode value that describes the whole period for the latest version. Then, *DataVinci* selects the average or the mode of the monitored number of replicas as the desired number of replicas for the new version. These two strategies aim at providing a suitable data distribution for the majority of task executions without creating heavy data transmission overhead. In contrast, maximum-based replication selects the maximum number of replicas observed during the lifespan of the prior version as the initial number of replicas for a new version. Maximum-based replication ensures — assuming that the demand for the particular data file remains similar — a sufficiently large number of replicas for parallel, fast task execution even in times of peak demand. A higher number of replicas, however, requires more data transfer. The steady replication strategy relies on the outcome of the continuous replication. When a new version becomes available, the scheduler restores the current number of copies in the system. The strategy assumes that continuous replication performs well and that it is not biased by short term peaks in the demand. Figure 6.3 depicts the number of replicas chosen by *DataVinci*'s strategies for an exemplary data file when a new version becomes available.

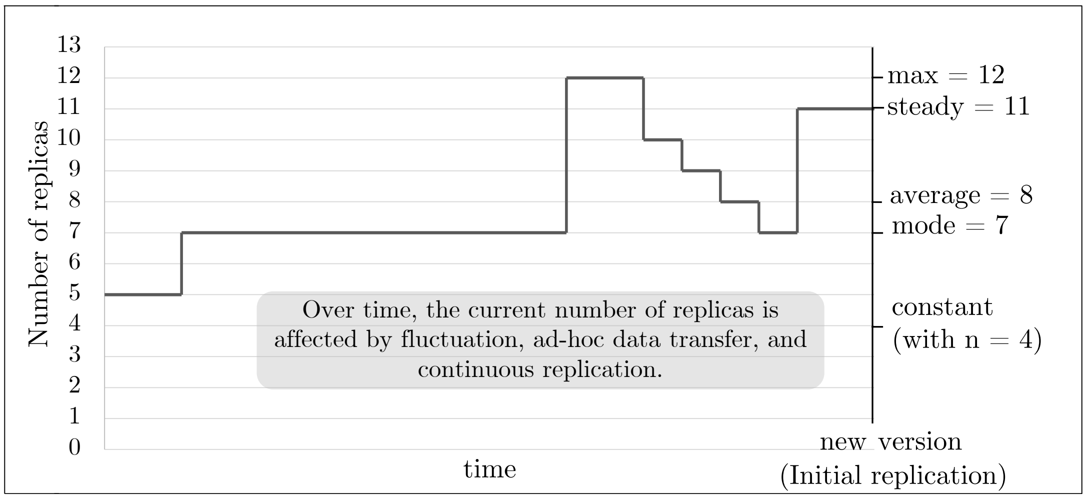


Figure 6.3.: Initial replication: An exemplary development of the number of replicas for one data file version. The resulting number of replicas for a new version under different strategies is shown on the right.

6.3.3. Continuous Replication

DataVinci uses continuous replication, which adjusts the current data distribution if required. Device fluctuation, different user behavior, and the uncertainty during initial replication may necessitate changes. Adjusting the data placement has proven to be effective in several related approaches [142, 144, 145]. *DataVinci* offers proactive continuous replication, which aims at avoiding bottlenecks in advance. *DataVinci* monitors the providers including the data files they store, the computational resources they offer, and the tasks that they are currently running. Here, resources may be any kind of computing instances that can be measured in discrete units such as idle processor cores or idle TVMs. Based on this overview of the system state, proactive replication ensures that a certain number of idle resources ("buffer") is continuously available for all data files. *DataVinci* triggers the creation of one or more new replicas if it detects that an insufficient number of idle resources is able to perform tasks on the data file. Thus, even in times of rising demand for computation involving certain data files, tasks can be scheduled on the buffer resources while proactive continuous replication creates new replicas in the background. Setting a high buffer ensures that sufficient replicas are available but also increases the amount of data transfer. Figure 6.4 illustrates proactive continuous replication in an exemplary case.

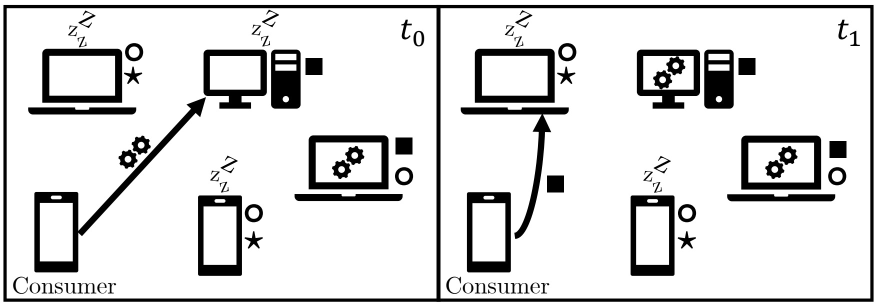


Figure 6.4.: Proactive continuous replication with a buffer value of 1 in an example with data files ⋆, ○, and ■. At t_0, continuous replication is triggered as the number of idle resources that store ■ falls below the buffer value. *DataVinci* distributes a new replica to the most suitable provider at t_1.

6.4. Task Placement Strategies

The superior goal of *DataVinci* is to improve the task completion times of data-intensive applications. The data placement that is realized with the strategies presented in the previous sections is only effective with regards to this goal if the task placement exploits the data distribution that is created and maintained. To achieve this, *DataVinci* offers a variety of task placement strategies: *random*, *data-aware*, *performance-aware*, and *hybrid* task placement. *Random task placement* allocates tasks randomly to idle providers. Since tasks are scheduled independently from the current data distribution in the system, additional data transfers may happen while using this strategy, even if the data files are present on other providers in the system. *Data-aware task placement* exploits the data distribution that has resulted from data replication. It analyzes the data requirements of a task and divides the providers into those that store all data files required for the task and those that lack data files. At first, the data-aware scheduling strategy randomly chooses providers from the group of idle providers that store all required data files. Only if the providers in this group do not offer sufficient resources, the strategy considers idle providers without all data files.

Performance-aware task placement sorts the idle providers based on their throughput. *DataVinci* then allocates the tasks to the fastest idle provider, independent from the data files stored on this provider. *Hybrid task placement* (Algorithm 1) takes both data distribution and provider performance into account. It first divides the providers into the group of idle providers that store all input data files and the group of idle providers that lack certain data files, similar to data-aware task placement. The providers that already store the data files are sorted by throughput. The other providers are sorted by the total amount of data that is missing for the task execution to keep additional data transfer at a minimum. The scheduling strategy would prefer a provider that stores all but one input data file over a provider that stores none of the required data files if a task requires multiple data files of various sizes.

Algorithm 1 Hybrid Task Placement (Task T with data requirements D_T)

```
1: let $P_{idle,data}$ and $P_{idle,noData}$ be empty provider lists
2: for all Providers $p$ do
3:     if $p$ is idle then
4:         if $p$ stores all data files $d \in D_T$ then
5:             add $p$ to $P_{idle,data}$
6:         else
7:             add $p$ to $P_{idle,noData}$
8: sort $P_{idle,data}$ by throughput in descending order
9: for all Provider p in $P_{idle,noData}$ do
10:     $dataStored_p \leftarrow \sum size((d \in D_T)|(p \text{ stores } d))$
11: sort $P_{idle,noData}$ by $dataStored_p$ in descending order
12: while $T$ requires more resources do
13:     if $P_{idle,data}$ is not empty then
14:         select first provider $p$ from $P_{idle,data}$
15:         remove $p$ from $P_{idle,data}$
16:     else
17:         select first provider $p$ from $P_{idle,noData}$
18:         remove $p$ from $P_{idle,noData}$
```

6.5. Evaluation

We evaluate *DataVinci* in two steps. First, we deploy it in a small-scale real-world experiment (Section 6.5.1). This pilot study provides valuable insights that are used in the second step — a large-scale simulator-based experiment (Section 6.5.2). In Section 6.5.3, we discuss the evaluation results obtained from the two studies. We present potential threats to validity in Section 6.5.4.

6.5.1. Real-World Pilot Study

We deploy *DataVinci* in a real-world testbed for this pilot study. The study focuses on the initial replication of new data files with the context-aware replication strategy. We investigate the long-term effects of new data versions and a comparison of the different initial replication strategies for new data versions, continuous replication strategies, and system parameters (e.g., system load or buffer value for proactive continuous replication) in the large-scale study presented in the next section.

The usage of a real-world testbed in this study allows us to show the practical feasibility of the approach and to gather real-world data about latencies or computational power for the subsequent study. Moreover, environmental details and realistic characteristics of the network layer influence the measurements. This comes at the cost of less controllability and natural variances compared to a simulator-based evaluation approach, which we use for the large-scale study.

Experimental Setup

We implemented a prototype of *DataVinci* in Java. To avoid possible influences of performance fluctuations or side effects of the Tasklet Middleware, we connected our prototype to a Tasklet system emulator. This emulator shows equivalent behavior as the Tasklet system but allows a more controlled and steady setup. To deploy the prototype, we created a real-world testbed consisting of eleven physical devices. One of the devices acts as the consumer and hosts the broker. The other ten devices are resource providers. To exclude hardware influences, we used homogeneous devices and configured them to have the characteristics of desktop PCs, laptops, and smartphones in terms of computational performance and fluctuation (cf. Table 6.2). Now, the setup resembles an office environment in the academic sector with three office rooms and a student lab depicted in Figure 6.5. Leaving devices delete all of their data and reenter the system after a randomly chosen time interval.

As shown in Table 6.3, we run three applications in a 60-minutes-workflow in this setup. The workflow contains 303 Tasklet executions. To compare the performance of the strategies, we measure the Tasklet completion time, the

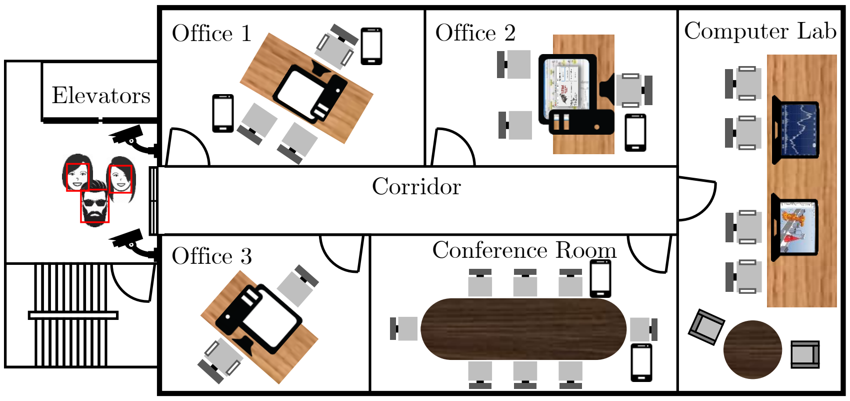


Figure 6.5.: The testbed used in the pilot study represents a typical office environment in the academic sector. Ten devices act as resource providers with different characteristics (cf. Table 6.2). In the evaluation, this edge computing environment runs three applications (cf. Table 6.3).

Device	**Benchmark**	μ **(in h)**	σ **(in h)**
3 x PC	7	7	2.5
2 x Laptop	10	4	2.7
3 x Phone (stable)	20	1.5	1
2 x Phone(unstable)	20	0.5	0.5

Table 6.2.: The ten evaluation devices in the pilot study with their benchmark performance, mean residence time μ and standard deviation of the residence time σ.

queuing time, the execution time, and the data transfer overhead. The queuing time quantifies the time span between the arrival of the Tasklet at the provider and the execution start. We assess context-aware replication in comparison to three baseline approaches: (i) no replication, (ii) a static replication strategy that always creates one replica ("*1-replication*"), and (iii) full replication. As the importance of a proper mechanism for initial replication of new data versions and continuous replication increases with the time horizon, we study the different strategy options only in the large-scale simulation in the next section.

We run the prototype with three applications of varying data and computational intensity in the academic office environment (cf. Table 6.3). First, a *face recognition* workflow consists of events that occur if cameras at the entrance of the office building use face recognition to grant access. Face recognition compares the current camera image to entries of a comparatively large database file. Second, a

Application	**# Tasks**	**Task completion times (in s)**	**Input data (in MB)**
Face detection	243	4	120
Machine learning	40	30, 60, 120, 180	10, 15, 20, 25, 30
Monte Carlo simulation	20	300	5
Combined	**303**	**-all-**	**-all-**

Table 6.3.: In the evaluation, the edge computing environment runs three applications with different characteristics concerning number of tasks, task complexity, and data intensity.

researcher in office 2 tests different *machine learning* algorithms on multiple input files of varying sizes. Third, students in the computer lab perform computationally intensive *simulations* on comparatively small input files. A poisson point process is used to generate realistic timing of task events in these workflows. To avoid unintended variations in the execution of the Tasklets, the prototype uses *emulated Tasklets* with fixed complexities.

Results

The initial replication of a new data file leads to a data transfer overhead. Figure 6.6 shows the total amount of data shipped from the consumer to the providers for all initial replication approaches. It becomes visible that the full replication strategy, which is most promising from a Tasklet completion time perspective, also leads to high data transfer overhead in the pre-execution phase. While applying a task placement strategy that considers the data placement, no data transfers occur during runtime for 1-replication, full replication, and context-aware replication. Contrary, the no replication strategy does not require initial data transfers but leads to a considerable amount of transferred data during runtime, i.e., ad-hoc data transfers. Since the number of tasks is higher than the number of providers, coupling data and tasks in the no replication strategy is considerably more costly in terms of data transfer compared to any other strategy.

After having distributed the data in the system according to the data placement strategy, *DataVinci* allocates tasks to resources. First, we apply the random task placement strategy as a baseline. Figure 6.7 depicts the average Tasklet completion times for each of the four initial replication strategies. As expected, a full replication strategy performs best in this setting. Since all providers already

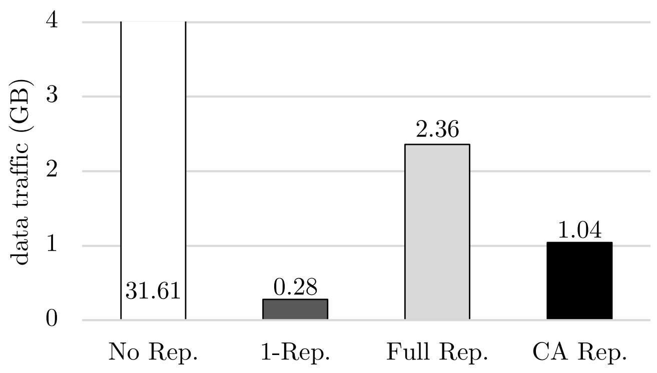


Figure 6.6.: The initial data transfer overhead for creating a 1-replication, a full replication, and a context-aware replication in comparison to the overhead of transferring the data coupled with the tasks.

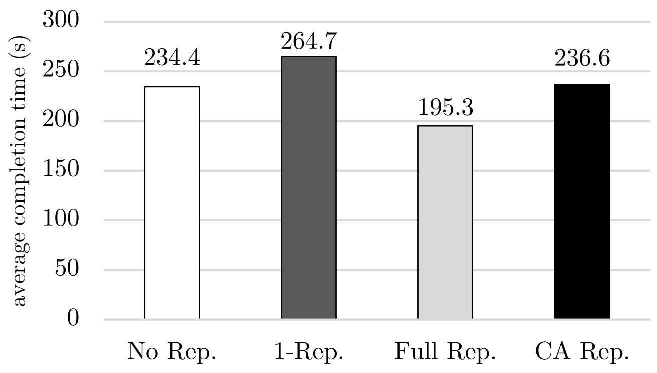


Figure 6.7.: Average task completion times for the different initial replication strategies when applying random task placement.

store the data, no data transfers are necessary and completion times only consist of queuing time and execution time. No replication, 1-replication, and context-aware replication perform substantially worse.

Now, we apply hybrid task placement. As depicted in Figure 6.8, the combination of context-aware data replication and hybrid task scheduling now reaches similar task completion times to the optimal combination of a constant full replication and hybrid task scheduling. Average task completion times are 32.7 s compared to 28.8 s in a full replication. Thus, context-aware replication together with appropriate task placement can be comparably as effective as a full replication while requiring a substantially lower data transfer overhead.

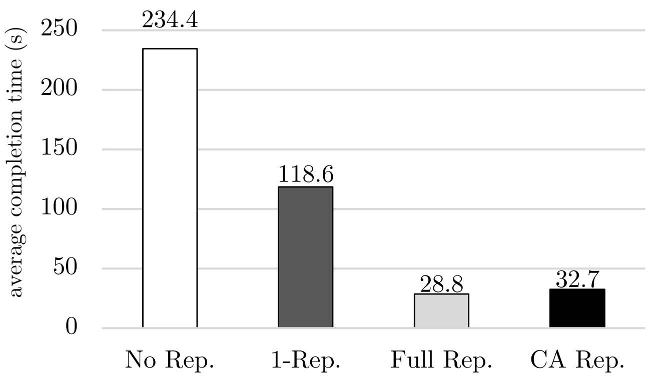


Figure 6.8.: Average task completion times for the different initial replication strategies when applying hybrid task placement.

6.5.2. Large-Scale Simulation

The previous section presents the small-scale evaluation of context-aware replication in a real-world testbed. It shows the effectiveness of *DataVinci*'s initial replication strategy for new data files. Now, we assess *DataVinci*'s potential over a larger time horizon in a subsequent large-scale study. In this study, we focus on a comparison of initial replication strategies, continuous replication strategies, and system parameters (e.g., system load or the buffer value of proactive continuous replication). This complements the findings from the small-scale pilot study. To meet the requirements of such a large-scale experiment with a high number of different settings, a simulation-based approach is most suitable. Thus, we implemented a prototype of *DataVinci* in the Tasklet Simulator, which allows us to assess the scheduling strategies with a high number of nodes in a controlled and reproducible setup. Additionally, we are able to evaluate the effects of several applications with different characteristics running on consumers in parallel. To parametrize the characteristics of the underlying network in the simulator with realistic values for, e.g., communication delay and bandwidth, we refer to observed values from the pilot study.

We designed ten applications that vary in the number of Tasklets per job, Tasklet complexity, and the rate in which they produce Tasklets. These applications have different requirements in terms of private and public data. Whereas the private data is always unique for each consumer, the public data is shared among all consumers. Both private and public data update after a certain interval. In a face recognition application, for instance, the database of known faces would change at times. We set these application parameters according to the behavior of typical offloading applications such as ray tracing, financial option pricing, and clustering that we developed for the Tasklet Core System (cf. Section 5.2.3). Table 6.4 gives an overview of the ten applications.

Application	A	B	C	D	E	F	G	H	I	J
Parallel Tasklets	4	2	12	4	3	4	6	10	2	5
Minimal complexity	480	10,000	7,200	2,200	4,000	400	6,000	600	2,000	10,000
Maximal complexity	800	14,000	12,000	3,000	8,000	500	12,000	800	3,000	12,000
Tasklet interval	120	600	2,000	500	750	80	1,000	200	400	680
Data size (public)	400	none	500, 500, 500	300	100	60	50, 200	370	50, 240	80
Data size (private)	50	100, 200	none	60, 120	10, 20, 30	15, 25	30, 70	45	75	none
Update interval (public)	2,400	-	20,000	5,000	10,000	1,200	2,000	4,000	2,900	10,000
Update interval (private)	1,200	3,600	-	2,500	5,000	600	1,000	2,000	1,450	-

Table 6.4.: Consumers run applications from this pool of ten applications. Minimal and maximal complexity are abstract integer values that represent the computational complexity of a Tasklet. Tasklet interval and update interval are average values in seconds. The entries for public and private data sizes are in MB.

Experimental Setup

The OMNeT++ simulation environment contains 200 providers and 25 consumers. One additional node acts as the central broker that runs *DataVinci*. A data cloud stores and offers the public data files. This system size resembles an edge computing scenario in an office building, a university department, or a public place with adjoining houses. Having more providers than consumers might seem counter-intuitive at first, but resembles the usage patterns that we have observed on multiple devices over several weeks. Provider and consumer nodes are configured to have the characteristics of three real-world device groups:

smartphones, desktop PCs, and laptops. The smartphone group contains twice as many nodes as the groups of desktop PCs and laptops, which are of equal size. Each time providers enter the system, they draw their residence time from a normal distribution with a mean of 420 min and a standard deviation of 150 min for desktop PCs, 240 min/162 min for laptops, and 60 min/60 min for smartphones. Devices that leave the system delete all their data files and reenter the system after a randomly chosen time interval. Each node has an individual performance value that quantifies its throughput. The fastest devices in the evaluation are four times faster than the slowest devices, which corresponds to the ratio that we observed during real-world measurements with the Tasklet system [126]. Providers host four TVMs. Each consumer runs between one and three randomly chosen applications from the pool of ten applications. The bandwidth between consumers and providers is 10 MBit/s. The download speed from the data cloud is 100 MBit/s. The latency varies between 50 and 250 ms including a TCP handshake.

In total, we evaluate 464 combinations of initial replication, continuous replication, and task placement strategies. Each combination is simulated for eight hours, which resembles one day of work. We evaluate each combination with 25 different system loads. Figure 6.9 depicts an exemplary progress of the system load over time for one of the 25 runs. This simulates the fluctuation of offloading demand over the day in an office, a university department, or a public space. Due to the varying system load, the consumer nodes start between 17,177 and 65,842 Tasklets per eight hour run, leading to a total of around 13,000,000 Tasklet executions. To compare the performance of the strategies, we measure Tasklet completion time and data transfer overhead. The Tasklet completion time is divided into data queuing time — the time span a Tasklet waits for required data files on the provider — and execution time. In our prototype, data transfers can result from initial replication, continuous replication, or from data enclosed to Tasklets.

We perform four experiments. In each experiment, we test all four task placement strategies in combination with different data replication strategies. In experiment 1, we compare the different continuous replication strategies. In experiment 2, we vary the buffer size for the proactive continuous replication strategy. In experiment 3, we vary the number of replicas for the constant initial replication strategy. In experiment 4, we evaluate different initial replication strategies. The setup of the experiments is summarized in Table 6.5.

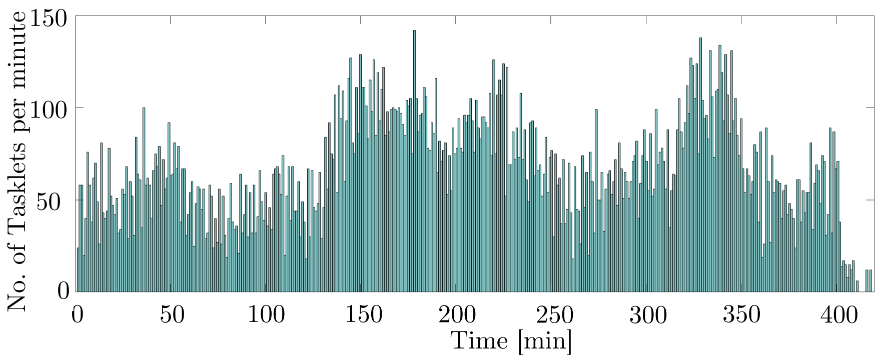


Figure 6.9.: System load during an exemplary evaluation run.

	Experiment 1	**Experiment 2**	**Experiment 3**	**Experiment 4**
Initial replication	none	none	constant	avg, mode, max, steady
Number of replicas	-	-	1,2,4,8,16,32	-
Continuous replication	none, static, reactive, proactive	proactive	proactive	proactive
Buffer size	-,-,-,4	2,4,8,16,32	4	4
Task placement	random, data-aware, performance-aware, hybrid			
Figure	6.10	6.11	6.12	6.13

Table 6.5.: Setup of the four experiments in the large-scale study.

Results

In the following, we discuss the results of the four experiments, which are visualized in Figures 6.10-6.13. In the upper half, each figure shows the mean completion time of the Tasklets for different strategies. The lower half shows the overall amount of data that is transferred in each setting. All figures consist of four columns, each representing one of the four task placement strategies.

Experiment 1 (Continuous Replication): We first evaluate the system's behavior without initial replication. This allows us to isolate the effect of *DataVinci*'s continuous replication. In this experiment, we compare the performance of proactive continuous replication to static and dynamic replication. Static replication distributes a new replica if a provider previously storing a replica went offline. Dynamic replication is a reactive strategy that distributes a new replica if the completion time of a previous task contains a proportion of data queuing time that is above a certain limit such as 25 %.

The results in Figure 6.10 (left column) show that all continuous replication strategies have a negligible effect if they are used together with random task placement. This confirms the results from the pilot study. In this case, continuous replication distributes a sufficient number of replicas, but the task placement strategy does not exploit the data distribution. The amount of transferred data is high due to many ad-hoc data transfers. When applying any of the three other task placement strategies (data-aware, performance-aware, hybrid), proactive continuous replication is faster than static and reactive replication.

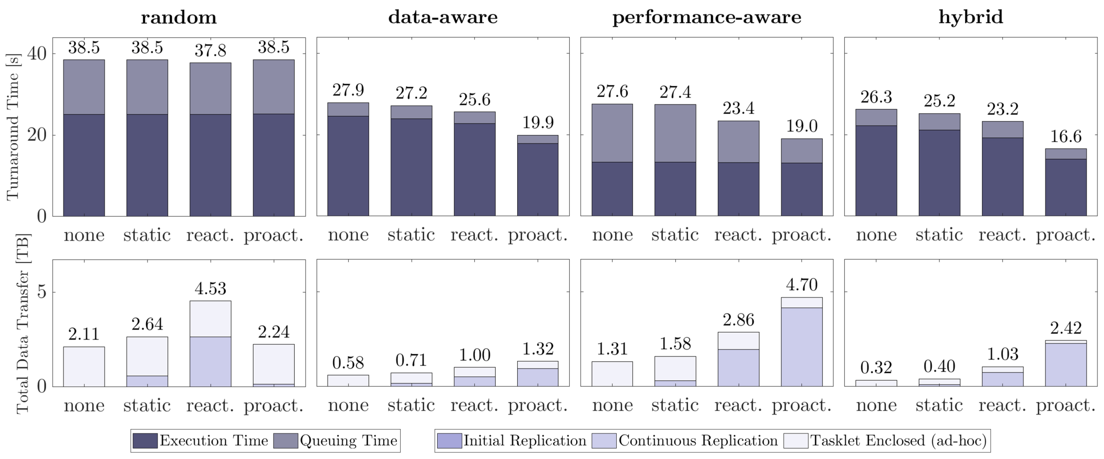


Figure 6.10.: **Experiment 1**: Comparison of Tasklet completion times and total data transfer overhead for static, reactive, and proactive continuous replication, depending on the task placement strategy (random, data-aware, performance-aware, or hybrid). Initial replication is disabled.

The results of this experiment provide four important insights. First, the proactive continuous replication strategy outperforms all other strategies in terms of task completion times. Thus, we always apply the proactive continuous replication strategy in the remaining experiments. Second, a decrease in the task completion time comes along with an increased amount of data transfer. Third, the majority of the additional data transfer is caused by continuous replication instead of ad-hoc data transfers, which indicates that the required data is often already present at the respective provider. Fourth, the combination of hybrid task scheduling and no continuous replication minimizes the data transfer but results in above average execution times.

Experiment 2 (Varied Buffer): Next, we study the effect of a varied buffer size for the proactive continuous replication strategy. The buffer parameter defines the number of idle resources (TVMs) that need to be available in the system for each data file. A sufficient buffer avoids situations where Tasklets need to wait for data at providers. When the number of idle resources for one data file falls below the buffer threshold, the scheduler distributes new replicas.

Figure 6.11 shows a clear trend when increasing the buffer from 2 to 32 idle resources. On the one hand, the mean task completion time is decreased, which is a consequence of the reduced data queuing time. On the other hand, increasing the buffer leads to more data transfer as the scheduler distributes more copies of each data file. As the configuration of the buffer results in a tradeoff between task completion time and data transfer, we argue that the optimal choice of n depends on the subjective preference of each system administrator. For the remaining experiments, we set the buffer size to 4. In terms of task placement, the hybrid strategy outperforms the random and performance-aware strategy as it delivers faster results while avoiding data transfer. The data-aware strategy also achieves low execution times and results in less data transfer than the hybrid strategy. Thus, the choice of the optimal strategy depends on personal preferences.

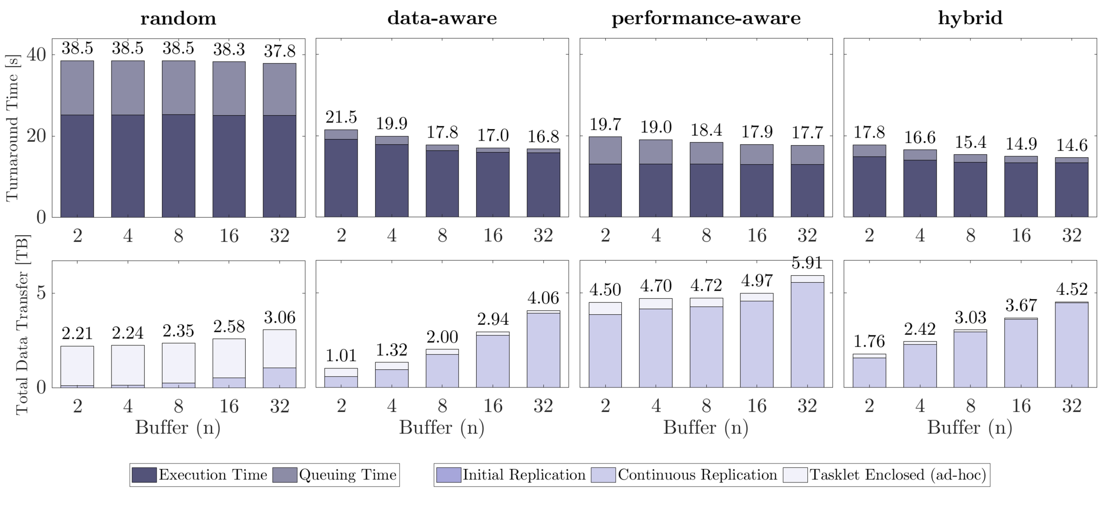


Figure 6.11.: **Experiment 2**: Comparison of average Tasklet completion times and total data transfer overhead for different buffer values in proactive continuous replication, depending on the task placement strategy (random, data-aware, performance-aware, or hybrid). Initial replication is disabled.

Experiment 3 (Constant Initial Replication): So far, we have not applied initial replication, which distributes replicas whenever a new version of a data file becomes available. Without initial replication, for each new version, the data allocation has to be established gradually by continuous replication. Initial replication can help to avoid delays in Tasklet execution since providers hold copies even of recently updated data files. In this setup, we evaluate the constant initial replication strategy which distributes n copies of each data file to providers, where n is set by the system administrator.

The results in Figure 6.12 show that increasing the number of replicas n has a positive effect on the task completion time. The more copies are distributed, the lower is the queuing time for Tasklets at the providers. Here, we observe the fastest average task completion time of the whole evaluation (13.2 s for $n = 32$ and hybrid task scheduling). The more important finding here, however, is that the introduction of initial replication does not only accelerate the execution of Tasklets, but also reduces the data transfer. For the hybrid task scheduling strategy, it even holds that a higher n is beneficial for both task completion time and data transfer for $n \leq 8$. Even though initial replication requires some data transfer, this is outweighed by the lower demand of the continuous replication. At $n = 16$, this effect turns around and a higher n leads to a higher demand of data transfer. The data-aware strategy requires less data transfer than the hybrid task scheduling strategy, but does not perform as well in terms of the task completion time, similar to our findings in experiment 2.

Experiment 4 (History-Based Initial Replication): The previous experiment shows that the choice of the number of replicas n has a great impact on the performance of the initial replication strategy. Setting n too small does not leverage the full potential of the strategy in terms of time and data transfer savings. Setting n too high might result in unnecessary data transfer while not further improving the task completion time. Thus, adjusting the constant initial replication parameter to a particular edge computing system is difficult for system administrators since they have to actively manage the tradeoff between task completion times and data transfer. Further, depending on applications or demand, the ideal n varies over time. The suitable number of replicas may also differ between highly demanded data files and less frequently used data files.

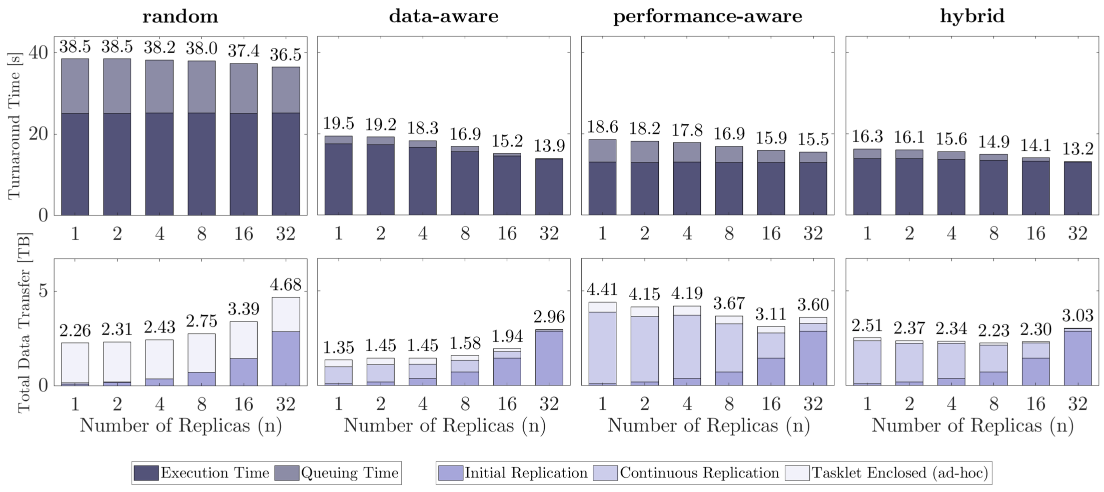


Figure 6.12.: **Experiment 3**: Comparison of average Tasklet completion times and total data transfer overhead for constant initial replication with different predefined numbers of replicas, depending on the task placement strategy (random, data-aware, performance-aware, or hybrid). Proactive continuous replication is applied.

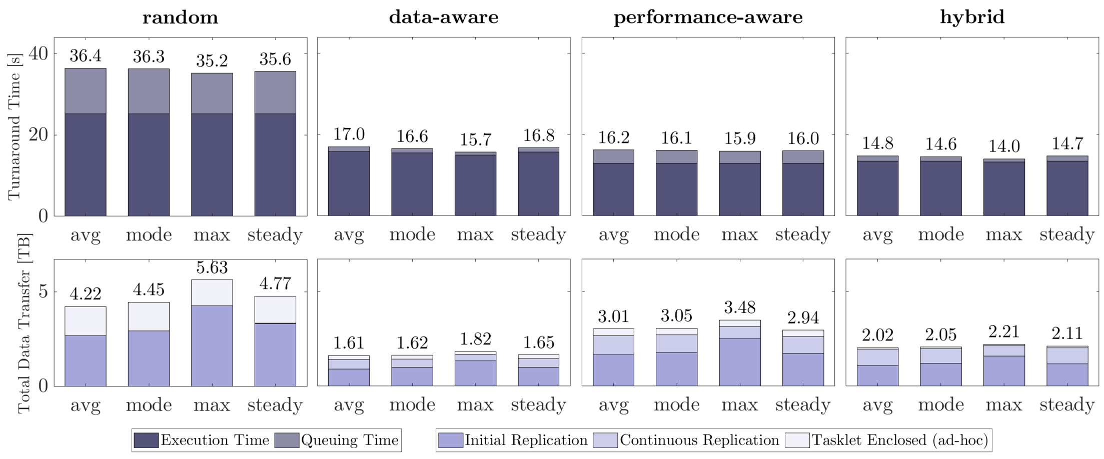


Figure 6.13.: **Experiment 4**: Comparison of average Tasklet completion times and total data transfer overhead for average-, mode-, maximum-based, and steady initial replication, depending on the task placement strategy (random, data-aware, performance-aware, or hybrid). Proactive continuous replication is applied.

To address these challenges, *DataVinci* offers history-based initial replication strategies (average, mode, max, steady). The strategies determine n dynamically based on the most recent usage statistics of each individual data file. The results in Figure 6.13 show that history-based initial replication performs equal or better than constant initial replication. The four strategies perform comparably well and only show minor tradeoffs between task completion time and data transfer. Thus, the results demonstrate that no manual configuration for the initial replication strategy is required and that *DataVinci* can adapt to varying load.

Overall, the four experiments show that proactive data placement and suitable task placement with *DataVinci* considerably reduce task completion times compared to a naive scheduler while keeping the amount of transferred data constant. For instance, using history-based initial replication, proactive continuous replication, and hybrid task placement reduces the average Tasklet completion time from 38.5 s to 14.0 s, which constitutes a relative performance gain of 63.6 %.

6.5.3. Discussion

In the evaluation, we have demonstrated the benefits of proactive data placement with *DataVinci* in the Tasklet system. In the pilot study, we showed the practical feasibility of the approach and the effectiveness of context-aware replication for the placement of new data files. In the subsequent simulation-based study, we assessed *DataVinci* on a larger scale. This study reveals the benefits of proactive continuous replication and the history-based replication strategies for new data versions. Therefore, we conclude that — with the *DataVinci* QoC mechanism — the Tasklet system is able to achieve fast task completion times even for data-intensive applications, which answers research question 2. In the following, we briefly discuss the impact of the results on data-intensive applications.

Is proactive replication always the best strategy? This depends on the goal of the respective use case (task completion time vs. data transfer overhead). In combination with data-aware, performance-aware, or hybrid task scheduling, proactive replication consistently outperforms the other replication strategies in terms of completion times. However, proactive replication leads to more data overhead. In combination with history-based initial replication, it leads to fast executions and a data transfer that is comparable to a naive scheduler.

How consistent are the results? We argue that a large number of applications benefits from *DataVinci*. We have evaluated each of the 10 applications individually for each of the 25 system loads in the large-scale simulation. The applications vary in data size, data version update intervals, degree of parallelization, task complexity, and task frequency (cf. Table 6.4). The load in the system fluctuates heavily between 20 and 140 Tasklets per minute. Nevertheless, the combination of proactive data placement and hybrid task placement performed best among all strategies in more than 75 % of the 250 cases and in 90 % of the cases it performs within 5 % of the best strategy. Thus, we argue that the choice of the optimal strategy depends neither on the application nor on the execution environment.

Is the approach fair? We performed an analysis over all 25 consumers in the simulation for each of the 25 different system loads and compared the average throughput of the providers, which were assigned to each consumer for their tasks. A high deviation in this metric would indicate an unfair system. We observe that 97 % are within a 6 % range. Thus, the strategy can be considered highly fair.

How complex is the configuration of the parameters? As far as context-aware replication is concerned, the choice of the weights in Equations 6.2 and 6.3 is crucial for the effectiveness of the approach. We acknowledge that it is in general difficult to determine proper weights for complex utility functions. In the pilot study, we did not perform any parameter tuning or optimization but used weights of either "0" or "1". Thanks to the results of the pilot study, we therefore argue that system administrators should be able to select proper weights without exceptional effort. In the large-scale simulation, we were able to show that proactive data replication in combination with hybrid task scheduling outperforms other strategies if it is configured properly. Two parameters can be changed. First, the number of replicas is configurable if constant initial replication is used (Experiment 3). However, this parameter becomes obsolete since history-based initial replication (Experiment 4) performs as well as constant initial replication but does not need to be configured. Second, the buffer size for proactive replication is also configurable (Experiment 2). Figure 6.11 shows the tradeoff between low data transfer overhead (for small buffer sizes) and a fast task completion (for large buffer sizes). Thus, the buffer size may either be selected by personal preferences or may be determined self-adaptively depending on the load [160].

6.5.4. Threats to Validity

This chapter presents an extensive evaluation of *DataVinci* in two studies. The pilot study uses emulated Tasklets instead of the Tasklet Core System. Although we are confident that both versions of the system behave identically, this is a potential threat to validity. The large-scale study suffers from the typical weaknesses of simulation-based experiments such as artificial workload, artificial applications, or potentially inaccurate modeling of application and device behavior. While we are convinced that the results are reliable thanks to the extensive validation procedure described in Section 5.2.4, a large-scale evaluation in a real-world testbed is an important avenue for future work. Such an evaluation should also assess the influence of different system sizes on the results, which was omitted in both the pilot study and the large-scale evaluation.

6.6. Summary

This chapter presents *DataVinci* — a scheduler for edge computing systems that is able to reduce the task completion times of data-intensive applications while maintaining a reasonable data transfer overhead. *DataVinci* can be used as a QoC mechanism in the Tasklet system to achieve the *Speed* QoC goal. *DataVinci* determines the number of data replicas in the system, places them proactively on providers, and schedules tasks accordingly. For the replication of new data files, *DataVinci* offers a context-aware replication strategy that approximates the ideal number of replicas. *DataVinci* further includes replication strategies for new data versions that analyze the usage patterns of the data file in the past. Additionally, it proactively adjusts the data placement to avoid bottlenecks even in peak times. Eventually, *DataVinci*'s task placement strategies exploit the data placement to minimize task completion times. We evaluate the performance in a (i) pilot study in a real-world testbed and a (ii) large-scale simulation. Both studies show that *DataVinci* considerably reduces task completion times compared to scheduling approaches without data focus while keeping the data transfer overhead constant.

7. DecArt

Prominent projects such as *SETI@home* [8–10] or *Folding@home* [11] have shaped the common perception of computation offloading as a technology that is particularly helpful for long-running tasks, often from a scientific background. In this chapter, we show that computation offloading in edge computing is also suitable for interactive applications. While using such user-facing applications, response times that exceed one second are usually considered as unacceptable [97–99]. In their groundbreaking work, Ousterhout *et al.* motivate the need for low-latency scheduling for sub-second tasks in clusters [78]. The authors develop an approach to schedule a large number of tasks with minimal delay that enables the deployment of user-facing services on clusters. The proposed solution achieves a significantly higher throughput than previous approaches by applying decentralized, randomized sampling.

Today, eight years later — with the advent of interactive and computationally intensive mobile applications — the demand for low-latency scheduling is still present. The requirements of these applications, which run on mobile or wearable devices, often exceed the local processing capability. Fine-grained computation offloading to remote providers can help to mitigate this bottleneck [161]. In this chapter, we focus on jobs with comparably short soft-real time constraints of several hundreds of milliseconds, such as face detection and recognition [71, 95] and natural language processing [77, 96]. Even for these jobs with rather low complexity, a remote execution in the edge is preferable to a local execution since (i) the job can be split up into multiple independent parallel tasks, (ii) the remote devices have a substantially higher processing speed, and/or (iii) the local device might be busy running other processes.

Low-latency scheduling in edge computing faces three new challenges compared to cloud or cluster computing. First, devices are heterogeneous in their nature which results in different computing capabilities. Thus, the computation time varies depending on the processing power of each individual device. Second, devices are unreliable and randomly join and leave the system. In contrast to resources in cloud or cluster computing, edge devices are not administered centrally but might be shut down by individual users. As a result, these devices have a limited availability and reliability, might not be reachable by other devices, and might drop the computation of a task at any time. Third, the devices are geographically distributed which results in considerable communication latencies of 100 milliseconds and more [162]. The characteristics of edge computing environments place a high demand on task schedulers that select suitable providers for task execution in user-facing applications.

Previous approaches to schedulers for grid and cluster environments such as the one by Ousterhout *et al.* [78] apply, e.g., random probing based on the power of two choices technique [163]. While this solution works well in environments where the communication delay is one or multiple magnitudes lower than the task execution time, probing is not applicable to low-latency edge computing where network latencies are in the same order as execution times. Other approaches use task serialization [164] and finish time estimation [165]. They are effective in environments where tasks can be moved across queues of tightly connected compute nodes but are not directly applicable to geographically distributed end-user devices.

In this chapter, we present *DecArt* (short for “decentralized scheduling is an art”) — a novel decentralized low-latency task scheduling approach for edge computing environments. *DecArt* is designed as a QoC mechanism to achieve the *Sub-second deadline* QoC goal in the Tasklet system. It therefore answers research question 3. The basic idea is that consumers do not need to request resources from a central broker but can make independent provider selection decisions. The broker monitors the status of the providers and periodically forwards this information to consumers in form of cache lists. Thus, consumers can perform the provider selection locally. As there is no coordination among the schedulers, however, a tradeoff arises with respect to scheduling collisions and task execution times, i.e., consistently selecting fast providers may well result in frequent collisions with tasks from other consumers, especially under high load. Conversely, load balancing, i.e., evenly distributing the load on the providers, reduces the risk of collisions at the price of task execution times in a heterogeneous environment, which also includes slow providers. *DecArt* includes two novel decentralized provider selection algorithms *Drift* and *Bandit*, which carefully manage this tradeoff. Both algorithms outperform centralized scheduling and basic decentralized algorithms under varying system load. Thus, *DecArt* is the first scheduler that enables low-latency edge computing on end-user devices.

The remainder of this chapter is structured as follows. We first discuss related work in Section 7.1. Section 7.2 summarizes the underlying system model and presents *DecArt*'s design. Section 7.3 describes the provider selection algorithms in more detail. Section 7.4 evaluates *DecArt* in a large-scale simulation. It shows that the scheduler performs within a 9 % range of a hypothetical omniscient scheduler that serves as a benchmark. The chapter is concluded with a brief summary in Section 7.5. This chapter bases on [166][1].

7.1. Related Work

We review related work from two research streams: (i) *decentralized scheduling* and (ii) *low-latency scheduling*. The two research streams do not entirely overlap as decentralized scheduling can be motivated by scalability or fault-tolerance instead of shorter task completion times. Similarly, not all low-latency approaches rely on decentralized scheduling. Table 7.1 summarizes related approaches.

In centralized scheduling, the broker has a global up-to-date view on all resources in the system. Consumers and providers report their context, e.g., piggybacked to heartbeat messages. The simplest way to provide this knowledge to all schedulers in a decentralized setting is *flooding* [169]. Due to poor scalability, the majority of the reviewed approaches, however, organizes the nodes in a certain way to decrease the communication overhead. The most common approach is to create an *overlay structure* on top of the physical network. In [167, 170, 171, 175, 188], consumers have detailed information only about their neighboring nodes in the overlay. These neighbors are the potential providers for computation offloading or can be used to forward tasks to their neighbors as relay nodes. Shehory uses a list of known nodes, which is similar to a neighborhood-based approach [168]. Another option for an overlay is to apply a tree structure [172, 174, 184, 185]. The child-parent-relationships can be used, for instance, to pass tasks onto the

[1]Reference [166] is joint work with J. Edinger, N. Gabrisch, D. Schäfer, C. Becker, and A. Rizk.

		Orga.				**Opti.**				**Edge**			**Ev.**	
Author/*System*	**Year**	**Overlay structure**	**Probing**	**Auction**	**Cache lists**	**Late binding**	**Work stealing**	**Grouping**	**Hop minimization**	**Fluctuation**	**Heterogeneity**	**Sub-second tasks**	**Simulation**	**Real-world testbed**
NWIRE [167]	1999	●												
Shehory [168]	1999	●			●					●	●		●	
Subramani [169]	2002					○	●		●		○		●	
Ogston [170, 171]	2004	●											●	
OurGrid [103]	2006	●				●					●		●	●
Kim [172]	2006	●								●	●		●	
Ranjan [173]	2008	●									●		●	
LATE [165]	2008					●					●			●
Celaya [174]	2011	●						●			●		●	
Dong [175]	2011	●						●		○	●		●	
Omega [176]	2013				●								●	
Sparrow [78]	2013		●			●						●	○	●
CASA [177]	2013			●	●		●			●	●		●	
Apollo [23]	2014				●	●			○			●		●
Jackson [178]	2014	●				●	●	●		●	●		●	
SocialCloud [179]	2014	●							●		●		●	
Hopper [180]	2015		●			●			○			●		●
Borg [24]	2015		●								●	●		●
Hawk [181]	2015		●			●	●		○			●	●	●
Mercury [182]	2015				●		●		○					●
Duan [183]	2017			●						●	●		●	
Organic Grid [184, 185]	2017	●					●				●		●	●
Aral [186, 187]	2019							●		●	●	●	●	
Cicconetti [84]	2021	●			●				●	●	●			○
MILP [188]	2021	●									●		●	
DecArt					●				●	●	●	●	●	

Table 7.1.: Overview of decentralized and low-latency scheduling approaches. (Orga. = Organization, Opti. = Optimization, Ev. = Evaluation) ● fulfilled ○ partially fulfilled

parent node if the child is not able to execute the task itself [172, 174]. Two approaches [172, 178] implement a Content-Addressable Network (CAN) [189], which helps to identify the node that is responsible for the execution of a task. *SocialCloud* [179] forms the overlay according to trust relationships, similar to popular social networks. Nodes that are connected in the social overlay (i.e., friends) are willing to share their computational resources. In general, overlays are useful to reduce the number of nodes about which a decentralized scheduler requires to have detailed context information. This, however, comes at the cost of worse scheduling decisions when considering only a subset of all nodes as providers.

Probing is an alternative to overlays. The idea is to request context information from multiple random providers. The schedulers then offload tasks to the most promising of these providers, e.g, the provider with the shortest task queue. Probing is especially powerful in large clusters, where communication latencies are low. Therefore, it is not surprising that the prominent cluster computing approaches *Sparrow* [78], *Hopper* [180], *Borg* [24], and *Hawk* [181] apply probing. As the communication latencies in edge computing are considerably higher, probing is not applicable for sub-second tasks in such environments.

CASA [177] and the approach by Duan *et al.* [183] allocate tasks to providers with an *auction* mechanism. In [183], for instance, consumers start an auction for a task by informing potential providers. The providers bid for the task, based on their capabilities and context. The consumer eventually chooses a winner and offloads the task. Similar to probing, auctions are too communication-heavy for edge computing applications with strict sub-second deadlines.

Some approaches such as Cicconetti *et al.* [84] and *CASA* [177] rely on *cache lists*, similar to *DecArt*. In this case, a central instance (i.e., the broker) periodically informs the decentralized schedulers about the current context. Unlike in most approaches that rely on overlays, the decentralized schedulers may thus select the most suitable provider among all providers in the system, not just from a subset. The distributed system context — the cache list — may, however, contain outdated entries. We show the advantages of cache lists for our use case in more detail in Section 7.2.2.

Independent from the chosen organization of the nodes, many related approaches perform certain *optimizations* to achieve lower task completion times. Especially in cluster computing — where a high utilization of the providers is desirable — *late binding* is common (cf. [23, 78, 103, 165, 178, 180, 181]). The schedulers offload multiple identical copies of a task to different providers. This happens either for all tasks or just for stragglers, i.e., tasks that run considerably longer than expected. All but one execution are later canceled or removed from the queue, e.g., after the first copy finished or the first copy was successfully scheduled. Although it is proven that late binding mitigates stragglers, it is not applicable in edge computing environments due to two reasons. First, a high utilization is not beneficial on user-owned devices. Second, communication latencies complicate a timely killing of duplicate tasks. Another potential optimization is *work stealing*. Providers that finished all of their tasks "steal" tasks from slower providers with a large task queue. Popular variations of work stealing are queue balancing or load shedding [182]. In edge computing environments with sub-second tasks, queue sizes vary rapidly and communication latencies are high. Therefore, work stealing is not applicable. *Grouping* of providers helps to reduce the complexity of scheduling. The scheduler only has to choose among groups of providers, not single providers (cf. [186]). Intra-group scheduling may, e.g., happen randomly. Grouping is also helpful to reduce the amount of context information that has to be distributed to each provider. Celaya and Arronategui [174], for instance, aggregate the context information about the subtree at each parent of their tree overlay. In *DecArt*, we apply *hop minimization*, i.e., a conscious reduction of all communication required for the scheduling decision. This is a beneficial optimization in scenarios where latencies contribute considerably to the completion time of a task. Only a few related approaches [84, 169, 179] also minimize hops.

We design *DecArt* as a decentralized scheduling approach for edge computing environments. Three challenges arise in particular: (i) *fluctuation*, (ii) *heterogeneity*, and (iii) *sub-second tasks*. While comparably many approaches from the literature consider heterogeneity in the scheduling process — for example by preferring fast providers — fluctuation is often neglected. Especially in decentralized scheduling, providers joining and leaving the system influence the scheduling effectiveness, as, e.g., cache list information may be outdated.

We observe a dominance of simulations as far as the evaluation of approaches is concerned. Most of the experiments in real-world testbeds happen in the context of cluster computing. The experiments assessing cluster schedulers such as *Apollo* [23], *Hawk* [181], or *Mercury* [182] therefore neglect the typical characteristics of edge environments, but are notwithstanding impressive. Cicconetti *et al.* [84] emulate an edge network. We argue that large-scale simulations that build upon real-world measurements of latencies or computing power are an effective instrument to evaluate decentralized scheduling approaches for edge computing, as they combine the best of two worlds: (i) they are easily scalable to a large number of providers and (ii) they reflect realistic behavior of edge networks and devices.

We conclude from the literature review that decentralized scheduling and low-latency scheduling are prominent research areas in cluster, grid, and edge computing. In this chapter, we propose *DecArt*, a decentralized scheduling approach that relies on cache lists and hop minimization. In addition, it is specifically designed for edge computing networks. *DecArt* is most related to the approach by Cicconetti *et al.* [84]. This approach, however, focuses less on collision prevention for sub-second tasks, which is one of the core contributions of *DecArt*.

7.2. Design

This section first summarizes the underlying system model. Based on this system model, we propose *DecArt* and outline its fundamental idea: decentralized scheduling with cache lists.

7.2.1. System Model

When an application creates a task, the consumer selects a provider for execution. The consumer sends the task to the selected provider, which executes it and returns the result to the consumer. Thus, the task completion time T amounts to:

$$T = S + Q + U, \tag{7.1}$$

with the scheduling time S, the computation time Q, and the result handling time U, which is required to send the result to the consumer. The scheduling time S is composed of the broker request time B and the task transfer time V. Since user-owned resources are unreliable, task failures may happen during computation. In this case, the consumer detects the failure through a timeout and offloads the task to another provider. The completion time for a particular task that experiences n task failures is increased by the scheduling times S_i with $i \in \{1, ..., n\}$ and the computation times Q'_i that passed in vain before the ith failure as well as the detection times D_i that the consumer requires to detect that the task has failed via implicit or explicit feedback:

$$T = S + Q + U + \sum_{i=1}^{n} (S_i + Q'_i + D_i) \tag{7.2}$$

Resource Management

In the Tasklet system, the central broker keeps track of all available devices. Providers register at the broker and share information such as their number of resources and task throughput rate. This rate is statically benchmarked and serves as an approximation for the throughput of the provider. As providers periodically send heartbeats, the broker recognizes a leave within a few seconds. Consumers register at the broker to receive information about available computing resources.

Interactive Applications

In this chapter, we focus on user-facing applications that require a sub-second job completion time to ensure a smooth user experience. Example applications are face detection and recognition [71, 95], natural language processing [77, 190], game AI [71, 95, 96], speech recognition [71, 77], document preparation [77, 191], ray tracing [192], and object recognition [193]. These applications have in common that they do not have strict real-time requirements but depend on timely task executions to contribute to a positive user experience. Response times that exceed one second are generally considered unacceptable and disrupt the feeling of a smooth application flow [97–99]. In our system, each user request issues one *job* that is split up into n parallel *tasks*, i.e. Tasklets, where the longest-running task determines the overall job completion time. Thus, an efficient scheduler does not only minimize task completion times but also mitigates stragglers [194].

Status Quo in the Tasklet System: Centralized Task Scheduling

As the central broker has global knowledge about all entities in the Tasklet system, it can make well-informed scheduling decisions. We use three centralized provider selection algorithms — *Random*, *Greedy*, and *Proportional* — as baselines for our decentralized scheduling approach. Random centralized provider selection allocates a task to a uniform randomly chosen provider from the set of providers that are currently idle. Greedy centralized provider selection always selects the fastest idle provider and thus minimizes the computation time Q. In proportional centralized provider selection, the probability to select a certain provider is proportional to its benchmark throughput. To inform the broker about their currently idle resources, providers send a notification to the broker upon finishing a task execution. Before a consumer offloads tasks of a job, it sends a resource request to the broker, which selects suitable providers for execution and returns this information to the consumer. Figure 7.1 (left) shows the task offloading procedure with a central scheduler. Despite the benefits of the broker, this centralized solution for scheduling and resource management has several downsides. Central entities constitute a performance bottleneck and single point of failure. Further, each broker downtime results in a complete standstill of the offloading system. Finally, communication latencies impact the task completion time as the broker request time B sums up to $2\overline{\delta}$ given an average communication delay between consumers and broker of $\overline{\delta}$. To address these downsides of the centralized scheduler, next, we suggest *DecArt* — a decentralized task scheduling approach for edge computing.

7.2.2. *DecArt*: Decentralized Scheduling with Cache Lists

DecArt is a decentralized scheduler that is located at each consumer. It makes task allocation decisions without prior negotiation with a central entity. This mitigates the single point of failure problem, reduces the workload for the broker, and eliminates the round-trip time ($2\overline{\delta}$) that is required for the resource request ($B = 0$). However, the decentralized scheduler needs to have information about the available resources in the system that can be highly volatile. Keeping this information up to date requires a considerable overhead not only for each consumer but also for the providers. Therefore, we suggest a scheduling model where the resource management is performed by the central broker while scheduling decisions

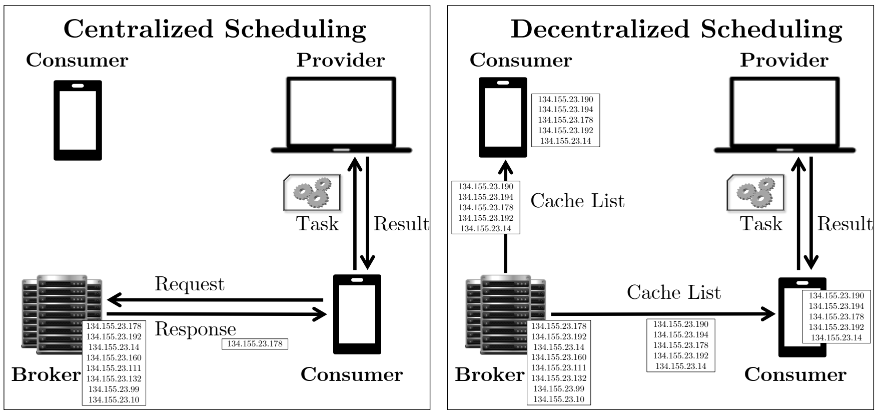


Figure 7.1.: In centralized scheduling, consumers request resources at the broker before offloading a task. In decentralized scheduling, the broker proactively distributes cache lists with active providers. Consumers schedule from these lists by themselves.

are made locally at the consumer. To share information about the current system state, the broker sends a list of all available providers to each consumer. For each task allocation, the consumers select one provider from these cached resource lists (or *cache lists* for short). As the cache lists become stale, the broker sends updates to the consumers periodically. The cache list distribution runs in parallel to the scheduling and, thus, is not added to the scheduling time. In contrast to centralized scheduling, a system that runs *DecArt* remains operational if the broker is temporarily unavailable since stale cache lists are still usable. Figure 7.1 (right) shows the approach.

The decentralized schedulers do not monitor the current load on each provider. This leads to collisions when a single provider receives more tasks than it can process. In this case, this provider rejects the excess tasks and the consumers need to reschedule. This results in a penalty of $2\overline{\delta}$ (round-trip time between provider and consumer). For each rejection k out of total $r \geq 0$ rejections, the scheduling time is increased by an additional task transfer time V_k, and the rejection time R_k, i.e., the time the negative acknowledgment of the provider takes to arrive at the consumer. Rejections may also occur when providers have left the system and cannot be reached anymore. In both cases, the rejection time R_k amounts to a full round-trip time. In total, the scheduling time S in our decentralized scheduling approach thus amounts to:

$$S = \sum_{k=1}^{r}(B_k + V_k + R_k) + B + V \tag{7.3}$$

Equations 7.2 and 7.3 show the effect of decentralized provider selection on an individual task. The task completion time T in Equation 7.2 increases through an increased S, i.e., with the amount of scheduling failures (rejections) r as given in Equation 7.3 and decreases when a fast provider is selected (decreased Q). To minimize the task completion time T, a decentralized scheduler thus needs to select the fastest provider that is currently available. However, as there is no coordination among the consumers, the decentralized schedulers face a tradeoff between selecting a fast provider and risking a collision as other consumers also tend to pick a fast provider.

Figure 7.2 visualizes this tradeoff. In Scenario 1, consumers send most tasks to fast providers, which minimizes the computation time Q but results in multiple rejections r due to congested providers. In Scenario 3, consumers uniformly distribute their tasks, which reduces the risk of collisions r but leads to long-running tasks on slow providers. The target distribution of tasks in Scenario 2 harmonizes the two goals. Achieving this distribution in a decentralized setting is non-trivial. *DecArt* therefore includes two novel algorithms for provider selection to approximate the target distribution.

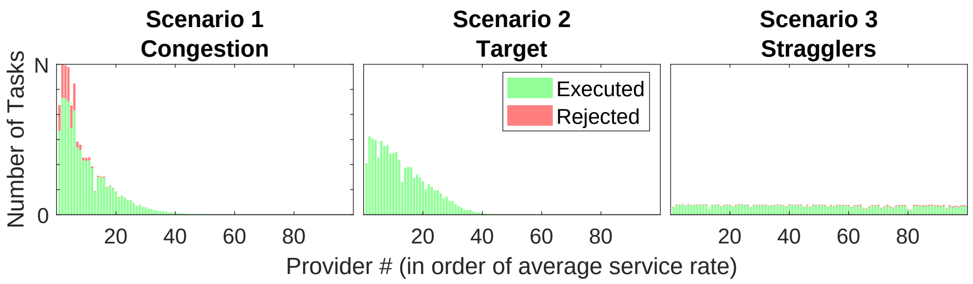


Figure 7.2.: Consumers try to minimize the task completion time T by selecting fast providers and at the same time avoiding collisions. In Scenario 1, consumers schedule most tasks to fast providers which leads to congestion and a high number of rejections. In Scenario 3, tasks are randomly distributed, which results in long execution times on slow providers. Scenario 2 shows an ideal distribution that always selects the fastest available provider. Approximating this target distribution is the goal of *DecArt*.

7.3. Decentralized Provider Selection Algorithms

Consumers periodically receive cache lists from the broker. These cache lists contain currently active providers that offer their computational resources. Based on this choice set, the consumer allocates tasks to providers. The consumer is only able to exploit its local knowledge for the provider selection and does not know about allocated tasks from other consumers to any of the providers. The consumer deletes a provider from its cache list when the provider turns out to be unreachable, i.e, when it has left the system. In this section, we present three basic provider selection algorithms (*Random*, *Greedy*, and *Proportional*) and propose two novel algorithms for decentralized provider selection, namely *Drift* and *Bandit*. Table 7.2 offers an overview of *DecArt*'s algorithms.

Algorithm	**Description**
Greedy	Selection of the fastest provider in the cache list.
Random	Uniform random choice from all providers in cache list.
Proportional	Probability of selecting a provider is proportional to its throughput.
Drift	Uniform random choice from providers in an adaptive scheduling window of varying size.
Bandit	Provider selection from one of multiple disjoint buckets; Bucket selection probability depends on the task throughput distribution.

Table 7.2.: *DecArt*'s decentralized provider selection algorithms.

7.3.1. Basic Provider Selection Algorithms

We introduce *Greedy*, *Random*, and *Proportional* provider selection as basic algorithms. *Greedy* provider selection chooses the fastest provider from the cache list, i.e., the provider with the highest task throughput, for execution. The consumer allocates the task to the next fastest provider if it was rejected previously. *Random* scheduling allocates a task to a uniform randomly chosen provider from the cache list. *Proportional* provider selection uses probabilities. The probability of a provider to be chosen for execution is equal to its relative performance. If a particular provider is expected to finish a task twice as fast as another provider, the probability of selecting this provider is twice as high as well.

7.3.2. Drift Provider Selection Algorithm

In essence, the *Drift* selection algorithm uses a window that contains the w fastest providers taken from the top of the provider list that is sorted by throughput. Only providers inside the window are potential offloading targets whereas all providers outside of the window are per se excluded from the selection. The algorithm selects providers uniformly at random from this window. Each consumer maintains its own scheduling window that might differ from other consumers' windows.

A simple version of the *Drift* algorithm might use a fixed window size w. For example, the ten fastest providers might always be the choice set for task allocation. However, in edge computing systems, context variables such as latency, number of providers, or overall system load change continuously. Therefore, we propose an adaptive approach to change the window size w at runtime. This adaptive approach can be parameterized using the scheduling history and two threshold parameters referred to as η_i and η_r, where the subscripts i, r denote *increasing* or *reducing* the window size w, respectively.

Each consumer maintains a scheduling history of size h containing its last h task requests in addition to whether each task was successfully scheduled or rejected (i.e., it was a scheduling fail). The consumer increases the scheduling window size w if the last h task requests contain more than η_i rejections and it decreases the window size w if the last h task requests contain less or equal than η_r rejects. The rationale behind this dynamics is based on increasing the chance of successful task scheduling at the expense of a higher expected task execution time (through inflating w). Reducing w at low numbers of scheduling failures provides a *drift* towards choosing the fastest provider, i.e., minimizing the expected task execution time. Figure 7.3 illustrates the approach.

7.3.3. Bandit Provider Selection Algorithm

The *Bandit* provider selection algorithm separates the providers from the cache list into disjoint buckets. It is inspired by the *multi-armed stochastic bandit* model [195]. System administrators can parametrize the algorithm with the number of buckets N and choose the *bucket creation heuristic* that determines how providers are allocated to buckets. We propose and evaluate *leapfrogged*

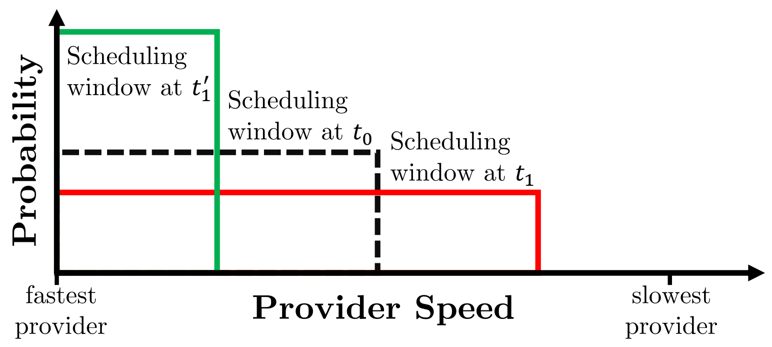

Probability
Scheduling window at t_1'
Scheduling window at t_0
Scheduling window at t_1
fastest provider
Provider Speed
slowest provider

Figure 7.3.: *Drift* provider selection chooses providers for execution uniformly from the scheduling window. The scheduling window increases adaptively in width if the last h scheduling attempts have lead to more than η_i rejections. It decreases in width for less or equal than η_r rejections in the last h attempts.

and *performance-aware* bucket creation. In both, the consumer maintains a list of all providers sorted by throughput. Leapfrogged bucket creation then uses the leapfrogged series of providers from this list, i.e., bucket #1 contains the providers $\{1, N+1, 2N+1, \dots\}$ while bucket #2 contains the providers $\{2, N+2, 2N+2, \dots\}$. Thus, the computing power of the whole system is approximately evenly distributed among the buckets. Performance-aware bucket creation uses the original series of providers sorted by throughput, i.e., bucket #1 contains the providers $\{1, \dots, N\}$ while bucket #2 contains the providers $\{N+1, \dots, 2N\}$. Some buckets therefore contain considerably more powerful resources than other buckets.

To allocate a task, the *Bandit* algorithm first selects a bucket and then selects a provider out of this bucket for task execution. The choice of a bucket is based on a bucket-specific performance metric that results from past task executions in this bucket. When a consumer receives a task result, it uses the task completion time to update the performance value of the respective bucket. Once the performance value of the current bucket falls below the performance of another bucket, the algorithm starts using the other bucket for scheduling. Figure 7.4 depicts how the performance values of the buckets vary over time in an exemplary run from our evaluation. After selecting the bucket, the algorithm chooses among the providers in this bucket proportionally similar to *Proportional* selection.

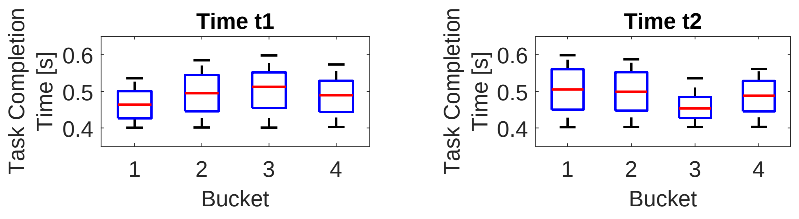


Figure 7.4.: The *Bandit* algorithm divides providers into buckets (here four). Each bucket's performance value (y-axis) is determined based on the task completion times in this bucket in the past. The bucket with the best performance value is chosen for scheduling. At t_1, the scheduler thus selects a provider from bucket 1. At t_2, bucket 3 has a better performance value and is therefore used for task execution this time.

Bandit provider selection has to balance exploration and exploitation. On the one hand, the algorithm should always select providers from the most promising bucket. On the other hand, it has to update the performance values of less attractive buckets to have an up-to-date basis for decision-making. Changes in the environment such as decreasing offloading demand can make a change of the bucket beneficial, even if the performance value of the current bucket is improving. Thus, a certain amount of exploration tasks is scheduled on providers of buckets that do not have the highest performance value.

7.4. Evaluation

In this section, we evaluate *DecArt* in a large-scale experiment. For better controllability, scalability, and reproducibility, we use the Tasklet Simulator. Section 7.4.1 describes the experimental setup. Section 7.4.2 offers an extensive overview of the results. The evaluation concludes with a discussion of the results in Section 7.4.3 and potential threats to validity in Section 7.4.4.

7.4.1. Experimental Setup

The OMNeT++ implementation simulates 200 providers, 100 consumers, and one broker. This system size resembles an edge computing scenario in an office building, a university department, or a public place with adjoining houses. Of the 200 provider nodes, 59 behave like desktop PCs, 53 like laptops, and 88 like smartphones in terms of churn and computing power. Each time providers enter the system, they draw their residence time from a normal distribution with a mean of 420 min and a standard deviation of 150 min for desktop PCs, 240 min/162 min for laptops, and 60 min/60 min for smartphones. Devices that leave the system cancel all running tasks and reenter the system after a randomly chosen time interval. All providers are able to execute four tasks in parallel. The broker distributes cache lists every 30 s if decentralized scheduling is applied. The network latency varies between 100 and 200 ms including a TCP handshake.

Each consumer node runs one application chosen from a pool of ten applications available in total. Consumers do not submit jobs consistently but only during active periods (approximately 10 % of the total time). This resembles a typical application usage on mobile devices. Each algorithm configuration is simulated for 60 min. We run each configuration in five load scenarios with job intervals in active periods of $\{0.9, 0.6, 0.4, 0.3, 0.2\}$ s on average. Each scenario is simulated 20 times, with a random walk that varies the system load. The resulting loads are constant across algorithm configurations, but change for each of the 20 repetitions.

7.4.2. Results

In our experiments, we compare centralized scheduling in the Tasklet system to decentralized scheduling with *DecArt*. For the central scheduler, we use *Random* (C-RND) and *Greedy* (C-GRD) provider selection. With *DecArt*, we test the three basic algorithms *Random* (D-RND), *Proportional* (D-PRP), and *Greedy* (D-GRD) as well as the two advanced algorithms *Drift* (D-DFT) and *Bandit* (D-BND). As a benchmark, we have further implemented an omniscient decentralized scheduler (D-OMNI) that has a global view on all providers in the system and knows immediately when resources become available. Thus, a consumer can directly send the next task to the fastest available provider. In reality, this hypothetical scheduler is not feasible due to communication latencies but, here, it serves as the benchmark. We perform four experiments. In experiment 1, we conduct a parameter study of the two advanced algorithms *Drift* (D-DFT) and *Bandit* (D-BND). In experiment 2 and 3, we assess *DecArt* with regards to task completion times (experiment 2) and job completion times (experiment 3). In experiment 4, we evaluate the effect of queuing on *DecArt*'s performance. Table 7.3 provides an overview of the abbreviations that are used throughout the experiments.

Abbr.	**Explanation**
C-RND	Centralized scheduling, *Random* provider selection
C-GRD	Centralized scheduling, *Greedy* provider selection
D-GRD	Decentralized scheduling, *Greedy* provider selection
D-RND	Decentralized scheduling, *Random* provider selection
D-PRP	Decentralized scheduling, *Proportional* provider selection
D-DFT(-n)	Decentralized *Drift* scheduling (max. queue size = n)
D-BND(-n)	Decentralized *Bandit* scheduling (max. queue size = n)
D-OMNI	An omniscient, decentralized scheduler as baseline

Table 7.3.: Abbreviations used in the evaluation.

Experiment 1 (Drift and Bandit Parameter Study)

We conduct an extensive parameter study to understand the effect of the parameters on the performance of the two algorithms. Both advanced algorithms depend on a set of parameters (cf. Section 7.3) that can be used to configure *DecArt* for each individual environment. We first analyze the *Drift* algorithm which is adjustable with the number of rejections that reduces the scheduling window size η_r, the number of rejections that increases the scheduling window size η_i, and the number of previous tasks in the scheduling history h. In this experiment, we set the history length to $h = 100$. We observed similar patterns for other history lengths in supplementary tests ($h = 10, 20, 50$). Figure 7.5 depicts job completion times for different η_r and η_i in the five load scenarios. Similar patterns are visible as far as deadline misses are concerned. In general, job completion times are reduced with cautious window reduction, i.e., only when the number of rejections is very low the target window is reduced. Further, we observe that the optimal η_i increases with the load. This happens because a small η_i quickly extends the window w which results in a larger task execution time.

We now perform a parameter study for the *Bandit* provider selection algorithm. We vary two parameters: the bucket creation heuristic (*leapfrogged* vs. *performance-aware*) and the bucket size. Figure 7.6 provides an overview of the job completion times for both heuristics and different bucket sizes under different loads. We first observe that the bucket size has a smaller influence for leapfrogged buckets. In general, leapfrogged bucket creation increases the mean and variance of job

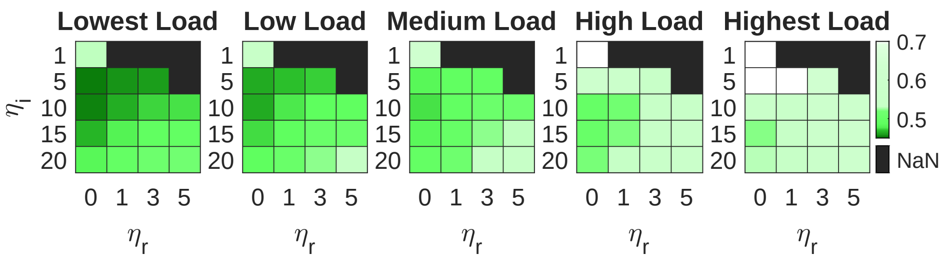


Figure 7.5.: Analysis of decentralized *Drift* scheduling. We study the impact of η_i and η_r on the average job completion time. Recall that $\eta_i > \eta_r$. For a fixed scheduling history size of $h = 100$, the scheduling window size is reduced if the algorithm observes less rejections than η_r. More rejections than η_i result in an increasing window size. We observe that, in general, low values for η_r, i.e., cautious window reduction, i.e., reduction only when the number of rejections is very low, yields lower job completion times. Further, we observe a shift in η_i for increasing load towards larger values which is because a small η_i quickly extends the window w and slower providers are selected.

completion times. This is due to the fact that all buckets also contain slow providers which are selected when a consumer sends out multiple tasks. Performance-aware bucket creation outperforms leapfrogged bucket creation in terms of job completion times. In the highest load scenario, performance-aware (480 ms, with bucket size 5) is 46 % faster than leapfrogged bucket creation (768 ms, with bucket size 20). In contrast to leapfrogged bucket creation, performance-aware bucket creation is sensitive to the bucket size parameter. Across all load scenarios, there is a tendency that small buckets perform better than larger bucket sizes. In the following, we use the optimal parameters from this study (*Drift*: $h = 100$, $\eta_r = 0$, $\eta_i = 10$, *Bandit*: performance-aware bucket creation, $N = 5$).

Experiment 2 (Task Completion Times)

Figure 7.7 shows the results for all strategies under the five different loads[2]. It displays the mean task completion times and how these times are composed. The most notable result is that centralized scheduling (C-RND and C-GRD) is outperformed by all decentralized algorithms. Under low and medium load, central scheduling takes about 250 ms for the consumer to request a resource from the

[2]More details are available in Table A.1 and Table A.2 in the appendix.

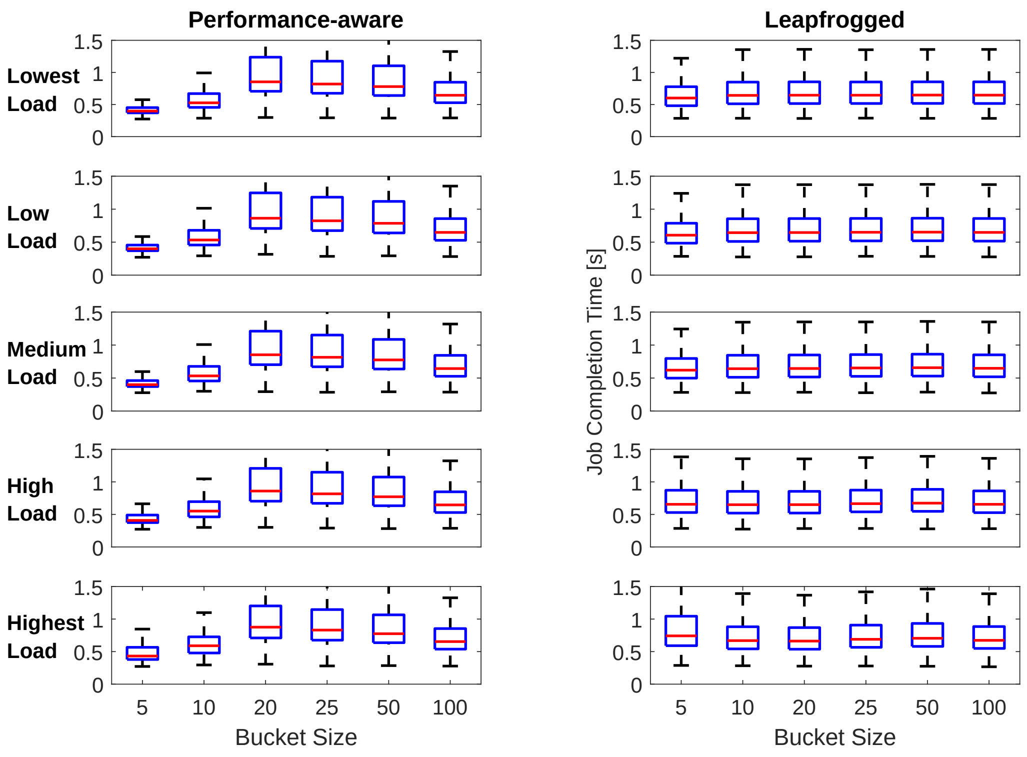


Figure 7.6.: Analysis of bucket creation heuristic (performance-aware vs. leapfrogged) and bucket size of the *Bandit* algorithm in different load scenarios. Performance-aware bucket creation leads to faster job completion times than leapfrogged bucket creation if the bucket size is set to small values. The leapfrogged creation generates buckets with comparable execution speeds.

broker, which resembles the average round-trip time in the system ($2\overline{\delta}$) including a TCP handshake. The time for resource requests increases dramatically under high loads because the broker cannot find idle resources. Instead, it has to wait for providers to send status updates about their available TVMs. This leads to a congested system and task completion times of up to 20 s for central scheduling (exceeds the limits of the plot).

In decentralized scheduling, *DecArt*'s advanced provider selection algorithms (D-DFT and D-BND) outperform the three basic strategies (D-RND, D-PRP, and D-GRD). The greedy algorithm (D-GRD) causes a high number of rejections as it always allocates tasks to the fastest providers which then become congested. The random algorithm (D-RND) suffers from a high execution time because it arbitrarily chooses among slow and fast providers. The proportional algorithm

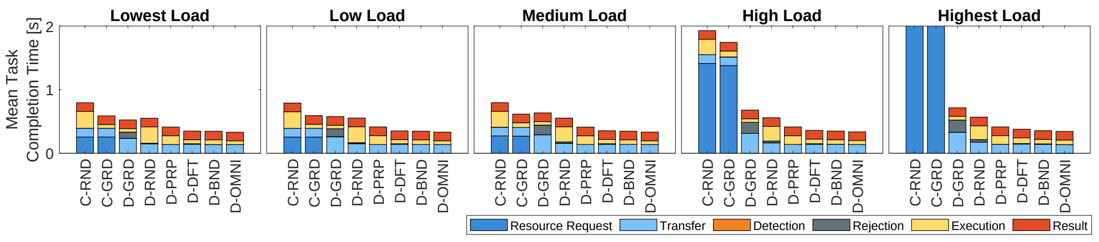


Figure 7.7.: Analysis of the task completion times. Only centralized scheduling requires time for the resource request. Decentralized scheduling algorithms select resource providers from their local cache lists. Our proposed scheduling algorithms (D-DFT & D-BND) successfully avoid congested providers (which leads to rejections) and selecting slow providers (which leads to long execution times). They perform within a 10 % range of an omniscient scheduler (D-OMNI).

(D-PRP) manages the workload quite well but cannot compete with D-DFT and D-BND in terms of execution times. The latter two perform within 5 to 10 % of the omniscient scheduler and produce comparable results under all five loads. These findings indicate that decentralized scheduling can handle low-latency requirements for individual tasks even under high loads.

Experiment 3 (Job Completion Times)

Next, we focus on job completion times. As described in Section 7.4.1, applications issue jobs with 6 to 16 parallel tasks. A job is only completed when all its tasks are finished, which means that the job completion time equals the maximum completion time of its tasks. In practice, job completion times are often more relevant than task completion times, as job completion times are directly visible to the user. This highlights the importance of avoiding stragglers. We consider jobs with a duration of more than 1 s as a deadline miss. Figure 7.8 shows the distribution of job completion times for each algorithm under all five loads. Each plot contains four types of information: First, the histogram shows the distribution of job completion times between 0.2 and 1 s. Second, the bar at the very right indicates the number of deadline misses, i.e., jobs that were completed after 1 s. The percentage in the upper right states the percentage of deadline misses in relation to all completed jobs. Third, the blue line shows the mean of all job completion times. A missing line signals that the mean is greater than 1 s. Fourth, the red dotted line shows the median of all job completion times.

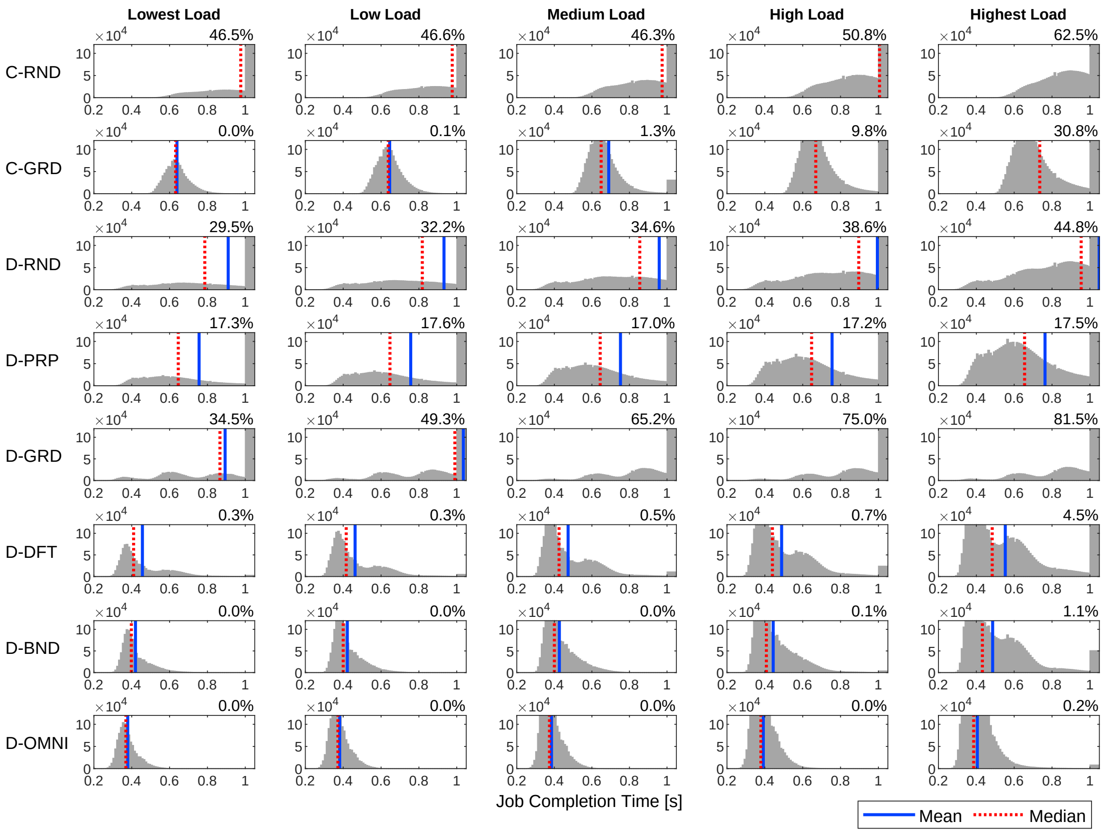


Figure 7.8.: Empirical job completion times for centralized and decentralized scheduling. The histograms show the distribution of the job completion times. The utter right bars count jobs that exceed the 1 s deadline. The percentage of missed deadlines is shown in the upper right corner of each plot. Our two proposed scheduling algorithms (D-DFT and D-BND) have a low mean and median job completion time and minimize the number of missed deadlines.

We observe that each algorithm results in a characteristic shape for the distribution of job completion times. Centralized random scheduling (C-RND) cannot meet the deadlines under any load as it chooses slow providers in many cases. The centralized greedy algorithm (C-GRD) shows job completion times symmetrically clustered around 0.6 s. The higher the load, the more jobs miss the deadline and, eventually, the mean increases to more than 1 s. Decentralized random and proportional (D-RND and D-PRP) manage to keep mean and median job completion times below or close to 1 s. However, their percentage of deadline misses range between 17 and 44.8 % since tasks are scheduled to slow providers even though fast providers are not running at full capacity. Decentralized greedy

(D-GRD) performs worst among all decentralized algorithms with up to 81.5 % deadline misses. In contrast to centralized greedy selection, the decentralized equivalent leads to collisions and re-scheduling due to the missing coordination of the decentralized schedulers, which all behave greedily with their limited, local view. Our proposed decentralized algorithms *Drift* and *Bandit* (D-DFT and D-BND) achieve both goals. They keep mean and median job completion times low and avoid deadline misses. Among the two, D-BND performs better and achieves results within 7 and 20 % of the omniscient algorithm (D-OMNI).

Even though the three basic decentralized algorithms (D-RND, D-PRP, and D-GRD) seem to be able to compete with the *Drift* and *Bandit* algorithms with respect to task completion time, they fail to consistently meet the job deadline of 1 s as they suffer from delayed tasks. In contrast, *Drift* and *Bandit* are robust to stragglers and, thus, enable the execution of responsive applications in edge computing environments with user-controlled devices as resource providers.

Experiment 4 (Queuing)

We observe that even under the highest load, the fast providers are not working to full capacity. Once they have finished the execution of a task, they have to wait for the next one to arrive. In the worst case, this happens just after a provider has rejected a task because it did not have any idle TVMs. Thus, we perform an additional experiment that introduces task queues at the resource providers. Task queues avoid idle times and allow the TVMs to seamlessly execute one task after another. As a downside, a long queue might increase the task completion time as it introduces queuing delays. We experiment with different queue sizes between one and five tasks for both the *Drift* and the *Bandit* algorithm. Providers with full queues reject incoming tasks. Figure 7.9 shows the results for the *Drift* algorithm. The results for the *Bandit* algorithm look similar. To better demonstrate the effect of queuing on the task completion time, we remove transfer and result handling times from the figure as they are similar across all settings.

We observe that a larger queue size increases the queuing time because tasks need to wait until they are processed. However, the execution time decreases with the queue size because fast providers can be used more efficiently. These opposing effects result in decreasing task and job completion times for small queue

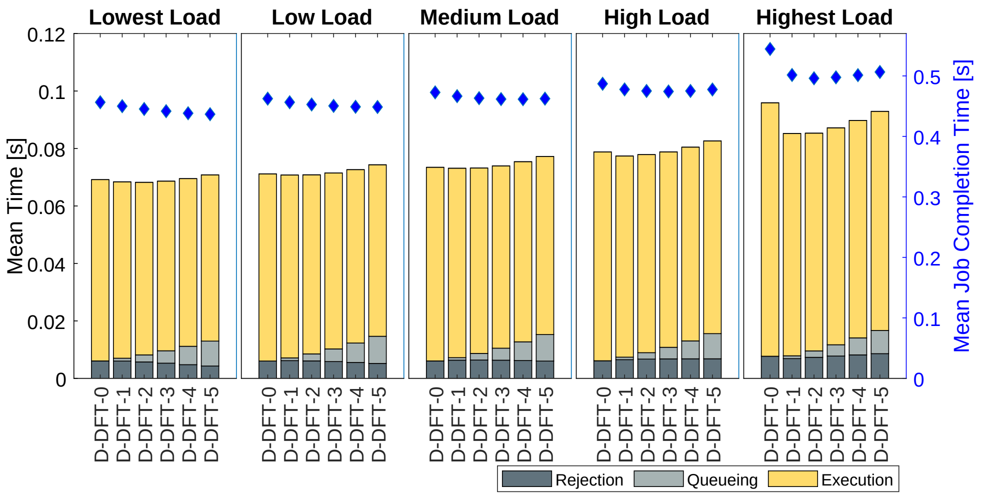


Figure 7.9.: Analysis of maximum queue sizes and their impact on task completion times (stacked bars) and job completion times (blue markers) for *Drift* scheduling. We observe that queuing reduces job completion times, especially in high load scenarios. Increasing the maximum queue size leads to lower execution times at the cost of increasing queuing times. We observe similar effects for *Bandit* scheduling.

sizes. Under high load, this effect reverses for larger queue sizes. The second effect of queuing is a substantial decrease in deadline misses. Whereas the *Drift* algorithm without queuing resulted in 4.51 % of deadline misses under the highest load, increasing the queue size led to 1.38 % misses for D-DFT-1 and 1.77 % for D-DFT-5. For the *Bandit* algorithm these numbers range from 0.23 % (D-BND-1) to 0.02 % (D-BND-5). In summary, queuing does not only reduce the mean job completion time but further eliminates most deadline misses.

7.4.3. Discussion

The empirical evaluation in the previous section shows that *DecArt* is able to manage the tradeoff between selecting fast providers and avoiding collisions. We come to three conclusions: First, *DecArt* decreases job completion times by at least 30 % (D-DFT) and 34 % (D-BND) compared to a central scheduler. Second, the algorithms perform within a range of 17 – 23 % (D-DFT) and 5 – 10 % (D-BND) of a hypothetical omniscient decentralized scheduler. Third, the algorithms eliminate most deadline misses even under very high system load. Thus, in this chapter we show that *DecArt* is the first approach to allow low-latency task offloading in edge environments for user-facing applications. The remainder of this section briefly discusses practical implications of the approach.

Is decentralized scheduling always superior to centralized scheduling? As far as task and job completion times of responsive applications are concerned, we observe that decentralized scheduling is always faster than centralized scheduling with the broker. Nonetheless, the results in Figure 7.8 suggest that the careful choice of the provider selection algorithm is crucial. For instance, the decentralized version of the greedy provider selection performs worse than the centralized equivalent. In addition, decentralized scheduling has the drawback that computationally intensive provider selection algorithms run on the consumer device instead of the broker, which is on average a more powerful resource connected to a constant power supply. While *DecArt*'s scheduling algorithms are of comparably low computational complexity, the algorithm applied in *Voltaire* described in the following chapter shows that centralized scheduling approaches still have their merits.

Is it better to apply the Drift or the Bandit algorithm? In our evaluation, the *Bandit* algorithm outperforms the *Drift* algorithm by a small margin in many metrics. Without queuing, the *Bandit* algorithm leads to 1.1 % deadline misses under the highest load while the *Drift* algorithm cannot prevent that 4.5 % of the deadlines are missed. In addition, the parameter study in Section 7.4.2 reveals that the optimal parameters for the *Bandit* algorithm are the same throughout the study whereas the configuration of the *Drift* algorithm has to be adapted to the load. For a final decision, a long-term comparison of the two algorithms in a real-world testbed or even a real-world deployment is important future work.

What is the effect of the cache list interval on the performance? In all experiments, the broker distributes cache lists every 30 s if decentralized scheduling is applied. Selecting a proper cache list interval is non-trivial as a tradeoff between up-to-dateness of the cache list information and distribution overhead exists. The resource management and cache list distribution itself, however, run asynchronously and have no direct effect on the scheduling times. We experimented with shorter and longer intervals but this did not have a significant effect on the completion times. Each cache list has a size of about 3 kB, which results in about 1 MB traffic for the broker per minute in our experiments.

Considering that the simulated system is comparably large, we argue that it is feasible to maintain up-to-date cache lists with manageable overhead in practice.

Would a context-aware choice of the strategy be a valuable addition? Similar to the evaluation of *DataVinci* in the previous chapter, we observe that an adaptive choice of the scheduling approach (decentralized vs. centralized) and the parametrization of algorithms such as *Drift* or *Bandit* provider selection based on the context is an essential avenue for future work. The QoC concept of the Tasklet system is already a first step, as the application programmer can select the respective QoC goal for each Tasklet separately. This enables the parallel usage of several approaches even in the same application. Nonetheless, self-adaptation remains an open challenge.

7.4.4. Threats to Validity

The evaluation in this chapter bases on a large-scale simulation. Although we validated the simulation carefully as described in Section 5.2.4, potential threats to validity encompass an inaccurate modeling of real-world applications, inaccurate device models, and unrealistic system loads. Furthermore, the evaluation only assesses sub-second tasks. An interplay with long-running tasks in the system could potentially — but unlikely — influence the results. *DecArt* does not consider the communication latency to providers in its decision making so far. In scenarios where the bandwidth or the communication channel to some providers differ considerably, this context dimension may have a notable influence. We compare the performance of *DecArt* with an omniscient scheduler that knows immediately when resources become available. Future work could compare *DecArt* with an ideal scheduler, i.e., a scheduler that — in hindsight — always makes the perfect scheduling decision to minimize completion times.

7.5. Summary

This chapter presents *DecArt* — a decentralized scheduling approach for low-latency computation offloading in edge computing environments. The approach is designed for responsive user-facing applications that require sub-second task completion times. Instead of requesting resources at a central broker before each task execution, consumers schedule from previously distributed cache lists by themselves. As there is no coordination among the consumers, this leads to a tradeoff between selecting a fast provider and risking a collision as other consumers also tend to pick a fast provider. To balance this tradeoff, we propose two advanced provider selection algorithms. We present the *Drift* algorithm where resource consumers maintain and utilize an individual adaptive scheduling window of resource providers and the *Bandit* algorithm that allocates providers to disjoint groups for which resource consumers calculate selection probabilities based on the history of observed task execution times. In an extensive simulation-based study, we show how *DecArt* significantly decreases the task and job completion times by more than 30 % in comparison to the centralized scheduler of the Tasklet system. In addition, *DecArt* performs within a 9 % range of a hypothetical omniscient scheduler that serves as a benchmark.

8. Voltaire

Battery constraints have become a key challenge with the advent of mobile computing and smartphones [7]. In addition to performance gains, computation offloading helps to reduce the energy consumption of mobile devices if the cost of transferring a task and receiving the results is lower than the cost of a local execution. Thus, a remote execution is in general beneficial for computationally intensive tasks with small input and result data. Analogously, less complex tasks with large input and result data are to be executed on the mobile device.

In edge computing systems, the decision whether to execute a task remotely or locally is non-trivial due to uncertainty. First, in contrast to a traditional batch system, the completion time of a task is not known a priori. Second, similar to the completion time, the size of the execution output might vary for each execution. This results in different transmission costs. Both task completion time and result size may even change for different executions of the same source code, as varying parameters and input data have a considerable influence. Third, important context variables such as connection type or bandwidth change frequently and need to be monitored at runtime [196].

Energy-aware offloading has received much attention in research [197]. Most notably, in *MAUI*, Cuervo *et al.* show the effect of different network interfaces on the energy consumption [71]. In *CloneCloud*, Chun *et al.* apply dynamic profiling to create profile trees that explore the impact of input parameters on the energy consumption of the task [70]. In *ThinkAir*, Kosta *et al.* build upon the two previous approaches and add elasticity and scalability of the remote resources [76]. However, none of the approaches focus on reducing the uncertainty in the prediction of the energy consumption from the devices's view. Instead, rather straightforward estimators are applied.

In this chapter, we design a QoC mechanism for the Tasklet system that minimizes the energy consumption of the consumer device, i.e., that achieves the *Energy* QoC goal. We propose *Voltaire* — a centralized scheduler that applies machine learning methods for regression analysis based on crowd-sourced data of past executions of similar tasks. This enables *Voltaire* to accurately predict the complexity and the result size of an upcoming task. Furthermore, *Voltaire* integrates device-specific energy profiles, which model the influence of CPU and network activity on the energy consumption. With additional context knowledge about the input data size of a task and the current bandwidth, *Voltaire* is able to dynamically decide for each Tasklet whether to offload or execute locally. Thus, *Voltaire* improves energy consumption from the perspective of a mobile device rather than a global system perspective for energy efficiency. We deploy *Voltaire* in the real world to extensively evaluate its effectiveness in realistic settings with three user-facing applications. The experiments show that *Voltaire* is able to predict the complexity and the result size of a task precisely. We observe that *Voltaire* reduces the consumer device's energy consumption for task execution by 12.5 % compared to the existing scheduler of the Tasklet system.

In the remainder of this chapter, we first discuss related work in Section 8.1. Section 8.2 describes the underlying system model and *Voltaire*'s fundamental design. A detailed description of the design is then presented in Section 8.3. Before concluding the chapter with a summary in Section 8.5, we extensively evaluate *Voltaire* in a real-world testbed in Section 8.4. This chapter bases on [198][1].

8.1. Related Work

Making energy-aware offloading decisions under uncertainty is a well-known problem in practice and academia. Previous approaches estimate context parameters, profile applications, and monitor network states to find strategies that "*make smartphones last longer*" [71, p.1]. A recent survey of energy-aware approaches in edge computing can be found in [197]. Despite all these valuable achievements, several issues remain unresolved and are, therefore, addressed in this chapter.

Multiple approaches do not consider all relevant context dimensions or make assumptions about them that conflict with real-world applications. Examples can be found in [70], [77], and [199] that consider the result size as static, which is not true for many real-world applications. Further, differences across devices in the

[1]Reference [198] is joint work with J. Edinger, S. Kaupmees, H. Trötsch, C. Krupitzer, and C. Becker.

amount of energy consumed per instruction are often neglected. Previous works differ in their approaches to predict context dimensions such as the complexity of tasks or the required data transfer. Some approaches perform a static code analysis without taking the effect of input parameters into account [200]. Several strategies include the computation of averages or linear models based on past executions [71, 104, 201]. We argue that machine learning can help to determine complex relationships between input parameters and the estimates that go beyond simple dependencies [202, 203]. The calculation whether a number is prime illustrates this phenomenon. The complexity of this task increases with the input n, e.g., 73 is more complex to test than 11. However, each even number is trivial to test as it is divisible by the number 2. While a manual definition of these rules is cumbersome, machine learning can detect these patterns autonomously.

Typically, the offloading decision is made on the mobile device. As this causes additional overhead and thus energy consumption, the decision making process is often kept as lightweight as possible. A remote decision making allows for more complex algorithms [204, 205]. An additional benefit of remote decision making is the number of samples that the scheduler can observe to create execution statistics for an application. This number is not limited to the execution statistics of a single device. A central decision maker can apply crowdsourcing and collect usage data from multiple devices. Only a subset of prior approaches is evaluated in real-world testbeds with actual energy monitoring. This makes the results which are gained through simulations complex to translate to actual energy savings (e.g., [206] and [207]). Most approaches are not integrated into offloading frameworks which raises questions about their generalizability and applicability in the physical world [111, 208, 209]. Furthermore, some of the previous works do not run real applications for the evaluations [210, 211]. We argue that real applications increase the validity of suggested approaches. Table 8.1 summarizes existing works.

We propose *Voltaire* — a QoC mechanism in the Tasklet system that applies machine learning methods on crowdsourced data to make well-informed offloading decisions. In addition, it monitors multiple context dimensions such as device type, bandwidth, and input data size to accurately decide whether to offload a task or to execute it locally. We integrate *Voltaire* into the Tasklet Core System to evaluate its effectiveness with real-world applications in a realistic setting.

		Context				Prediction					Dec.			Eval.		
Author/*System*	**Year**	**Task complexity**	**Data transfer**	**Network**	**Device**	**Static analysis**	**Cost graphs/tables**	**Averages**	**Linear models**	**Machine learning**	**Mobile device**	**Cloud/Broker**	**Crowdsourcing**	**Real-world testbed**	**Offloading framework**	**Real-world applications**
Othman [212]	1998	●		●	●			●			●					●
Rudenko [213]	1998													●		●
Spectra [77]	2002	●	○	●	●				●		●			●	●	●
Xian [208]	2007	●			●				●		●			●		●
MAUI [71]	2010	●	○	●	●			●			●	●		●	●	●
CloneCloud [70]	2011	●									●	●		●	●	●
Giurgiu [214]	2012	●		●	●						●	●		●	●	●
ThinkAir [76]	2012	●	○	●	●						●			●		●
MACS [104]	2012	●	○	○	○	●		●			●			●	●	●
SP Energizer [215]	2013	●	●	●	●					●	●	●		●	●	●
TDM [216]	2013	●	●	●	●		●				●			●		●
Magurawalage [204]	2014	○	○	○	○				●		●	●				
Niu [206]	2014	○				●	●				●					
Ravi [200]	2014				○	●					●			●		●
DREAM [217]	2015	○	○	●	●			●	●		●			●		●
Jade [110,111]	2015	○	○	●	●		●				●			●		●
Saab [209]	2015		○	●	●						●			●		●
EECOF [218]	2015		○	●							●			●		●
Zhou [201]	2015	●	○	●	●			●			●	●		●		●
Wu [207]	2016	●	○	●			●				●					
EECO [210]	2016	●	●	●	●						●					
EA-OSGi [211]	2017	○	○	○	○			●			●					
Karim [202]	2017		○							●	●					
Crutcher [203]	2017	○	●							●	●					
EMCO [75]	2018	○	○	●	○					●	●	●	●	●	●	●
Jadad [205]	2018	●		●	○		●	●				●				
Hao [219]	2018	●	○	●					●			●				
Lyu [220]	2018	●	○	●	○				●			●				
HetNet [221]	2019	○	○	●					●			●				
Nguyen [222]	2019	○	●	●					●							
Xu [223]	2019	●	●	●		●										
Voltaire		●	●	●	●			●		●		●	●	●	●	●

Table 8.1.: Overview of related energy-aware offloading approaches. (Dec. = Decision making, Eval. = Evaluation)
● fulfilled ○ partially fulfilled

8.2. Design

This section first describes the system model that we use for our approach. After that, we outline *Voltaire*'s basic design.

8.2.1. System Model

Voltaire is a scheduler for an environment where a mobile device offloads tasks to remote resource providers to reduce local battery consumption. A consumer runs several applications which issue tasks. Tasks logically consist of a *type*, i.e., underlying source code, and *parameters*. Additionally, tasks may require *input data*, such as images for face recognition, that have to be transferred to the providers. We assume that both consumers and providers run TVMs. Thus, a local execution is possible. In the system, a central broker performs resource management, i.e., decides whether and to which device to offload a task.

8.2.2. Energy-Aware Computation Offloading with *Voltaire*

Voltaire reduces the energy consumption of the consumer device by deciding whether a task is offloaded or not. In contrast to related work that minimizes the total energy consumption of a whole distributed computing system [210, 224], we focus on improving the energy consumption of a battery-constrained consumer device only, since cloud or edge computing environments typically contain offloading targets with constant power supply.

Voltaire suggests to offload a task when the energy required for remote execution is expected to be smaller than the energy required for a local execution, i.e. $E_{offload} < E_{local}$. The energy consumption for a local execution can be approximated by the task complexity measured by the number of bytecode instructions that have to be executed (I_{task}) and the average energy required for a single bytecode instruction ($\overline{E_i}$) in mJ. The energy consumption for a remote execution is determined by the size of the input data in bytes (D_{input}), the result size in bytes (D_{result}), the inbound and outgoing bandwidths β_{in} and β_{out} of the consumer device in bytes/s, and the power for data transmission and data reception in mW ($P_{transmit}$ and $P_{receive}$). Thus, the offloading decision is:

$$\frac{D_{input}}{\beta_{out}} * P_{transmit} + \frac{D_{result}}{\beta_{in}} * P_{receive} < I_{task} * \overline{E_i} \tag{8.1}$$

Precisely knowing all the variables prior to task execution is not feasible in practice due to three reasons. First, the number of bytecode instructions and the size of the result data need to be predicted for each upcoming task execution. Due to a different parametrization, even tasks of the same type may vary in their number of executed instructions as well as in the sizes of their result data. Second, the energy consumption for sending and receiving data, as well as for executing code is highly device-dependent. Third, edge computing systems can be highly mobile, resulting in constantly changing context situations. Even a stationary system might be affected by the dynamic nature of its environment such as a varying bandwidth. Thus, the offloading decision is non-trivial.

Variable	Definition	In
D_{input}	Size of the input data in bytes	known a priori
D_{result}	Size of the result data in bytes	Section 8.3.1
I_{task}	Number of bytecode instructions of a task	Section 8.3.1
$\overline{E_i}$	Average energy consumption of 1 bytecode instruction on the consumer in mJ	Section 8.3.2
$P_{transmit}$	Energy consumption per s data transmission in mW	Section 8.3.2
$P_{receive}$	Energy consumption per s data reception in mW	Section 8.3.2
β_{out}	Outgoing bandwidth in bytes/s	Section 8.3.3
β_{in}	Inbound bandwidth in bytes/s	Section 8.3.3

Table 8.2.: Variables influencing the offloading decision (see Eq. 8.1).

We design *Voltaire* on the basis of Equation 8.1. *Voltaire*'s goal is to determine accurate values for all variables in the equation (cf. Table 8.2). This results in three tasks:

Predicting the number of bytecode instructions and the result size. Prior to task execution, the application informs *Voltaire* about task type, parameters, and size of the input data D_{input}. Based on crowd-sourced data of past executions of the same task, *Voltaire* predicts the number of bytecode instructions I_{task}, as well as the size of the result data D_{result}. In Section 8.3.1, we present how *Voltaire* integrates different regression analysis methods from machine learning for this.

Integrating device-dependent energy profiles. The power for sending and receiving data ($P_{transmit}$ and $P_{receive}$) and the average cost for executing a bytecode instruction locally ($\overline{E_i}$) depend on the consumer device that issues the task and the context. In Section 8.3.2, we show how *Voltaire* uses device-specific energy profiles to retrieve these values.

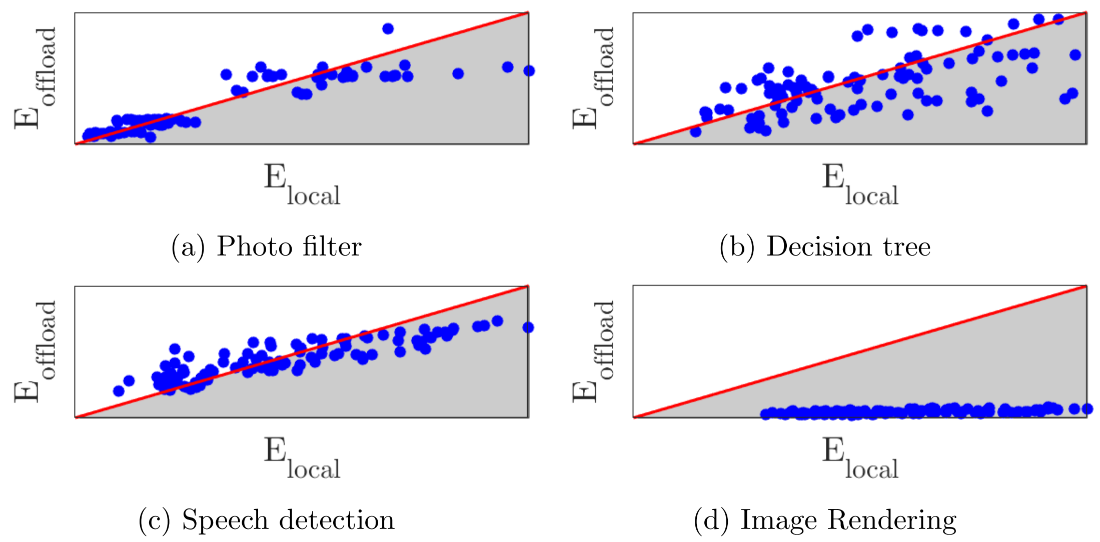
Estimating inbound and outgoing bandwidth. *Voltaire* estimates inbound bandwidth β_{in} and outgoing bandwidth β_{out} with low overhead by analyzing past data transmissions in the system. Section 8.3.3 describes this in more detail.

8.3. An Energy-Aware Scheduler for Precise Offloading Decisions

Voltaire is a centralized scheduler that runs on the broker of the Tasklet system. A centralized approach avoids massive communication overhead and is able to collect and reason on more data. Additionally, running on the broker allows *Voltaire* to apply more complex prediction and decision making mechanisms compared to approaches such as [71] and [215], where the offloading decision is made by the consumer device itself and kept simple to save energy. As a central instance for resource management, the broker ideally runs on a stable and powerful machine, which is in most cases connected to a constant power supply. *Voltaire*'s computationally intensive parts (e.g., updating the machine learning models) run mostly asynchronously to the offloading process, which reduces the influence on task completion times.

As *Voltaire* uses machine learning techniques that may be computationally intensive, a scalable deployment is important. Thus, it might be necessary to replicate *Voltaire* on several cloud or edge servers, which is a non-trivial task. Further options are to reduce the number of machine learning models per application, to update the models less frequently, or to limit the training data sets to a certain size. The realization of scalability is part of future work.

For each task, the mobile device sends a request to the broker. *Voltaire* analyzes the incoming request and decides whether to execute the task on the consumer device's TVMs or remotely. Figure 8.1 compares the energy consumption on the consumer device of four real-world applications from our experiments when executed locally or remotely. We observe that some applications such as the ray tracing-based image rendering always benefit from a remote (or local) execution. Due to its accurate predictions, *Voltaire* is, however, especially attractive for applications that sometimes benefit from a remote execution and sometimes from a local execution, depending on the task and the context. Here, offloading decisions are particularly complex. *Voltaire*'s offloading decision is not only based on the knowledge of previous task executions from this particular mobile device but from all consumers together. *Voltaire* thus exploits crowd-sourced data for the offloading decision. The broker chooses an appropriate provider for remote execution if *Voltaire* suggested a remote execution. The exact choice of the provider is out of this chapter's scope and does not affect the consumer's energy consumption if we assume a similar network connection to all providers. In the following sections, we show how *Voltaire* performs three basic tasks to determine the variables for a well-informed offloading decision from Equation 8.1 and Table 8.2.



(a) Photo filter (b) Decision tree

(c) Speech detection (d) Image Rendering

Figure 8.1.: Four real-world applications from our experiments. Each point represents the energy consumption of a local execution (E_{local}) and the energy consumption of a remote execution ($E_{offload}$). Thus, points in the gray area should be executed remotely and points above the line locally. *Voltaire* is especially attractive for applications (a) to (c) that sometimes benefit from a remote and sometimes from a local execution, based on task and context.

8.3.1. Predicting Number of Bytecode Instructions and Result Size

The three variables (i) number of bytecode instructions I_{task}, (ii) input data size D_{input}, (iii) and result size D_{result} differ for every task execution. Whereas the input data size D_{input} is known beforehand and reported from the consumer to *Voltaire*, *Voltaire* needs to predict the number of bytecode instructions I_{task} and the result size D_{result}. We propose to leverage feedback from past executions of the same task type to predict these two variables. Several executions of the same code may still vary considerably in the number of instructions that are executed and the result size due to different parameters and input data. To collect a larger data basis, *Voltaire* crowd-sources information about all previous executions of this task type in the whole system instead of only considering past executions by the current consumer device. The providers report a feedback after task execution to *Voltaire* including the task type, the number of bytecode instructions, and the result size. Crowd-sourced information is helpful for the current decision since the number of instructions and the result size are device-independent. Other devices may run the same application and, hence, execute tasks of the same type, which is valuable data for prediction. This is especially effective if the behavior of a user barely deviates from the usual usage of an application. Many applications, such as a photo filter or speech recognition, even do not allow much variance in their use in the first place. *Voltaire* integrates three methods for prediction based on the crowd-sourced data. In addition to the (i) average of prior executions and (ii) exponential smoothing, we propose to use regression analysis (iii) as a machine learning method to make precise predictions. In the following, we describe the prediction of bytecode instructions. *Voltaire* predicts the result size analogously.

Average

Similar to related work such as *MAUI* [71] or *MACS* [104], *Voltaire* integrates a strategy that predicts the number of bytecode instructions based on the average of past executions. This method works well if the number of bytecode instructions is mostly independent from the parameters or the input data.

Exponential Smoothing

Exponential smoothing [225] is a statistical method where previous executions are weighted less the older they are. It generally performs well in cases with periodic fluctuations in the data [226]. This can be expected for the number of bytecode instructions as consecutive tasks of a similar type often originate from a single application that is likely to start similar tasks. Let S_n be the prediction for instructions executed for the n'th execution and I_n be the corresponding actual number of instructions completed. The parameter α with $0 < \alpha < 1$ determines how much weight is assigned to the previous, true value.

$$I_{task} = S_n = \begin{cases} I_0 & \text{for } n = 1 \\ S_{n-1} * \alpha + I_{n-1} * (1 - \alpha) & \text{for } n \neq 1 \end{cases} \tag{8.2}$$

Regression Analysis

In experiments with real-world applications, we observe that executions of the same source code still vary considerably in terms of required bytecode instructions. We identify three reasons for these deviations. First, the size of the input data may differ. Classifying a small data set, for instance, requires less instructions than applying the same classifier to a larger data set. Second, the parameters of the particular task execution have a considerable influence. For example, calculating whether a number — which is passed as a parameter — is prime requires on average more instructions the higher this number is. Third, the special characteristics of the input data may affect the number of bytecode instructions. For instance, a photo filter task that converts all colors but red to grayscale, may run considerably longer if there are only a few red pixels in the image. To learn these complex influences of parameters and input data on the bytecode instructions, we integrate a machine learning module into *Voltaire*. For each task type, *Voltaire* trains three types of regression models based on these three reasons.

T1: Prediction based on the input data size. This type of regression model uses the input data size as a single feature to predict the number of bytecode instructions. Some tasks perform operations on all elements of the input data, e.g., on all pixels of a bitmap. In these cases, the number of instructions may be well predictable based on the input data size of the upcoming execution and the knowledge from past executions.

T2: Prediction based on the parameters. In addition to the input data size as a feature, *Voltaire* uses each parameter as a separate feature for this type of model. The input data size and the parameters of earlier executions are known to *Voltaire* anyways, which eliminates additional effort for developers.

T3: Prediction based on application-specific features. Experiments with real-world applications revealed that the number of bytecode instructions depends on the characteristics of the input data in some cases. Accurate predictions are difficult in these cases without any domain knowledge about the task. As an extreme and rather theoretical example, a task may perform a complex operation on every pixel of a bitmap, but only if the first pixel has a certain RGB value. *Voltaire*'s models of type `T3` allow the developer to pass further customized machine learning features for these complex cases. Some features may be easily computed, while more sophisticated ones may require more computation and more energy on the mobile device. A calculation on, e.g., only a fraction of the input data is thus recommended. In our experiments, we use 0.1 % of the input data.

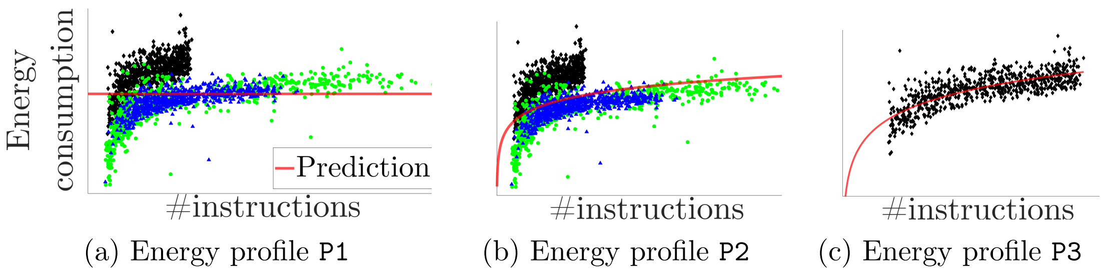
Voltaire performs online learning for all three types of models. When new feedback is available, *Voltaire* has a new training sample, consisting of the parameters, the input size, the number of bytecode instructions, the result size, and — if provided by the application developer — additional features. *Voltaire* uses multiple regression models, including decision tree, random forest, gradient boosting, linear regression, adaboost, and bayesian ridge. It periodically performs a 5-fold cross validation for the different regression models on the data and stores the best model for each task type.

8.3.2. Integrating Device-Dependent Energy Profiles

Voltaire uses device-specific energy profiles to retrieve the average energy consumption of executing one bytecode instruction locally ($\overline{E_i}$) and the energy consumption of sending and receiving data for one second ($P_{transmit}$ and $P_{receive}$) in the offloading process. All three parameters are device-dependent. The energy profile of a smartphone, for instance, differs from the energy profile of a laptop or a Raspberry Pi. We perform hardware-based profiling for each device type. For cases where hardware measurements are infeasible such as certain wearables, software-based profiling approaches (e.g., [227]) are viable alternatives. In future work, a combination of hardware- and software-based profiling may further improve accuracy, as, e.g., different OS versions on devices with the same hardware lead to different energy profiles. *Voltaire* uses two energy profiles per device type: a *CPU energy profile* and a *network energy profile*.

CPU Energy Profile

This profile models the energy consumption while executing a single bytecode instruction on a local TVM ($\overline{E_i}$). Our experiments show a tradeoff between accuracy of the profile and the effort for profile creation. We thus propose three approaches to create such profiles.



(a) Energy profile `P1` (b) Energy profile `P2` (c) Energy profile `P3`

Figure 8.2.: Three methods to create CPU energy profiles for *Voltaire* based on (a) a constant value for the individual device type, (b) a fitted curve for the individual device type, and (c) a fitted curve for the individual application. The values — originating from exemplary measurements in the evaluation — show the average energy consumption of a single bytecode instruction depending on the total number of instructions. Tasks from the same application are depicted in the same color.

P1: A constant value for the individual device type. We first assume that each bytecode instruction consumes the same amount of energy. In this case, a CPU energy profile may consist of a single value $\overline{E_i}$ that is device-dependent. Thus, one energy measurement with arbitrary applications is required for each device type. This method (cf. Figure 8.2a) is rather simple and thus applicable with low overhead in real-world systems.

P2: A fitted curve for the individual device type. During our measurements, we observe that the average energy consumption of a single bytecode instruction depends on the total number of instructions. Shorter tasks with less instructions also require less energy per instruction, whereas longer tasks with a high number of instructions tend to require more energy per instruction. To include such effects, we propose to run multiple applications and measure the energy consumption of a single instruction depending on the overall number of instructions of a task. The value for $\overline{E_i}$ is then calculated as a function of the (predicted) number of instructions of a task as $f_{instr,device}(I_{task})$, where $f_{instr,device}$ is a device-dependent function fitted on the measurement data (cf. Figure 8.2b).

P3: A fitted curve for the individual application. We also observe that the energy consumption of a fixed number of bytecode instructions varies across applications. Due to a more complex underlying interpretation by the TVM, some bytecode instructions run longer and thus consume more energy than others. This has a measurable effect on the energy consumption of different applications as the frequency with which certain instructions are used, varies. Therefore, we propose to perform energy measurements for each application separately as the most accurate method to create a CPU energy profile (cf. Figure 8.2c). Whereas methods `P1` and `P2` only require one measurement per device type, this method requires $a * d$ measurements in total, where a is the number of applications and d is the number of device types. New devices are introduced comparably rarely to the market, while new applications could be written by any developer at any time. Hence, we believe that in practice, $a >> d$ holds. To eliminate the need to perform energy measurements for each new application, we propose an alternative approach. We observe a linear relation between task completion time and energy consumption in our experiments. Thus, the average energy consumption of a single bytecode instruction of a new application $\overline{E}_{i,new}$ can be approximated by:

$$\overline{E}_{i,new} = \overline{E}_{i,bench\_device} * \frac{r_{instr,new}}{r_{instr,bench\_app}} \tag{8.3}$$

where $\overline{E}_{i,bench\_device}$ is the average energy consumption per instruction of a benchmark device, $r_{instr,new}$ the number of instructions of the new application executed per second on the benchmark device, and $r_{instr,bench\_app}$ the number of instructions of a benchmark application executed per second on the benchmark device[2]. With this approach, no energy measurements are required for a new application. New applications only have to be executed on a benchmark device with known energy consumption, which is feasible in real-world use cases.

Network Energy Profile

The network energy profile models the energy consumption of a certain device per second while transmitting and receiving data ($P_{transmit}$ and $P_{receive}$). The energy consumption is independent from the type of data that is transmitted or

[2] $\overline{E}_{i,new}$ and $\overline{E}_{i,bench\_device}$ are both functions of the total number of instructions as in `P2`. We omit this in the equation for better readability.

received. Thus, the network energy profile is application-independent. As shown in Figure 8.3, the network energy profile ideally contains separate energy values for transmitting and receiving under varying bandwidths.

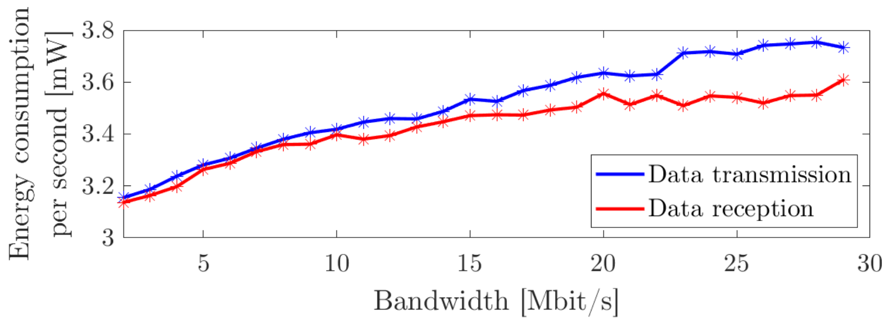


Figure 8.3.: Network energy profile based on exemplary measurements in the evaluation. Separate models for transmitting and receiving data characterize the respective energy consumption per second dependent on the bandwidth.

8.3.3. Estimating Inbound and Outgoing Bandwidth

Inbound and outgoing bandwidth β_{in} and β_{out} have to be measured at runtime as they fluctuate [228]. *Voltaire* uses an approach with low overhead. Consumers estimate the current bandwidth by measuring the duration of transmitting and receiving input and result data of known size. Thus, no additional data transfers are required. The bandwidth estimations are piggybacked to the task request, such that the broker and, hence, also *Voltaire* is informed about the recent value.

8.4. Evaluation

In this section, we evaluate *Voltaire* in a real-world testbed. First, we summarize the experimental setup in Section 8.4.1. We provide a detailed tutorial that explains how to reproduce the results in [229][3]. Second, we present the evaluation results in Section 8.4.2. Third, Section 8.4.3 discusses implications of the results. Fourth, Section 8.4.4 outlines potential threats to validity.

[3]Reference [229] is joint work with J. Edinger, S. Kaupmees, H. Trötsch, C. Krupitzer, and C. Becker.

8.4.1. Experimental Setup

We apply *Voltaire* in a real-world edge computing system that runs the Tasklet Middleware. We extend the Tasklet system to perform the crowd data collection at the broker, which hosts *Voltaire*. In the Tasklet system, providers inform the broker after task execution about the TVM that has become idle. We piggyback the task type, number of instructions, and result size to these messages to collect data for *Voltaire* with minimal overhead. A Raspberry Pi 4B with a 1.5 GHz ARM Quad Core CPU offloads tasks to providers. The Raspberry Pi works battery-powered in many use cases and is predestined for computation offloading due to its limited computational power. Since we aim to improve the battery consumption of the consumer device, the choice of the provider devices does not affect the measurements. A UM34C digital voltmeter measures the energy consumption of the consumer device. In the following, we isolate and discuss the energy consumption of the applications by subtracting the idle energy consumption from the energy consumption during the local or remote execution. The devices communicate over an IEEE 802.11n Wifi network. The net bit rate achieved at the application layer is 10 Mbit/s. We experimented with different setups (e.g., closer to the access point) and observed similar effects. Figure 8.4 shows an excerpt of a typical power measurement during the evaluation.

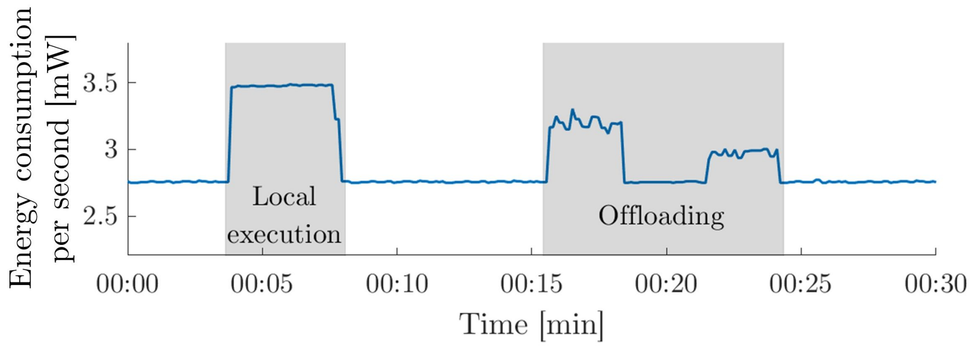


Figure 8.4.: Exemplary power consumption of the evaluation device.

We run three real-world applications in this setup (cf. Section 5.2.3): (i) a grayscale photo filter, (ii) a decision tree classifier, and (iii) a speech detection. All applications are offloaded to remote devices or executed locally on TVMs. For the photo filter application, we collect a database of 3,000 smartphone images of different sizes that are edited with the filter application in the experiments. We use 3,000 randomly created decision tree models with 2,500 to 5,000 nodes for the decision tree classifier use case. Additionally, we created 3,000 random data sets with 1,000,000 to 5,000,000 samples and 4 to 25 features each. For the speech detection application, we create 3,000 audio files, ranging from 15 to 150 s. The audio files originate from a total of 2.5 h of audio recordings that we collected in a room with five people doing group work.

8.4.2. Results

We perform three experiments that evaluate (i) the effectiveness of regression analysis for the prediction of the number of bytecode instructions and the result size, (ii) the influence of the device energy profiling method on prediction quality, and (iii) *Voltaire*'s potential to reduce energy consumption in a full-stack evaluation.

Experiment 1 (Predicting Task Complexity and Result Size)

In the first experiment, we evaluate the performance of regression analysis for predicting the number of bytecode instructions I_{task} and the result size D_{result} of an upcoming task. Thus, we isolate *Voltaire*'s machine learning part and predict the two variables with different regression methods. For each method, we evaluate the three model types `T1` to `T3` with 3,000 tasks per application. Table 8.3 describes the features that were used.

For each application, we perform a 5-fold cross validation with Python's scikit-learn [230] library. Table 8.4 shows the respective R^2 scores for the prediction of the number of bytecode instructions. We omit the values for the prediction of the result size here for reasons of clarity and comprehensibility. In general, the result size is well-predictable by *Voltaire* as it is often directly dependent on the input data size (e.g., for the photo filter).

The R^2 scores provide four insights. First, we observe that the quality of the prediction improves for all applications when extending the feature set from `T1` to `T3`. The additional features of `T2` and `T3` help *Voltaire* to better learn the reasons for the behavior of certain tasks.

Second, our experiment reveals that the performance of model type `T1` varies considerably for different applications. The number of bytecode instructions

Photo filter	
T1	Input data size
T2	Input data size, photo width, photo height, 9 parameters that describe the RGB range that remains in the original color
T3	Features T2, 4 ratios of pixels that need to be converted (according to the R, G, B, and RGB values) from a sample of 0.1 percent of all pixels
Decision tree classifier	
T1	Input data size
T2	Input data size, tree size, #rows, #features, #tree nodes
T3	Features T2, coefficient describing the balance of the tree, tree height, #nodes in the tree/maximum #nodes in a tree of this height
Speech detection	
T1	Input data size
T2	Input data size, amplitude threshold, #values skipped after speech is detected
T3	Features T2, 12 features from a sample of 0.1 percent of the overall audio file (ratio of sample values above threshold, average amplitude, a histogram with 10 groups showing the amplitude distribution of the sample)

Table 8.3.: Machine learning features used for regression analysis.

App/Model	**Decision Tree**			**Random Forest**			**Gradient Boost.**		
	T1	T2	T3	T1	T2	T3	T1	T2	T3
Photo filter	.838	.834	.967	.857	.916	.985	.860	.919	.986
Decision tree classifier	-.191	.994	.997	.129	.996	.998	.398	.997	.999
Speech detection	.324	.347	.865	.363	.650	.947	.483	.681	.960
App/Model	**Lin. Regression**			**Adaboost**			**Bayesian Ridge**		
	T1	T2	T3	T1	T2	T3	T1	T2	T3
Photo filter	.875	.901	.945	.872	.905	.968	.875	.875	.875
Decision tree classifier	.411	.997	.999	.406	.994	.994	.411	.997	.997
Speech detection	.508	.682	.929	.501	.661	.884	.508	.644	.644

Table 8.4.: Average R^2 scores for different regression analysis methods in a 5-fold cross validation on all 3,000 tasks per application. We highlight the highest scores for each application in gray.

for executing a photo filter task is strongly related to the input data size as the algorithm traverses and potentially converts each pixel. Thus, model type T1 performs comparably well here. Tasks that originate from the decision tree classifier are difficult to predict for model type T1. For this application, the input data size can be approximated by $r * f$, where r is the number of rows and f is the number of features in the input data set that has to be classified. The same classifier, however, requires considerably more bytecode instructions to classify a data set with many samples but a few features in comparison to a data set with few samples and many features.

Third, we observe that both photo filter and decision tree classifier are well-predictable with models of type `T2`. Only the speech detection application requires additional, application-specific features. We therefore argue that *Voltaire* is able to predict the number of bytecode instructions for many tasks very precisely, without requiring any additional programming effort by the application developer. *Voltaire*'s models of type `T2` only use input data size and parameters, which are reported to the broker anyways, as features for an accurate prediction. However, we also acknowledge that *Voltaire* requires additional domain knowledge to deliver high-quality predictions for tasks such as the speech detection tasks. Thus, it is important to offer application developers an easy-to-use API with the Tasklet Library to pass such features to *Voltaire*.

Fourth, this experiment shows that *Voltaire* is able to predict the number of instructions with high confidence when using models of type `T3`, which is an important step towards precise energy-aware offloading decisions. The cross validation reveals that gradient boosting is the most accurate regression method for these applications. In the following experiments, we therefore only show the results for gradient boosting.

Experiment 2 (Device-Specific Energy Profiles)

Voltaire uses device-specific CPU and network energy profiles. In practice, *Voltaire* estimates the energy for local and remote execution based on the predictions of the regression analysis, i.e., the quality of both regression analysis and energy profiling has an influence on the quality of *Voltaire*'s offloading decisions. In this experiment, we isolate the effect of the energy profiling methods. To achieve this, we calculate the estimated energy of local and remote execution under the assumption that the regression method predicted the correct number of instructions and result size. We compare this estimation with the true energy consumption of local and remote executions of the task in Figure 8.5.

We observe that the quality of the estimation of a local execution depends on the CPU energy profiling method. Method `P3` performs best for all applications. The profile is based on application-specific measurements and is thus more accurate than the more generic approaches `P1` and `P2`. We therefore recommend to create custom CPU energy profiles for each application with method `P3` to unleash

Voltaire's full potential. Whether method `P1` or `P2` performs better, depends on the particular application and its similarity to the applications used to create the profiles. We additionally observe that the energy consumption of a remote execution based on profiled values for data transmission $P_{transmit}$ and data reception $P_{receive}$ is well-predictable (see right part of Figure 8.5).

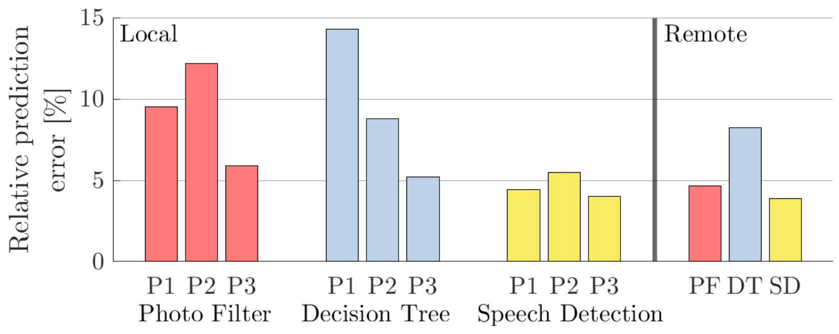


Figure 8.5.: Average deviation of the estimations for the energy consumption of local and remote executions with different strategies. This experiment assumes that the number of instructions is predicted correctly. (PF= Photo filter, DT= Decision tree, SD = Speech detection).

Experiment 3 (Full-Stack Evaluation)

In the final experiment, we evaluate *Voltaire*'s energy-saving potential by applying it to ten workflows of around 2.5 h each. Each workflow consists of 300 tasks that include 100 photo filter, 100 decision tree classifier, and 100 speech detection tasks. Figure 8.6 summarizes the average energy consumption per task across all workflows. Tables 8.5 to 8.7 provide an overview of the energy consumption per application (Table 8.5), the relative improvement (Table 8.6), and the number of correct offloading decisions (Table 8.7) in comparison to an ideal scheduler. As a baseline, we run two scheduling strategies in the Tasklet system that execute all tasks remotely or all tasks locally, respectively. We observe that for the 3,000 tasks in the evaluation, a remote execution is on average preferable to a local execution. Simply offloading all tasks, however, only decreases the energy consumption by 1.8 %. We therefore conclude that task-dependent, context-aware, and precise offloading decisions are necessary to realize the full energy-saving potential of computation offloading.

Appl.	**Energy consumption per task** (mJ)					
	Local	**Remote**	**Avg.**	**Exp. Sm.**	**GB** T3	**Ideal**
Photo filter	8210	8131	8296	8260	7340	7203
Decision tree classifier	5828	5942	5181	5224	4885	4876
Speech detection	8196	7770	8087	8087	7225	7144
Total	7411	7281	7188	7190	6483	6408

Table 8.5.: Energy consumption in the experiments (Exp. Sm. = Exponential Smoothing, GB `T3` = Gradient boosting with model type `T3`).

Appl.	**Improvement to local execution** (%)					
	Local	**Remote**	**Avg.**	**Exp. Sm.**	**GB** T3	**Ideal**
Photo filter	-	1.0	-1.0	-0.6	10.6	12.3
Decision tree classifier	-	-2.0	11.1	10.3	16.2	16.3
Speech detection	-	5.2	1.3	1.3	11.8	12.8
Total	-	1.8	3.0	3.0	12.5	13.5

Table 8.6.: The improvement to local execution (Exp. Sm. = Exponential Smoothing, GB `T3` = Gradient boosting with model type `T3`).

Appl.	**Correct decisions** (%)					
	Local	**Remote**	**Avg.**	**Exp. Sm.**	**GB** T3	**Ideal**
Photo filter	50.5	49.5	45.8	47.0	86.7	100
Decision tree classifier	53.6	46.4	71.6	70.5	96.2	100
Speech detection	43.3	56.7	47.1	47.2	86.6	100
Total	49.1	50.9	54.8	54.9	89.8	100

Table 8.7.: Percentage of correction decisions (Exp. Sm. = Exponential Smoothing, GB `T3` = Gradient boosting with model type `T3`).

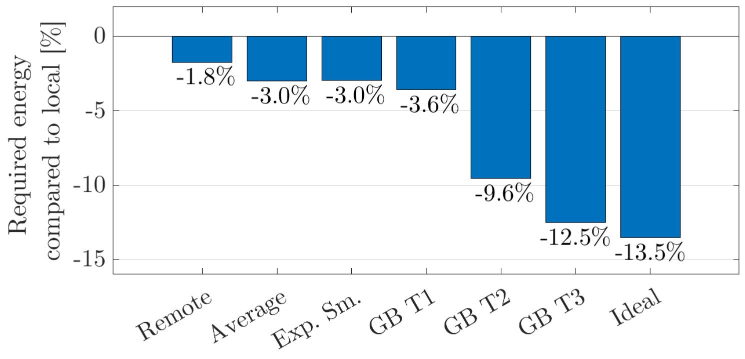


Figure 8.6.: Average improvement of the energy consumption by *Voltaire* in comparison to the status quo in the Tasklet system.
(Exp. Sm. = exponential smoothing, GB `T1` = gradient boosting with model type `T1`).

A first step is to use the average of past executions as a predictor of upcoming executions, which was proposed in related literature [71, 104] and which is also possible with *Voltaire*. We additionally apply an exponential smoothing method with $\alpha = 0.5$ as a logical extension that exploits the similarity of consecutive executions. Both strategies perform better than the baseline strategies with energy savings of 3.0 %. The first experiment led to the conclusion that *Voltaire* is able to accurately predict the number of bytecode instructions and the result size of an upcoming task. The final experiment shows that accurate prediction, together with precise device profiling, has a considerable positive effect on the energy consumption. *Voltaire*'s regression analysis with gradient boosting of model type `T3` reduces the energy consumption by 12.5 % in comparison to the status quo. This is an improvement of 9.0 % compared to related work that makes offloading decisions based on the average of past executions. *Voltaire* achieves an energy consumption that is on average only 1.0 % worse than a hypothetical, ideal scheduler that always makes the correct decision. Of 3,000 tasks, a small subset of 10.2 % of the tasks are incorrectly executed on the local or a remote device. We further observe in Table 8.5 that *Voltaire*'s performance is especially beneficial for the photo filter and the decision tree classifier, which confirms the results of the first experiment. Figure 8.6 underlines that *Voltaire*'s prediction based on parameters and application-specific features is a considerable improvement to a prediction solely on the input data size.

8.4.3. Discussion

In the evaluation, we have demonstrated *Voltaire*'s effectiveness with three real-world applications. *Voltaire* is able to predict the number of bytecode instructions and the result data size precisely with machine learning. Device-dependent energy profiles allow *Voltaire* to estimate the energy consumption of a local and a remote execution. In this section, we briefly discuss the implications of the results.

How generalizable are the results? *Voltaire* is an approach that is designed for neither a particular computation offloading system nor a special use case. We argue that *Voltaire* is additionally applicable to a wide range of (i) applications, (ii) devices, and (iii) transmission technologies. As far as applications are concerned, we have evaluated *Voltaire* with three representative real-world applications. We have shown that the prediction based on the input data size and the parameters (model type `T2`) leads to satisfactory results in many cases. For applications that require further features for the prediction, *Voltaire* offers a programming interface. We therefore argue that the results are generalizable to a variety of applications. We evaluate *Voltaire* with two Raspberry Pi. To use *Voltaire* with different device types, it is required to create energy profiles for the respective devices. This process is straightforward for all devices that allow hardware-based energy measurements. For other devices, software-based solutions may be used as a fallback. Thus, *Voltaire* is usable with a wide range of devices. Similar reasoning is applicable to alternative transmission technologies such as 4G or Bluetooth. The creation of novel network energy profiles is required, but the approach itself remains unaffected. We are therefore confident that the results are generalizable.

What is the effect of crowdsourcing on the prediction accuracy? The usage of crowdsourcing is a major design aspect of *Voltaire*. Instead of only analyzing past executions of the same task on the same device, *Voltaire* integrates crowd-sourced data from all prior executions of the task in the system. Figure 8.7 depicts the R^2 score of the gradient boosting models of type `T1` to `T3` as a function of the number of training data samples. The models are tested on data from 1,000 task executions for each application. We observe that the prediction quality improves with an increasing number of samples. Especially an accurate prediction of the speech detection application requires more samples ($R^2 = .909$ for 200 training samples and $R^2 = .960$ from the cross validation on 3,000 samples from Table 8.4). Thus, we conclude that applying a crowd-sourcing strategy is able to improve *Voltaire*'s prediction quality.

Would it be useful to distinguish between different instruction types? All types of CPU energy profiles discussed in Section 8.3.2 assume that the average energy cost of a certain bytecode instruction is constant. Although the accuracy of the energy profiles is satisfactory even with this assumption, distinguishing between different instruction types may be an important avenue for future work. In complementary measurements with the Tasklet system, we have observed that some bytecode instructions such as simple additions consume less energy than bytecode instructions that, e.g., load data from the memory. It is likely that this phenomenon also exists in other VM-based programming languages such as Java. We could refine *Voltaire* in two ways. First, the regression models may predict

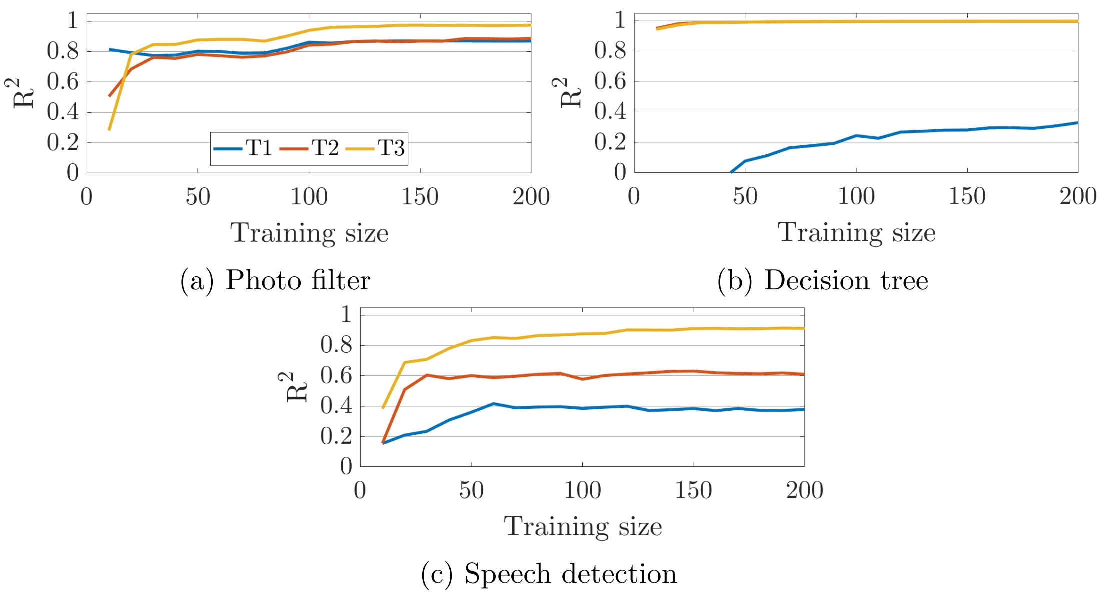


Figure 8.7.: Effect of training data size on R^2 score for three types of gradient boosting models (`T1` = prediction on input data size, `T2` = prediction on input data size and parameters, `T3` = prediction on input data size, parameters, and application-specific features). Crowdsourcing increases the training size in practice. In Figure (b), the curves for `T2` and `T3` overlap.

the number of bytecode instructions for each instruction type separately instead of the total number of instructions only. Second, the CPU energy profiles may contain separate values for each instruction type. Based on our complementary measurements and our observations during the experiments, this may be a valuable addition to *Voltaire* to further improve its performance.

8.4.4. Threats to Validity

In this chapter, we focus on the energy savings of computation offloading from a device-driven perspective. In our experiments, we neglect the consideration of task completion times. If task completion times are important, offloading offers further opportunities that should be taken into account in the decision making such as the choice of powerful providers [122]. A thorough analysis of the interplay of energy-awareness and task completion times is part of future work. We perform measurements in a real-world testbed with the Tasklet system. Therefore, measuring errors cannot be ruled out completely. Additionally, future work may include an evaluation with other device types, bandwidths, system load, or connection types such as Bluetooth. The same applies to the choice of machine learning techniques and energy profiling methods, which was extensive but not exhaustive. *Voltaire*'s effectiveness depends on the accuracy of the energy-profiling method. In the evaluation, we perform hardware-based profiling on the Raspberry Pi only. Profiling other devices — potentially also with software-based methods — may be less accurate and thus affect *Voltaire*'s performance negatively.

8.5. Summary

This chapter presents *Voltaire* — a scheduler for sophisticated energy-aware offloading decisions. *Voltaire* decides whether local or remote execution is beneficial for the energy consumption of a mobile device depending on the current context. Based on crowd-sourced data of past executions, *Voltaire* is able to accurately predict the complexity and the result size of an upcoming task with regression methods from machine learning. In addition, it uses device-dependent energy profiles that describe the energy consumption while computing and while transferring data. We apply *Voltaire* in an extensive real-world evaluation with three applications: (i) grayscale photo filter, (ii) decision tree classifier, and (iii) speech detection. During the evaluation, we perform hardware-based energy measurements on a Raspberry Pi. The results indicate that *Voltaire* reduces the energy consumption for task execution by 12.5 % compared to the status quo.

9. Discussion

The previous chapters presented the Tasklet system for computation offloading and the scheduling approaches *DataVinci*, *DecArt*, and *Voltaire* that serve as QoC mechanisms for the edge. Both the Tasklet system itself and the three QoC mechanisms are designed according to the requirements that we gathered in Chapter 3. In this chapter, we verify whether the artifact fulfills these requirements. Section 9.1 discusses the functional requirements; Section 9.2 the non-functional requirements. Table 9.1 summarizes the discussion.

Functional Requirement	**E**	**Non-Functional Req.**	**E**
R_F1 — Computation offloading	●	$R_{NF}1$ — Performance	●
R_F2 — Task-specific requirements	●	$R_{NF}2$ — Scalability	○
R_F3 — Heterogeneity support	●	$R_{NF}3$ — Robustness	●
R_F4 — Context-awareness	●	$R_{NF}4$ — Usability	○
R_F5 — Data placement	●	$R_{NF}5$ — Extensibility	●
R_F6 — Decentralized scheduling	●		
R_F7 — Energy-awareness	●		

Table 9.1.: Verification of functional and non-functional requirements. Column E (Evaluation) indicates whether a requirement is fulfilled.
(Req. = Requirement)
● fulfilled ○ partially fulfilled

9.1. Functional Requirements

We discuss the seven functional requirements from Section 3.3.

***R_F1* — Computation offloading:** The purpose of the Tasklet system is to offload workload from consumers to providers in form of Tasklets. Tasklets are self-contained units of computation that consist of executable bytecode, parameters, input data, and metadata. The Tasklet Middleware handles the exchange of Tasklets between consumers and providers. The providers run TVMs that execute the Tasklet bytecode. After the execution, the Tasklet Middleware transfers the results to the consumer via the network. Thanks to the Tasklet Library — an easy-to-use API — application programmers can launch Tasklets and receive results from various host languages with low effort. Thus, application programmers can integrate computation offloading with Tasklets into a variety of applications. The offloading process is entirely transparent to both application users and programmers as it is managed by the Tasklet Middleware. The broker is the central resource management software that performs task placement decisions. In various experimental evaluations (Section 6.5, Section 7.4, Section 8.4, and [122,123,126]), we show that the broker's task placement decisions lead to fast and energy-efficient computation offloading. We conclude that requirement R_F1 is fulfilled.

$\boldsymbol{R_F}$**2 — Task-specific requirements:** The Tasklet approach allows application programmers to specify so-called QoC goals, which are specific requirements for each Tasklet. The Tasklet Middleware applies a best-effort approach that does not provide any execution guarantees if the application programmer does not select any QoC goals. If desired, the application programmer can select one or multiple QoC goals such as *Reliability*, *Speed*, or *Energy*. The Tasklet Middleware enforces the selected QoC goals with a multitude of QoC mechanisms. It selects the proper mechanism based on the current context. The selection and the application of the QoC mechanisms is transparent for application users and programmers. The three scheduling approaches *DataVinci*, *DecArt*, and *Voltaire* that form the core contribution of this thesis are QoC mechanisms for data-intensive tasks, sub-second tasks, and energy-efficient tasks. As the QoC concept permits the fine-granular selection of QoC goals for each Tasklet, we consider requirement R_F2 to be fulfilled.

$\boldsymbol{R_F}$**3 — Heterogeneity support:** Overcoming the heterogeneity in the edge is a major design goal of the Tasklet system. Ideally, all device types shall be able to share their computing resources. The Tasklet Middleware is executable on Windows, macOS, Linux, Android, and iOS. The TVM abstracts from the local hardware and allows Tasklet execution on many device types. In real-world experiments, we offload Tasklets from and to smartphones [122,123,126], tablets [122,123], laptops [126], desktop PCs [122,126], cloud servers [122,123], and single-board computers such as Raspberry Pi [198]. In addition, we evaluate the Tasklet system with various real-world applications from different domains such as ray tracing or option pricing (cf. Section 5.2.3). These heterogeneous applications offload Tasklets with different characteristics. Some Tasklets such as the decision tree classifier are particularly data-intensive, others are rather long-running tasks with high complexity (ray tracing), or originate from user-facing and responsive applications with stricter deadlines (game AI, chess problems). The Tasklet Library, which is currently available in Java, C#, and Dart, allows application programmers to perform computation offloading without extensive programming effort. Launching Tasklets is even possible from other host languages via socket-based inter-process communication. We therefore conclude that the artifact fulfills requirement R_F3.

R_F4 — Context-awareness: The Tasklet system is context-aware. The providers inform the broker about their current state via the heartbeat channel. The broker exploits this context information while making task placement decisions. Discussing all situations where the Tasklet system uses context information would exceed the scope of this discussion. We therefore only briefly mention the context dimensions considered for the QoC mechanisms *DataVinci*, *DecArt*, and *Voltaire*. *DataVinci* monitors context of (i) the task such as input data size, (ii) the application such as desired availability and level of parallelism, (iii) the providers such as storage capacity, computational power, residence times, current load, and data queue size. *DecArt* makes task placement decisions based on the providers' computational power and the scheduling history. *Voltaire* optimizes the energy consumption of the consumer by considering (i) characteristics of the task including input data size and parameters, (ii) characteristics of the providers such as CPU and network energy profiles, and (iii) the available bandwidth. These lists of context dimensions do not claim to be exhaustive. Nonetheless, they prove that the Tasklet system is context-aware and therefore fulfills requirement R_F4.

R_F5 — Data placement: *DataVinci* is a QoC mechanism in the Tasklet system for data-intensive tasks. We introduce it in Chapter 6. *DataVinci* performs proactive data placement as specified in requirement R_F5. We evaluate *DataVinci* in detail in Section 6.5. We show with a real-world testbed that *DataVinci*'s context-aware replication strategy achieves task completion times within 14 % of a full replication while requiring less than 50 % of the data transfer overhead compared to this optimal but practically infeasible replication scheme. In the large-scale simulation, we additionally show further benefits of *DataVinci* for new data versions. The approach is able to manage the tradeoff between task completion times and data transfer overhead autonomously under varying system loads. *DataVinci* therefore realizes the benefits of computation offloading for data-intensive applications, which fulfills requirement R_F5.

***R_F*6 — Decentralized scheduling:** With *DecArt* (cf. Chapter 7), consumers in the Tasklet system are able to perform decentralized scheduling. The broker periodically informs these consumers about the available providers in form of cache lists. Based on the cache lists, the consumers make independent task placement decisions. We develop the provider selection algorithms *Drift* and *Bandit* to achieve low task completion times while avoiding collisions on fast providers. In the evaluation in Section 7.4, we show that *DecArt* decreases completion times of sub-second tasks by more than 30 % in comparison to centralized scheduling. In addition, *DecArt* performs within a 9 % range of a hypothetical omniscient scheduler. Thus, we conclude that *DecArt* fulfills requirement R_F6.

***R_F*7 — Energy-awareness:** Application programmers can set the *Energy* QoC goal for tasks that shall lead to minimal energy consumption on the consumer device. The Tasklet Middleware enforces this QoC goal with *Voltaire* — the QoC mechanism for energy-efficient Tasklet execution. We present *Voltaire* in Chapter 8. *Voltaire* uses regression models to predict the task complexity and the result data size of an upcoming task. With the help of device-dependent CPU and network energy profiles, it accurately predicts the energy consumption of local and remote execution, which makes it possible to select the more energy-efficient option. We apply *Voltaire* in a real-world experiment with the Tasklet system in Section 8.4. We set the *Energy* QoC goal for all Tasklets in this evaluation. As a result, *Voltaire* improves the energy consumption of the consumer device by 12.5 % in comparison to local executions and by 10.7 % in comparison to remote executions. We therefore consider requirement R_F7 to be fulfilled.

9.2. Non-Functional Requirements

We present five non-functional requirements for the Tasklet system in Section 3.4. We discuss them in the following.

***R_{NF}*1 — Performance:** The two essential performance metrics for computation offloading systems are average task completion time and energy consumption on the consumer device. The Tasklet system has a strong focus on task completion times. It offers several optimizations, i.e., QoC mechanisms, to reduce task completion times. We give an overview of these optimizations in Section 5.1.4. The optimizations improve different components of the task completion time. *DecArt* minimizes the scheduling time. Instead of contacting the broker, consumers offload tasks autonomously and, hence, faster. *DataVinci* shortens the time span for transferring task and input data to providers with proactive data placement. Workload partitioning [126] adapts the workload of a provider to its computational power and therefore minimizes the completion time of multi-task jobs. Similarly, the speed filter ensures that the selected providers have a higher throughput than a threshold value, which also decreases task execution times. Migration [126] reduces the amount of computation that is lost after a provider leave. It therefore improves the task completion times in case of failures. Fault-avoidance with Tasklets [90] selects reliable providers for execution. This mechanism avoids task abortions and, hence, also improves the average task completion times. The Tasklet system therefore has a large spectrum of QoC mechanisms available for fast computation offloading. It can choose the appropriate mechanism based on the context. The Tasklet system furthermore optimizes the energy consumption of the consumer device with *Voltaire*, which leads to an energy efficiency comparable to an ideal scheduler (cf. Section 8.4). While we recognize that there are always options for further improvement, e.g., by making the implementation more efficient, we can confidently conclude that the Tasklet system fulfills requirement $R_{NF}1$.

***R_{NF}*2 — Scalability:** The Tasklet system is scalable thanks to its hybrid peer-to-peer design. The load on the broker is reduced as consumers and providers directly exchange Tasklets and results. Nonetheless, the broker may become a bottleneck if the system size grows excessively. This is particularly important if the broker runs rather complex algorithms such as *Voltaire* to achieve certain QoC goals. In such cases, the Tasklet system allows spawning new brokers that are responsible for a subset of the participating devices each. These brokers form an overlay structure for coordination. As far as the three proposed QoC mechanisms for the edge — *DataVinci*, *DecArt*, and *Voltaire* — are concerned, *Voltaire* is the least scalable approach. *DataVinci* relies on heuristics and algorithms with low computational complexity. Even a single broker with decent computational power is able to handle this workload in systems with hundreds of devices. *DecArt* is a decentralized scheduling approach, which already implies good scalability as coordination among devices is not required. The distribution of the cache lists by the broker happens asynchronously and with low overhead as we illustrate in Section 7.4.3. *Voltaire* requires the execution of online machine learning on the broker, which is computationally intensive. We briefly discuss some measures to mitigate the computational complexity such as using fewer regression models in Section 8.3. If these measures are not sufficient, the remaining option is to spawn more brokers that share the workload. Thus, we infer that the Tasklet system is scalable to an extent that is suitable for edge computing systems. However, we consider requirement $R_{NF}2$ as only partially fulfilled as the coordination of multiple brokers was neither implemented nor evaluated so far.

$\boldsymbol{R_{NF}}$3 — Robustness: Two kinds of failures may occur in the Tasklet system: provider failures and broker failures. Both can happen if communication links fail, devices turn off, or users occupy the whole computing power of a device for their own purposes. Provider failures are more likely as the broker software typically runs on reliable devices and as the number of providers usually exceeds the number of brokers. The Tasklet system includes several measures to cope with provider failures, which we introduce in Section 5.1.4. Fault-avoidance [90] and migration [126] lead to fewer provider failures during Tasklet execution or to a less severe impact of such failures, respectively. The Tasklet system additionally reschedules aborted Tasklets if the application programmer sets the *Reliability* QoC goal. In this case, the system guarantees the correct execution even if multiple failures occur, which makes the Tasklet system robust to provider failures. Broker failures are more severe as brokers perform the central matchmaking between consumers and providers. In case of such failures, the affected consumers and providers can connect to other brokers that are working correctly. Additionally, *DecArt* allows the consumers to offload Tasklets autonomously based on the cache list. Although the information on the cache list will become outdated eventually, it is highly likely that at least some of the providers on the list are still available for a longer period of time. The Tasklet system is therefore also robust to broker failures and, hence, fulfills requirement $R_{NF}3$.

$\boldsymbol{R_{NF}}$**4 — Usability:** We design the Tasklet system to be easy to use for application users, resource providers, and application programmers. First, the offloading process is transparent for all three stakeholder groups. Second, consumers and providers are able to use their devices as usual while the Tasklet Middleware is running in the background. Third, application programmers can offload computation with Tasklets easily from several host languages with the Tasklet Library. The Tasklet Library offers an intuitive API for sending and receiving Tasklets as illustrated in Listing 5.2. In addition, application programmers should become familiar with the Tasklet language C-- quickly thanks to its simplicity and its resemblance with popular programming languages. While we are confident that the usability of the Tasklet system is high for all stakeholder groups, conducting a rigorous user study to prove this claim is part of future work. Thus, we consider requirement $R_{NF}4$ to be partially fulfilled.

$\boldsymbol{R_{NF}}$**5 — Extensibility:** We attest a high extensibility of the Tasklet system. First, launching Tasklets from a new host language is in theory possible without any changes to the Tasklet system code. Application programmers could in theory create the language-independent representation of a Tasklet without the help of the Tasklet library and communicate it to the middleware via sockets. Making the usage of a new host language convenient for the application programmer requires porting the Tasklet library to the new language, which we did in the past and which is possible with comparably low effort. Second, Tasklets could in principle even work with other languages for the Tasklet code apart from C-- if providers offer the appropriate execution environment. So far, we have experimented with the alternative programming languages WebAssembly and Lua and we plan to explore this path further in the future. Third, integrating new device types is possible by adjusting the Tasklet Middleware implementation. Reference [231] shows, for instance, how GPUs are able to execute Tasklets. Fourth, the separation of QoC goals and QoC mechanisms as described in Section 5.1.4 makes it possible to extend the range of available goals and mechanisms in a modular way. All these extensions have in common that they do not require any changes in the protocol or the core characteristics of the Tasklet system. We are currently working on an open-source release such that other researchers may extend the system in the future. We conclude that the Tasklet system fulfills requirement $R_{NF}5$.

10. Conclusion

The demand for computing power of software in areas such as AR/VR, machine learning, or image processing increases. In addition, users often execute such applications on mobile devices, which offer less computing power than, e.g., desktop PCs and furthermore suffer from limited battery capacity. Thus, the requirements of today's software can exceed the capabilities of the user device. This leads to unacceptable waiting times for the user and a quick drain of the device's battery. Computation offloading is a technology that is able to overcome these challenges. The user device — the consumer — offloads computationally intensive tasks to remote resource providers. The providers execute the computation and return the result via the network.

As we show in Chapter 2, computation offloading has been applied in various distributed computing paradigms such as cluster or cloud computing. In this thesis, we propose a computation offloading approach with a strong focus on modern edge computing environments. In such environments, end-user devices such as laptops or desktop PCs serve as computational resource providers. This introduces new challenges such as fluctuation and heterogeneity in terms of software and hardware. In Chapter 3, we specify requirements for a computation offloading system that makes edge computing faster and more energy-efficient. We review related work extensively in Chapter 4 and conclude that no existing approach fulfills the requirements. Thus, we introduce the Tasklet computation offloading system in Chapter 5. Tasklets are self-contained units of computation that can be exchanged seamlessly between consumers and providers with the Tasklet Middleware. Thanks to virtualization by the TVM, heterogeneous devices can participate in a single resource sharing system. With the Tasklet Library — an easy-to-use API — programmers can write applications that launch Tasklets and process the results. They can conveniently use their preferred programming language for the application as the Tasklet Library is available for several host languages. Only the computationally intensive part of the application is written in the Tasklet language C-- and offloaded as a Tasklet. To tailor the execution to application-specific requirements, the application programmer can select one or multiple QoC goals such as reliability, speed, or energy-awareness for each Tasklet. The Tasklet Middleware integrates several QoC mechanisms for enforcing these QoC goals transparently.

In this thesis, we design QoC mechanisms that overcome three main challenges for computation offloading in the edge. First, data-intensive tasks suffer from prolonged task completion times due to the time-consuming transfer of input data to remote providers. We introduce *DataVinci* in Chapter 6. *DataVinci* is a QoC mechanism for the Tasklet system that performs proactive placement of input data on one or multiple providers. These providers are then able to immediately start the Tasklet execution without a time-consuming ad-hoc data transfer. Second, responsive and user-facing tasks with sub-second deadlines are challenging in edge environments where communication latencies are in the same order of magnitude as execution times. We therefore present the QoC mechanism *DecArt* in Chapter 7. *DecArt* performs decentralized scheduling based on previously distributed cache lists. It eliminates the need for coordination with other peers or a central resource manager before offloading a task. In addition, *DecArt* integrates sophisticated provider selection algorithms, which avoid that powerful providers become congested. Third, optimizing the energy consumption of the consumer device requires the careful consideration of the context as even the same task may sometimes consume more energy if executed locally and sometimes if executed remotely. We design *Voltaire* — a QoC mechanism for energy efficiency — in Chapter 8. *Voltaire* uses regression models to predict the task complexity and the result data size of an upcoming task. With the help of device-dependent CPU and network energy profiles, it accurately predicts the energy consumption of local and remote execution, which makes it possible to select the more energy-efficient option.

We evaluate all three QoC mechanisms extensively in large-scale simulations and real-world testbeds. We observe that the Tasklet system in combination with *DataVinci*, *DecArt*, and *Voltaire* is able to realize the two main benefits of computation offloading — low task completion times and high energy efficiency — even in challenging edge computing environments. The discussion in Chapter 9 shows that the Tasklet system fulfills the majority of the requirements.

Outlook

Although the Tasklet system has proven its benefits in multiple experiments, opportunities for improvement exist. First, the design and implementation of the broker overlay in future work as discussed in Chapter 9 will increase the scalability of the Tasklet system. So far, we have focused on systems with a single broker. Especially when applying comparably complex QoC mechanisms such as *Voltaire*, larger systems may profit from having multiple brokers that are each responsible for a subset of the devices.

Second, we work towards releasing the Tasklet system as an open-source project. In the future, we plan to deploy the system in large-scale experiments with real users. Although we have performed multiple experiments in real-world testbeds, experiments that include dozens of users interacting with applications in their natural way would provide even more insights. We also plan to conduct a user study that assesses the usability of the Tasklet system for application programmers and application users. This will fulfill the usability requirement $R_{NF}4$.

Third, future versions of the Tasklet system should balance multiple objectives. Instead of optimizing either task completion times or energy consumption, the system should offer various ways to consider both metrics. A deadline approach, where the system optimizes the energy consumption as long as a certain deadline is not violated, is a conceivable alternative. Another approach is to monitor context dimensions such as the battery percentage to automatically select fast or energy-efficient execution.

Fourth, future work will address the incentivization of users. This thesis concentrates solely on the technical aspects of computation offloading. To achieve high market penetration in practice, however, user acceptance and participation are essential. Computation offloading in the edge only works if sufficiently many users share the computing power of their devices. A first step towards an incentive mechanism for the Tasklet system is presented in [232]. Quantifying the economic viability of the Tasklet system and assessing business models for a deployment in practice is another important avenue for future research that is closely connected to incentive mechanisms.

Bibliography

[1] M. Weiser, “The Computer for the 21st Century,” *Scientific American*, vol. 265, no. 3, 1991.

[2] M. Satyanarayanan, P. Bahl, R. Cáceres, and N. Davies, “The Case for VM-Based Cloudlets in Mobile Computing,” *IEEE Pervasive Comput.*, vol. 8, no. 4, 2009.

[3] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava, “A Survey of Computation Offloading for Mobile Systems,” *Mobile Netw. Appl.*, vol. 18, 2013.

[4] P. Mach and Z. Becvar, “Mobile Edge Computing: A Survey on Architecture and Computation Offloading,” *IEEE Commun. Surveys Tuts.*, vol. 19, no. 3, 2017.

[5] L. Lin, X. Liao, H. Jin, and P. Li, “Computation Offloading Toward Edge Computing,” *Proc. IEEE*, vol. 107, no. 8, 2019.

[6] M. J. Litzkow, M. Livny, and M. W. Mutka, “Condor - A Hunter of Idle Workstations,” University of Wisconsin, Tech. Rep., 1987.

[7] M. Satyanarayanan, “Pervasive Computing: Vision and Challenges,” *IEEE Pers. Commun.*, vol. 8, no. 4, 2001.

[8] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky, “SETI@home - Massively Distributed Computing for SETI,” *Comput. Sci. Eng.*, vol. 3, no. 78, 2001.

[9] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, “SETI@home: An Experiment in Public-Resource Computing,” *Commun. ACM*, vol. 45, no. 11, 2002.

[10] E. J. Korpela, “SETI@home, BOINC, and Volunteer Distributed Computing,” *Annu. Rev. Earth Planet. Sci.*, vol. 40, no. 1, 2012.

[11] A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, and V. S. Pande, "Folding@home: Lessons From Eight Years of Volunteer Distributed Computing," in *Proc. IPDPS.* IEEE, 2009.

[12] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud Computing and Grid Computing 360-Degree Compared," in *Proc. GCE.* IEEE, 2008.

[13] R. L. Grossman, "The Case for Cloud Computing," *IT Prof.*, vol. 11, no. 2, 2009.

[14] M. Armbrust et al., "A View of Cloud Computing," *Commun. ACM*, vol. 53, no. 4, 2010.

[15] P. Garcia Lopez et al., "Edge-centric Computing: Vision and Challenges," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 5, 2015.

[16] W. Shi and S. Dustdar, "The Promise of Edge Computing," *Comput.*, vol. 49, no. 5, 2016.

[17] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," *IEEE Internet Things J.*, vol. 3, no. 5, 2016.

[18] D. Schäfer, J. Edinger, J. M. Paluska, S. VanSyckel, and C. Becker, "Tasklets: Overcoming Heterogeneity in Distributed Computing Systems," in *Proc. ICDCSW.* IEEE, 2016.

[19] M. van Steen and A. S. Tanenbaum, "A Brief Introduction to Distributed Systems," *Comput.*, vol. 98, 2016.

[20] I. Bird, "Computing for the Large Hadron Collider," *Ann. Rev. Nucl. Particle Sci.*, vol. 61, 2011.

[21] K. Krauter, R. Buyya, and M. Maheswaran, "A Taxonomy and Survey of Grid Resource Management Systems for Distributed Computing," *Softw. Pract. Exper.*, vol. 32, no. 2, 2002.

[22] M. Baker and R. Buyya, "Cluster Computing: The Commodity Supercomputer," *Softw. Pract. Exper.*, vol. 29, no. 6, 1999.

[23] E. Boutin et al., "Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing," in *Proc. OSDI.* USENIX, 2014.

[24] A. Verma et al., "Large-scale Cluster Management at Google with Borg," in *Proc. EuroSys.* ACM, 2015.

[25] G. L. Valentini et al., "An Overview of Energy Efficiency Techniques in Cluster Computing Systems," *Cluster Comput.*, vol. 16, 2013.

[26] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, no. 1, 2008.

[27] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, "Parallel Data Processing with MapReduce: A Survey," *SIGMOD Record*, vol. 40, no. 4, 2011.

[28] C.-T. Chu et al., "Map-Reduce for Machine Learning on Multi-Core," in *Proc. NIPS.* MIT Press, 2006.

[29] A. McKenna et al., "The Genome Analysis Toolkit: A MapReduce Framework for Analyzing Next-Generation DNA Sequencing Data," *Genome Research*, vol. 20, no. 9, 2010.

[30] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proc. HotCloud.* USENIX, 2010.

[31] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," in *Proc. EuroSys.* ACM, 2007.

[32] I. Gog, M. Schwarzkopf, A. Gleave, R. N. M. Watson, and S. Hand, "Firmament: Fast, Centralized Cluster Scheduling at Scale," in *Proc. OSDI.* USENIX, 2016.

[33] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *Int. J. Supercomput. Appl. High Perform. Comput.*, vol. 15, no. 3, 2001.

[34] D. Thain, T. Tannenbaum, and L. Miron, "Distributed Computing in Practice: The Condor Experience," *Concurr. Comput. Pract. Exp.*, vol. 17, no. 2, 2005.

[35] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *Int. J. Supercomput. Appl. High Perform. Comput.*, vol. 11, no. 2, 1997.

[36] ——, "The Globus Project: A Status Report," in *Proc. HCW.* IEEE, 1998.

[37] R. Buyya, D. Abramson, and J. Giddy, "Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid," in *Proc. HPC-Asia.* IEEE, 2000.

[38] F. Berman et al., "Adaptive Computing on the Grid Using AppLeS," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 4, 2003.

[39] M. B. Qureshi et al., "Survey on Grid Resource Allocation Mechanisms," *J. Grid Comput.*, vol. 12, no. 2, 2014.

[40] D. K. Patel, D. Tripathy, and C. R. Tripathy, "Survey of Load Balancing Techniques for Grid," *J. Netw Comput. Appl.*, vol. 65, 2016.

[41] R. McClatchey, A. Anjum, H. Stockinger, A. Ali, I. Willers, and M. Thomas, "Scheduling in Data Intensive and Network Aware (DIANA) Grid Environments," *J. Grid Comput.*, vol. 5, no. 1, 2007.

[42] J. Blythe et al., "Task Scheduling Strategies for Workflow-based Applications in Grids," in *Proc. CCGRID.* IEEE, 2005.

[43] D. P. Anderson, "BOINC: A System for Public-Resource Computing and Storage," in *Proc. GRID.* IEEE/ACM, 2004.

[44] ——, "BOINC: A Platform for Volunteer Computing," *J. Grid Comput.*, vol. 18, no. 1, 2020.

[45] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," National Institute of Standards and Technology, Tech. Rep., 2011.

[46] T. Mastelic, A. Oleksiak, H. Claussen, I. Brandic, J. M. Pierson, and A. V. Vasilakos, "Cloud Computing: Survey on Energy Efficiency," *ACM Comput. Surveys*, vol. 47, no. 2, 2015.

[47] N. Fernando, S. W. Loke, and W. Rahayu, "Mobile Cloud Computing: A Survey," *Futur. Gener. Comput. Syst.*, vol. 29, no. 1, 2013.

[48] Z. Sanaei, S. Abolfazli, A. Gani, and R. Buyya, "Heterogeneity in Mobile Cloud Computing: Taxonomy and Open Challenges," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 1, 2014.

[49] A. ur Rehman Khan, M. Othman, S. A. Madani, and S. U. Khan, "A Survey of Mobile Cloud Computing Application Models," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 1, 2014.

[50] L. Atzori, A. Iera, and G. Morabito, “The Internet of Things: A Survey,” *Comput. Netw.*, vol. 54, 2010.

[51] S. Sarkar, S. Chatterjee, and S. Misra, “Assessment of the Suitability of Fog Computing in the Context of Internet of Things,” *IEEE Trans. Cloud Comput.*, vol. 6, no. 1, 2018.

[52] S. Choy, B. Wong, G. Simon, and C. Rosenberg, “The Brewing Storm in Cloud Gaming: A Measurement Study on Cloud to End-User Latency,” in *Proc. NetGames.* IEEE, 2012.

[53] L. M. Vaquero and L. Rodero-Merino, “Finding Your Way in the Fog: Towards a Comprehensive Definition of Fog Computing,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 5, 2014.

[54] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos, “Challenges and Opportunities in Edge Computing,” in *Proc. SmartCloud.* IEEE, 2016.

[55] M. Heck, J. Edinger, D. Schäfer, and C. Becker, “IoT Applications in Fog and Edge Computing: Where Are We and Where Are We Going?” in *Proc. ICCCN.* IEEE, 2018.

[56] R. Hasan, M. M. Hossain, and R. Khan, “Aura: An IoT Based Cloud Infrastructure for Localized Mobile Computation Outsourcing,” *Proc. MobileCloud*, 2015.

[57] E. Miluzzo, R. Cáceres, and Y. F. Chen, “Vision: mClouds - Computing on Clouds of Mobile Devices,” in *Proc. MCS.* ACM, 2012.

[58] N. Fernando, S. W. Loke, and W. Rahayu, “Dynamic Mobile Cloud Computing: Ad hoc and Opportunistic Job Sharing,” in *Proc. UCC.* IEEE, 2011.

[59] A. Mishra and G. Masson, “MoCCA: A Mobile Cellular Cloud Architecture,” in *Proc. Sarnoff.* IEEE, 2012.

[60] G. A. McGilvary, A. Barker, and M. Atkinson, “Ad Hoc Cloud Computing,” in *Proc. CLOUD.* IEEE, 2015.

[61] I. Yaqoob et al., “Mobile Ad Hoc Cloud: A Survey,” *Wirel. Commun. Mob. Comput.*, vol. 16, 2016.

[62] A. J. Ferrer, J. M. Marquès, and J. Jorba, "Towards the Decentralised Cloud: Survey on Approaches and Challenges for Mobile, Ad Hoc, and Edge Computing," *ACM Comput. Surv.*, vol. 51, no. 6, 2019.

[63] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, "Mobile Edge Computing: A Survey," *IEEE Internet Things J.*, vol. 5, no. 1, 2018.

[64] N. Hassan, K. L. A. Yau, and C. Wu, "Edge Computing in 5G: A Review," *IEEE Access*, vol. 7, 2019.

[65] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog Computing and Its Role in the Internet of Things," in *Proc. MCC.* ACM, 2012.

[66] H. Flores, P. Hui, S. Tarkoma, Y. Li, S. Srirama, and R. Buyya, "Mobile Code Offloading: From Concept to Practice and Beyond," *IEEE Commun. Magazine*, vol. 53, no. 3, 2015.

[67] G. Fedak, C. Germain, V. Néri, and F. Cappello, "XtremWeb : A Generic Global Computing System," in *Proc. CCGrid.* IEEE, 2001.

[68] S. Kosta, V. C. Perta, J. Stefa, P. Hui, and A. Mei, "Clone2Clone (C2C): Peer-to-Peer Networking of Smartphones on the Cloud," in *Proc. HotCloud.* USENIX, 2013.

[69] R. Kemp, N. Palmer, T. Kielmann, and H. Bal, "Cuckoo: a Computation Offloading Framework for Smartphones," in *Proc. MobiCASE.* Springer, 2010.

[70] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: Elastic Execution Between Mobile Device and Cloud," in *Proc. EuroSys.* ACM, 2011.

[71] E. Cuervo et al., "MAUI: Making Smartphones Last Longer with Code Offload," in *Proc. MobiSys.* ACM, 2010.

[72] G. C. Hunt and M. L. Scott, "The Coign Automatic Distributed Partitioning System," in *Proc. OSDI.* USENIX, 1999.

[73] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "COMET: Code Offload by Migrating Execution Transparently," in *Proc. OSDI.* USENIX, 2012.

[74] A. Jonathan, M. Ryden, K. Oh, A. Chandra, and J. Weissman, "Nebula: Distributed Edge Cloud for Data Intensive Computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 11, 2017.

[75] H. Flores et al., "Evidence-Aware Mobile Computational Offloading," *IEEE Trans. Mob. Comput.*, vol. 17, no. 8, 2018.

[76] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "ThinkAir: Dynamic Resource Allocation and Parallel Execution in the Cloud for Mobile Code Offloading," in *Proc. INFOCOM.* IEEE, 2012.

[77] J. Flinn, S. Park, and M. Satyanarayanan, "Balancing Performance, Energy, and Quality in Pervasive Computing," in *Proc. ICDCS.* IEEE, 2002.

[78] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, Low Latency Scheduling," in *Proc. SOSP.* ACM, 2013.

[79] R. K. Balan, M. Satyanarayanan, S. Park, and T. Okoshi, "Tactics-Based Remote Execution for Mobile Computing," in *Proc. MobiSys.* USENIX, 2003.

[80] S. Wu, C. Niu, J. Rao, H. Jin, and X. Dai, "Container-Based Cloud Platform for Mobile Computation Offloading," in *Proc. IPDPS.* IEEE, 2017.

[81] I. Baldini et al., "Serverless Computing: Current Trends and Open Problems," in *Research Advances in Cloud Computing*, S. Chaudhary, G. Somani, and R. Buyya, Eds. Springer, 2017.

[82] L. Baresi, F. Mendonça, and M. Garriga, "Empowering Low-Latency Applications Through a Serverless Edge Computing Architecture," in *Proc. ESOCC.* Springer, 2017.

[83] C. Cicconetti, M. Conti, and A. Passarella, "Low-Latency Distributed Computation Offloading for Pervasive Environments," in *Proc. PerCom.* IEEE, 2019.

[84] ——, "A Decentralized Framework for Serverless Edge Computing in the Internet of Things," *IEEE Trans. Netw. Service Manag.*, vol. 18, no. 2, 2021.

[85] A. K. Dey and G. D. Abowd, "Towards a Better Understanding of Context and Context-Awareness," in *Proc. HUC.* Springer, 1999.

[86] B. Schilit, N. Adams, and R. Want, "Context-Aware Computing Applications," in *Proc. WMCSA.* IEEE, 1994.

[87] M. Salehie and L. Tahvildari, "Self-Adaptive Software : Landscape and Research Challenges," *ACM Trans. Autonom. Adapt. Syst.*, vol. 4, no. 2, 2009.

[88] Y. Brun et al., "Engineering Self-Adaptive Systems through Feedback Loops," in *Software Engineering for Self-Adaptive Systems*, B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Springer, 2009.

[89] T.D. Braun et al., "A Comparison of Eleven Static Mapping Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems," *J. Parallel Distrib. Comput.*, vol. 61, 2001.

[90] J. Edinger, D. Schäfer, C. Krupitzer, V. Raychoudhury, and C. Becker, "Fault-Avoidance Strategies for Context-Aware Schedulers in Pervasive Computing Systems," in *Proc. PerCom.* IEEE, 2017.

[91] J. Sonnek, A. Chandra, and J. B. Weissman, "Adaptive Reputation-Based Scheduling on Unreliable Distributed Infrastructures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 11, 2007.

[92] F. Kon, R. H. Campbell, M. D. Mickunas, K. Nahrstedt, and F. J. Ballesteros, "2K: A Distributed Operating System for Dynamic Heterogeneous Environments," in *Proc. HPDC.* IEEE, 2000.

[93] T. Braud, P. Zhou, J. Kangasharju, and P. Hui, "Multipath Computation Offloading for Mobile Augmented Reality," in *Proc. PerCom.* IEEE, 2020.

[94] M. S. Elbamby, M. Bennis, and W. Saad, "Proactive Edge Computing in Latency-Constrained Fog Networks," in *Proc. EuCNC.* IEEE, 2017.

[95] F. Berg, F. Dürr, and K. Rothermel, "Optimal Predictive Code Offloading," in *Proc. MobiQuitous.* ACM, 2014.

[96] Y. Zhang, H. Liu, L. Jiao, and X. Fu, "To Offload or Not To Offload: An Efficient Code Partition Algorithm for Mobile Cloud Computing," in *Proc. CloudNet.* IEEE, 2012.

[97] R. B. Miller, "Response Time in Man-Computer Conversational Transactions," in *Proc. AFIPS FJCC.* ACM, 1968.

[98] Z. Chen et al., "An Empirical Study of Latency in an Emerging Class of Edge Computing Applications for Wearable Cognitive Assistance," in *Proc. SEC.* ACM/IEEE, 2017.

[99] N. Tolia, D. G. Andersen, and M. Satyanarayanan, "Quantifying Interactive User Experience on Thin Clients," *Computer*, vol. 39, no. 3, 2006.

[100] E. Cuervo et al., "Kahawai: High-Quality Mobile Gaming using GPU Offload," in *Proc. MobiSys.* ACM, 2015.

[101] A. Chien, B. Calder, S. Elbert, and K. Bhatia, "Entropia: Architecture and Performance of an Enterprise Desktop Grid System," *J. Parallel Distrib. Comput.*, vol. 63, no. 5, 2003.

[102] B. Calder, A. A. Chien, J. Wang, and D. Yang, "The Entropia Virtual Machine for Desktop Grids," in *Proc. VEE.* ACM, 2005.

[103] W. Cirne et al., "Labs of the World, Unite!!!" *J. Grid Comput.*, vol. 4, 2006.

[104] D. Kovachev and R. Klamma, "Framework for Computation Offloading in Mobile Cloud Computing," *Int. J. Artif. Intell. Interact. Multimed.*, vol. 1, no. 7, 2012.

[105] H.-Y. Chen, Y.-H. Lin, and C.-M. Cheng, "COCA: Computation Offload to Clouds using AOP," in *Proc. CCGrid.* IEEE, 2012.

[106] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura, "Serendipity: Enabling Remote Computing among Intermittently Connected Mobile Devices," in *Proc. MobiHoc.* ACM, 2012.

[107] R. N. Calheiros, C. Vecchiola, D. Karunamoorthy, and R. Buyya, "The Aneka Platform and QoS-Driven Resource Provisioning for Elastic Applications on Hybrid Clouds," *Futur. Gener. Comput. Syst.*, vol. 28, 2012.

[108] C. Vecchiola, R. N. Calheiros, D. Karunamoorthy, and R. Buyya, "Deadline-Driven Provisioning of Resources for Scientific Applications in Hybrid Clouds with Aneka," *Futur. Gener. Comput. Syst.*, vol. 28, 2012.

[109] D. Chae et al., "CMcloud: Cloud Platform for Cost-Effective Offloading of Mobile Applications," in *Proc. CCGrid.* IEEE/ACM, 2014.

[110] H. Qian and D. Andresen, "Jade: An Efficient Energy-Aware Computation Offloading System with Heterogeneous Network Interface Bonding for Ad-Hoc Networked Mobile Devices," in *Proc. SNPD.* IEEE, 2014.

[111] ——, "An Energy-Saving Task Scheduler for Mobile Devices," in *Proc. ICIS.* IEEE, 2015.

[112] I. Zhang et al., “Customizable and Extensible Deployment for Mobile/Cloud Applications,” in *Proc. OSDI.* USENIX, 2014.

[113] C. Shi, K. Habak, P. Pandurangan, M. Ammar, M. Naik, and E. Zegura, “COSMOS: Computation Offloading as a Service for Mobile Devices,” in *Proc. MobiHoc.* ACM, 2014.

[114] K. Habak, M. Ammar, K. A. Harras, and E. Zegura, “Femto Clouds: Leveraging Mobile Devices to Provide Cloud Service at the Edge,” in *Proc. CLOUD.* IEEE, 2015.

[115] C. Borcea, X. Ding, N. Gehani, R. Curtmola, M. A. Khan, and H. Debnath, “Avatar: Mobile Distributed Computing in the Cloud,” in *Proc. MobileCloud.* IEEE, 2015.

[116] G. Orsini, D. Bade, and W. Lamersdorf, “CloudAware: Empowering Context-Aware Self-Adaptation for Mobile Applications,” *Trans. Emerging Tel. Tech.*, vol. 29, no. 4, 2018.

[117] L. Lin, P. Li, X. Liao, H. Jin, and Y. Zhang, “Echo: An Edge-Centric Code Offloading System with Quality of Service Guarantee,” *IEEE Access*, vol. 7, 2019.

[118] H. Gedawy, K. A. Harras, K. Habak, and M. Hamdi, “FemtoClouds Beyond the Edge: The Overlooked Data Centers,” *IEEE Internet Things Mag.*, vol. 3, no. 1, 2020.

[119] M. Satyanarayanan et al., “Edge Analytics in the Internet of Things,” *IEEE Pervasive Comput.*, vol. 14, no. 2, 2015.

[120] ——, “An Open Ecosystem for Mobile-Cloud Convergence,” *IEEE Commun. Mag.*, vol. 53, no. 3, 2015.

[121] R. V. Van Nieuwpoort et al., “Ibis: A Flexible and Efficient Java-Based Grid Programming Environment,” *Concurrency Computat.: Pract. Exper.*, vol. 17, 2005.

[122] D. Schäfer, J. Edinger, J. M. Paluska, S. VanSyckel, and C. Becker, “Tasklets : ”Better than Best-Effort” Computing,” in *Proc. ICCCN.* IEEE, 2016.

[123] D. Schäfer, J. Edinger, J. Eckrich, M. Breitbach, and C. Becker, “Hybrid Task Scheduling for Mobile Devices in Edge and Cloud Environments,” in *Proc. PerCom Workshops.* IEEE, 2018.

[124] M. Breitbach, D. Schäfer, J. Edinger, and C. Becker, "Context-Aware Data and Task Placement in Edge Computing Environments," in *Proc. PerCom.* IEEE, 2019.

[125] J. Edinger, "Context-Aware Task Scheduling in Distributed Computing Systems," Ph.D. dissertation, University of Mannheim, 2019.

[126] D. Schäfer, J. Edinger, M. Breitbach, and C. Becker, "Workload Partitioning and Task Migration to Reduce Response Times in Heterogeneous Computing Environments," in *Proc. ICCCN.* IEEE, 2018.

[127] D. Schäfer, "Elastic Computation Placement in Edge-based Environments," Ph.D. dissertation, University of Mannheim, 2019.

[128] D. Schäfer, J. Edinger, C. Becker, and M. Breitbach, "Writing a Distributed Computing Application in 7 Minutes with Tasklets," in *Proc. Middleware Posters and Demos.* ACM, 2016.

[129] J. Edinger, D. Schäfer, M. Breitbach, and C. Becker, "Developing Distributed Computing Applications with Tasklets," in *Proc. PerCom Workshops.* IEEE, 2017.

[130] A. Ramakrishnan et al., "Scheduling Data-Intensive Workflows onto Storage-Constrained Distributed Resources," in *Proc. CCGrid.* IEEE, 2007.

[131] T. Kokilavani and D. George Amalarethinam, "Load Balanced Min-Min Algorithm for Static Meta-Task Scheduling in Grid Computing," *Int. J. Comput. Appl.*, vol. 20, no. 2, 2011.

[132] M. Breitbach, J. Edinger, D. Schäfer, and C. Becker, "DataVinci: Proactive Data Placement for Ad-Hoc Computing," in *Proc. IPDPS Workshops.* IEEE, 2021.

[133] T. Kosar and M. Balman, "A New Paradigm: Data-Aware Scheduling in Grid Computing," *Futur. Gener. Comput. Syst.*, vol. 25, no. 4, 2009.

[134] B. Allcock et al., "Data Management and Transfer in High-Performance Computational Grid Environments," *Parallel Comput.*, vol. 28, no. 5, 2002.

[135] X. Xia, F. Chen, Q. He, J. C. Grundy, M. Abdelrazek, and H. Jin, "Cost-Effective App Data Distribution in Edge Computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 1, 2021.

[136] B. Tang et al., "Incorporating Intelligence in Fog Computing for Big Data Analysis in Smart Cities," *IEEE Trans. Ind. Informatics*, vol. 13, no. 5, 2017.

[137] A. H. Alhusaini, V. K. Prasanna, and C. Raghavendra, "A Unified Resource Scheduling Framework for Heterogeneous Computing Environments," in *Proc. HCW*. IEEE, 1999.

[138] H. Casanova, G. Obertelli, F. Berman, and R. Wolski, "The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid," in *Proc. SC*. ACM/IEEE, 2000.

[139] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman, "Heuristics for Scheduling Parameter Sweep Applications in Grid Environments," in *Proc. HCW*. IEEE, 2000.

[140] K. Ranganathan and I. Foster, "Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications," in *Proc. HPDC*. IEEE, 2002.

[141] X. He, X. Sun, and G. von Laszweski, "QoS Guided Min-Min Heuristic for Grid Task Scheduling," *J. Comput. Sci. Technol.*, vol. 18, no. 4, 2003.

[142] D. G. Cameron, A. P. Millar, C. Nicholson, R. Carvajal-Schiaffino, K. Stockinger, and F. Zini, "Analysis of Scheduling and Replica Optimisation Strategies for Data Grids Using OptorSim," *J. Grid Comput.*, vol. 2, 2004.

[143] F. Desprez and A. Vernois, "Simultaneous Scheduling of Replication and Computation for Data-Intensive Applications on the Grid," *J. Grid Comput.*, vol. 4, no. 1, 2006.

[144] M. Tang, B. S. Lee, X. Tang, and C. K. Yeo, "The Impact of Data Replication on Job Scheduling Performance in the Data Grid," *Futur. Gener. Comput. Syst.*, vol. 22, 2006.

[145] R.-S. Chang, J.-S. Chang, and S.-Y. Lin, "Job Scheduling and Data Replication on Data Grids," *Futur. Gener. Comput. Syst.*, vol. 23, 2007.

[146] S. Venugopal, R. Buyya, and L. Winton, "A Grid Service Broker for Scheduling e-Science Applications on Global Data," *Concurr. Comput. Pract. Exp.*, vol. 18, 2006.

[147] A. Chervenak et al., “Data Placement for Scientific Applications in Distributed Environments,” in *Proc. GRID*. IEEE, 2007.

[148] A. Chakrabarti and S. Sengupta, “Scalable and Distributed Mechanisms for Integrated Scheduling and Replication in Data Grids,” in *Proc. ICDCN*. Springer, 2008.

[149] D. T. Nukarapu, B. Tang, L. Wang, and S. Lu, “Data Replication in Data Intensive Scientific Applications with Performance Guarantee,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 8, 2011.

[150] H. Liu, A. Abraham, V. Snášel, and S. McLoone, “Swarm Scheduling Approaches for Work-Flow Applications with Security Constraints in Distributed Data-Intensive Computing Environments,” *Inf. Sci. (Ny).*, vol. 192, 2012.

[151] R. Van Den Bossche, K. Vanmechelen, and J. Broeckhove, “Online Cost-Efficient Scheduling of Deadline-Constrained Workloads on Hybrid Clouds,” *Futur. Gener. Comput. Syst.*, vol. 29, no. 4, 2013.

[152] J. Taheri, Y. Choon Lee, A. Y. Zomaya, and H. J. Siegel, “A Bee Colony Based Optimization Approach for Simultaneous Job Scheduling and Data Replication in Grid Environments,” *Comput. Oper. Res.*, vol. 40, no. 6, 2013.

[153] O. Choudhury, D. Rajan, N. Hazekamp, S. Gesing, D. Thain, and S. Emrich, “Balancing Thread-level and Task-level Parallelism for Data-Intensive Workloads on Clusters and Clouds,” in *Proc. ICCC*. IEEE, 2015.

[154] X. Li, T. Jiang, and R. Ruiz, “Heuristics for Periodical Batch Job Scheduling in a MapReduce Computing Framework,” *Inf. Sci. (Ny).*, vol. 326, 2016.

[155] I. Casas, J. Taheri, R. Ranjan, L. Wang, and A. Y. Zomaya, “A Balanced Scheduler with Data Reuse and Replication for Scientific Workflows in Cloud Computing Systems,” *Futur. Gener. Comput. Syst.*, vol. 74, 2017.

[156] M. I. Naas, P. R. Parvedy, J. Boukhobza, and L. Lemarchand, “IFogStor: An IoT Data Placement Strategy for Fog Infrastructure,” in *Proc. ICFEC*. IEEE, 2017.

[157] Y. Li, J. Luo, J. Jin, R. Xiong, and F. Dong, "An Effective Model for Edge-Side Collaborative Storage in Data-Intensive Edge Computing," in *Proc. CSCWD.* IEEE, 2018.

[158] A. Aral and T. Ovatman, "A Decentralized Replica Placement Algorithm for Edge Computing," *IEEE Trans. Netw. Service Manag.*, vol. 15, no. 2, 2018.

[159] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design*, 4th ed. Addison-Wesley, 2005.

[160] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker, "A Survey on Engineering Approaches for Self-Adaptive Systems," *Pervasive and Mobile Computing*, vol. 17, 2015.

[161] K. Ousterhout et al., "The Case for Tiny Tasks in Compute Clusters," in *Proc. HotOS Workshop.* USENIX, 2013.

[162] M. Satyanarayanan, "The Emergence of Edge Computing," *IEEE Computer*, vol. 50, no. 1, 2017.

[163] M. Mitzenmacher, "The Power of Two Choices in Randomized Load Balancing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 10, 2001.

[164] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized Task-Aware Scheduling for Data Center Networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, 2015.

[165] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments," in *Proc. OSDI.* USENIX, 2008.

[166] J. Edinger, M. Breitbach, N. Gabrisch, D. Schäfer, C. Becker, and A. Rizk, "Decentralized Low-Latency Task Scheduling for Ad-Hoc Computing," in *Proc. IPDPS.* IEEE, 2021.

[167] U. Schwiegelshohn and R. Yahyapour, "Resource Allocation and Scheduling in Metasystems," in *Proc. HPCN.* Springer, 1999.

[168] O. Shehory, "A Scalable Agent Location Mechanism," in *Proc. ATAL.* Springer, 1999.

[169] V. Subramani, R. Kettimuthu, S. Srinivasan, and P. Sadayappan, "Distributed Job Scheduling on Computational Grids Using Multiple Simultaneous Requests," in *Proc. HPDC*. IEEE, 2002.

[170] E. Ogston and S. Vassiliadis, "Local Distributed Agent Matchmaking," in *Proc. CoopIS*. Springer, 2001.

[171] ——, "Matchmaking Among Minimal Agents Without a Facilitator," in *Proc. AGENTS*. ACM, 2004.

[172] J. S. Kim, B. Nam, P. Keleher, M. Marsh, B. Bhattacharjee, and A. Sussman, "Resource Discovery Techniques in Distributed Desktop Grid Environments," in *Proc. ICGRID*. ACM/IEEE, 2006.

[173] R. Ranjan, M. Rahman, and R. Buyya, "A Decentralized and Cooperative Workflow Scheduling Algorithm," in *Proc. CCGrid*. IEEE, 2008.

[174] J. Celaya and U. Arronategui, "A Highly Scalable Decentralized Scheduler of Tasks with Deadlines," in *Proc. Grid*. IEEE/ACM, 2011.

[175] Z. Dong, Y. Yang, C. Zhao, W. Guo, and L. Li, "Computing Field Scheduling: A Fully Decentralized Scheduling Approach for Grid Computing," in *Proc. ChinaGrid*. IEEE, 2011.

[176] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, Scalable Schedulers for Large Compute Clusters," in *Proc. EuroSys*. ACM, 2013.

[177] Y. Huang, N. Bessis, P. Norrington, P. Kuonen, and B. Hirsbrunner, "Exploring Decentralized Dynamic Scheduling for Grids and Clouds Using the Community-Aware Scheduling Algorithm," *Futur. Gener. Comput. Syst.*, vol. 29, no. 1, 2013.

[178] G. Jackson, P. Keleher, and A. Sussman, "Decentralized Scheduling and Load Balancing for Parallel Programs," in *Proc. CCGrid*. ACM/IEEE, 2014.

[179] A. Mohaisen, H. Tran, A. Chandra, and Y. Kim, "Trustworthy Distributed Computing on Social Networks," *IEEE T. Serv. Comput.*, vol. 7, no. 3, 2013.

[180] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu, “Hopper: Decentralized Speculation-Aware Cluster Scheduling at Scale,” in *Proc. SIGCOMM.* ACM, 2015.

[181] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel, “Hawk: Hybrid Datacenter Scheduling,” in *Proc. ATC.* USENIX, 2015.

[182] K. Karanasos et al., “Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters,” in *Proc. ATC.* USENIX, 2015.

[183] Z. Duan, W. Li, and Z. Cai, “Distributed Auctions for Task Assignment and Scheduling in Mobile Crowdsensing Systems,” in *Proc. ICDCS.* IEEE, 2017.

[184] A. J. Chakravarti, G. Baumgartner, and M. Lauria, “The Organic Grid: Self-Organizing Computation on a Peer-to-Peer Network,” *IEEE Trans. Syst., Man, Cybern. A, Syst. Humans*, vol. 35, no. 3, 2005.

[185] B. Peterson, G. Baumgartner, and Q. Wang, “A Decentralized Scheduling Framework for Many-Task Scientific Computing in a Hybrid Cloud,” *Serv. Trans. Cloud Comput.*, vol. 5, no. 1, 2017.

[186] A. Aral, I. Brandic, R. B. Uriarte, R. De Nicola, and V. Scoca, “Addressing Application Latency Requirements through Edge Scheduling,” *J. Grid Comput.*, vol. 17, no. 4, 2019.

[187] V. Scoca, A. Aral, R. De Nicola, and R. B. Uriarte, “Scheduling Latency-Sensitive Applications in Edge Computing,” in *Proc. CLOSER.* SCITEPRESS, 2018.

[188] X. Wang, Z. Ning, and S. Guo, “Multi-Agent Imitation Learning for Pervasive Edge Computing: A Decentralized Computation Offloading Algorithm,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 2, 2021.

[189] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, “A Scalable Content-Addressable Network,” in *Proc. SIGCOMM.* ACM, 2001.

[190] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang, “Refactoring Android Java Code for On-Demand Computation Offloading,” *ACM SIGPLAN Notices*, vol. 47, no. 10, 2012.

[191] M. A. Hassan and S. Chen, “Mobile MapReduce: Minimizing Response Time of Computing Intensive Mobile Applications,” in *Proc. MobiCom.* Springer, 2011.

[192] M. Goudarzi, M. Zamani, and A. T. Haghighat, “A Fast Hybrid Multi-Site Computation Offloading for Mobile Cloud Computing,” *J. Netw. and Comput. Appl.*, vol. 80, 2017.

[193] R. Kemp et al., “eyeDentify: Multimedia Cyber Foraging from a Smartphone,” in *Proc. ISM.* IEEE, 2009.

[194] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, “Effective Straggler Mitigation: Attack of the Clones,” in *Proc. NSDI.* USENIX, 2013.

[195] P. Auer, N. Cesa-Bianchi, and P. Fischer, “Finite-time Analysis of the Multiarmed Bandit Problem,” *Mach. Learn.*, vol. 47, no. 2, 2002.

[196] C. Becker and G. Schiele, “Middleware and Application Adaptation Requirements and Their Support in Pervasive Computing,” in *Proc. ICDCSW.* IEEE, 2003.

[197] C. Jiang et al., “Energy Aware Edge Computing: A Survey,” *Comput. Commun.*, vol. 151, 2020.

[198] M. Breitbach, J. Edinger, S. Kaupmees, H. Trötsch, C. Krupitzer, and C. Becker, “Voltaire: Precise Energy-Aware Code Offloading Decisions with Machine Learning,” in *Proc. PerCom.* IEEE, 2021.

[199] D. Huang, P. Wang, and D. Niyato, “A Dynamic Offloading Algorithm for Mobile Computing,” *IEEE Trans. Wirel. Commun.*, vol. 11, no. 6, 2012.

[200] A. Ravi and S. K. Peddoju, “Handoff Strategy for Improving Energy Efficiency and Cloud Service Availability for Mobile Devices,” *Wirel. Pers. Commun.*, vol. 81, 2015.

[201] B. Zhou, A. V. Dastjerdi, R. N. Calheiros, S. N. Srirama, and R. Buyya, “A Context Sensitive Offloading Scheme for Mobile Cloud Computing Service,” in *Proc. CLOUD.* IEEE, 2015.

[202] S. A. Karim and J. J. Prevost, “A Machine Learning Based Approach to Mobile Cloud Offloading,” in *Proc. SAI Computing Conference.* IEEE, 2017.

[203] A. Crutcher, C. Koch, K. Coleman, J. Patman, F. Esposito, and P. Calyam, "Hyperprofile-Based Computation Offloading for Mobile Edge Networks," in *Proc. MASS*. IEEE, 2017.

[204] C. M. Sarathchandra Magurawalage, K. Yang, L. Hu, and J. Zhang, "Energy-Efficient and Network-Aware Offloading Algorithm for Mobile Cloud Computing," *Comput. Networks*, vol. 74, 2014.

[205] H. Jadad, A. Touzene, K. Day, and N. Alzeidir, "A Cloud-Side Decision Offloading Scheme for Mobile Cloud Computing," *Int. J. Mach. Learn. Comput.*, vol. 8, no. 4, 2018.

[206] J. Niu, W. Song, and M. Atiquzzaman, "Bandwidth-Adaptive Partitioning for Distributed Execution Optimization of Mobile Applications," *J. Netw. Comput. Appl.*, vol. 37, 2014.

[207] H. Wu, W. Knottenbelt, K. Wolter, and Y. Sun, "An Optimal Offloading Partitioning Algorithm in Mobile Cloud Computing," in *Proc. QEST*. Springer, 2016.

[208] C. Xian, Y.-H. Lu, and Z. Li, "Adaptive Computation Offloading for Energy Conservation on Battery-Powered Systems," in *Proc. ICPADS*. IEEE, 2007.

[209] S. A. Saab, F. Saab, A. Kayssi, A. Chehab, and I. H. Elhajj, "Partial Mobile Application Offloading to the Cloud for Energy-Efficiency with Security Measures," *Sustain. Comput. Informatics Syst.*, vol. 8, 2015.

[210] K. Zhang et al., "Energy-Efficient Offloading for Mobile Edge Computing in 5G Heterogeneous Networks," *IEEE Access*, vol. 4, 2016.

[211] S.-J. Lee and X. Lin, "Energy-Aware Paired Sampling-Based Decision Model for Dynamic Mobile-to-Mobile Service Offloading," *IEEE Access*, vol. 5, 2017.

[212] M. Othman and S. Hailes, "Power Conservation Strategy for Mobile Computers Using Load Sharing," *ACM SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 2, no. 1, 1998.

[213] A. Rudenko, P. Reiher, G. J. Popek, and G. H. Kuenning, "Saving Portable Computer Battery Power through Remote Process Execution," *ACM SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 2, no. 1, 1998.

[214] I. Giurgiu, O. Riva, and G. Alonso, “Dynamic Software Deployment from Clouds to Mobile Devices,” in *Proc. Middlew.* ACM/IFIP/USENIX, 2012.

[215] A. Khairy, H. H. Ammar, and R. Bahgat, “Smartphone Energizer: Extending Smartphone’s Battery Life with Smart Offloading,” in *Proc. IWCMC.* IEEE, 2013.

[216] Y. D. Lin, E. T. Chu, Y. C. Lai, and T. J. Huang, “Time-and-Energy-Aware Computation Offloading in Handheld Devices to Coprocessors and Clouds,” *IEEE Syst. J.*, vol. 9, no. 2, 2015.

[217] J. Kwak, Y. Kim, J. Lee, and S. Chong, “DREAM: Dynamic Resource and Task Allocation for Energy Minimization in Mobile Cloud Systems,” *IEEE J. Sel. Areas Commun.*, vol. 33, no. 12, 2015.

[218] M. Shiraz, A. Gani, A. Shamim, S. Khan, and R. W. Ahmad, “Energy Efficient Computational Offloading Framework for Mobile Cloud Computing,” *J. Grid Comput.*, vol. 13, no. 1, 2015.

[219] Y. Hao, M. Chen, L. Hu, M. S. Hossain, and A. Ghoneim, “Energy Efficient Task Caching and Offloading for Mobile Edge Computing,” *IEEE Access*, vol. 6, 2018.

[220] X. Lyu, H. Tian, W. Ni, Y. Zhang, P. Zhang, and R. P. Liu, “Energy-Efficient Admission of Delay-Sensitive Tasks for Mobile Edge Computing,” *IEEE Trans. Commun.*, vol. 66, no. 6, 2018.

[221] S. Li, Y. Tao, X. Qin, L. Liu, Z. Zhang, and P. Zhang, “Energy-Aware Mobile Edge Computation Offloading for IoT over Heterogenous Networks,” *IEEE Access*, vol. 7, 2019.

[222] T. T. Nguyen, L. Le, and Q. Le-Trung, “Computation Offloading in MIMO Based Mobile Edge Computing Systems Under Perfect and Imperfect CSI Estimation,” in *IEEE Trans. Serv. Comput.*, 2019.

[223] X. Xu et al., “An Energy-Aware Computation Offloading Method for Smart Edge Computing in Wireless Metropolitan Area Networks,” *J. Netw. Comput. Appl.*, vol. 133, 2019.

[224] W. Zhang, Y. Wen, and H.-H. Chen, “Toward Transcoding as a Service: Energy-Efficient Offloading Policy for Green Mobile Cloud,” *IEEE Netw.*, vol. 28, no. 6, 2014.

[225] R. G. Brown and R. F. Meyer, “The Fundamental Theorem of Exponential Smoothing,” *Oper. Res.*, vol. 9, no. 5, 1961.

[226] S. Casolari and M. Colajanni, “On the Selection of Models for Runtime Prediction of System Resources,” in *Run-time Models for Self-managing Systems and Applications*, D. Ardagna and L. Zhang, Eds. Springer, 2010.

[227] D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. De Lucia, “Software-Based Energy Profiling of Android Apps: Simple, Efficient and Reliable?” in *Proc. SANER.* IEEE, 2017.

[228] R. Süselbeck, G. Schiele, P. Komarnicki, and C. Becker, “Efficient Bandwidth Estimation for Peer-to-Peer Systems,” in *Proc. P2P.* IEEE, 2011.

[229] M. Breitbach, J. Edinger, S. Kaupmees, H. Trötsch, C. Krupitzer, and C. Becker, “Artifact: Voltaire: Precise Energy-Aware Code Offloading Decisions with Machine Learning,” in *Proc. PerCom Workshops.* IEEE, 2021.

[230] F. Pedregosa et al., “Scikit-learn: Machine Learning in Python,” *J. Mach. Learn. Res.*, vol. 12, 2011.

[231] D. Schäfer, J. Edinger, and C. Becker, “GPU-Accelerated Task Execution in Heterogeneous Edge Environments,” in *Proc. ICCCN.* IEEE, 2018.

[232] J. Edinger, L. M. Edinger-Schons, D. Schäfer, A. Stelmaszczyk, and C. Becker, “Of Money and Morals - The Contingent Effect of Monetary Incentives in Peer-to-Peer Volunteer Computing,” in *Proc. HICSS.* ScholarSpace, 2019.

Appendix

A. Results of the *DecArt* Evaluation

Load	**Algorithm**	**Tasks**		**Jobs**		
		Total	Exec.	Mean	Median	% Misses
Lowest	C-RND	0.794	0.265	1.111	0.976	46.39
	C-GRD	0.588	0.060	0.639	0.632	0.01
	D-RND	0.550	0.257	0.910	0.786	29.39
	D-PRP	0.413	0.139	0.756	0.646	17.24
	D-GRD	0.524	0.055	0.893	0.866	34.39
Low	C-RND	0.789	0.261	1.109	0.976	46.51
	C-GRD	0.592	0.064	0.646	0.638	0.06
	D-RND	0.553	0.251	0.932	0.818	32.14
	D-PRP	0.414	0.140	0.757	0.647	17.58
	D-GRD	0.576	0.055	1.035	0.990	49.18
Medium	C-RND	0.795	0.249	1.134	0.974	46.15
	C-GRD	0.615	0.073	0.690	0.649	1.30
	D-RND	0.552	0.237	0.958	0.855	34.56
	D-PRP	0.412	0.137	0.752	0.644	16.96
	D-GRD	0.635	0.055	1.216	1.161	65.18
High	C-RND	1.928	0.242	3.612	1.006	50.71
	C-GRD	1.744	0.093	3.139	0.669	9.78
	D-RND	0.558	0.229	0.995	0.896	38.47
	D-PRP	0.413	0.138	0.755	0.647	17.17
	D-GRD	0.678	0.056	1.391	1.315	74.96
Highest	C-RND	20.116	0.230	40.504	1.158	62.40
	C-GRD	19.921	0.133	39.986	0.735	30.74
	D-RND	0.568	0.216	1.049	0.954	44.66
	D-PRP	0.414	0.138	0.763	0.655	17.42
	D-GRD	0.715	0.056	1.585	1.476	81.45

Table A.1.: Task and job metrics for basic centralized and decentralized scheduling algorithms (Highlighted: lowest task completion times, lowest job completion times, and fewest deadline misses per load).

Load	**Algorithm**	**Tasks**		**Jobs**		
		Total	Exec.	Mean	Median	% Misses
Lowest	**D-OMNI**	**0.330**	**0.056**	**0.379**	**0.369**	**0.00**
	C-GRD	0.588	0.060	0.639	0.632	0.01
	DFT	0.348	0.059	0.442	0.401	0.21
	BND	0.346	0.072	0.419	0.398	0.01
Low	**D-OMNI**	**0.332**	**0.058**	**0.381**	**0.372**	**0.00**
	C-GRD	0.592	0.064	0.646	0.638	0.06
	DFT	0.352	0.061	0.451	0.408	0.30
	BND	0.346	0.073	0.419	0.399	0.01
Medium	**D-OMNI**	**0.333**	**0.059**	**0.385**	**0.374**	**0.00**
	C-GRD	0.615	0.073	0.690	0.649	1.30
	DFT	0.354	0.063	0.462	0.415	0.51
	BND	0.345	0.071	0.418	0.397	0.01
High	**D-OMNI**	**0.336**	**0.062**	**0.392**	**0.379**	**0.01**
	C-GRD	1.744	0.093	3.139	0.669	9.78
	DFT	0.360	0.068	0.476	0.426	0.77
	BND	0.346	0.072	0.421	0.400	0.01
Highest	**D-OMNI**	**0.343**	**0.070**	**0.405**	**0.386**	**0.17**
	C-GRD	19.921	0.133	39.986	0.735	30.74
	DFT	0.369	0.075	0.500	0.445	1.30
	BND	0.347	0.071	0.427	0.404	0.02

Table A.2.: Task and job metrics for best centralized and decentralized scheduling algorithms in comparison to the omniscient decentralized scheduler.

B. Publications Contained in This Thesis

[123] D. Schäfer, J. Edinger, J. Eckrich, M. Breitbach, and C. Becker, "Hybrid Task Scheduling for Mobile Devices in Edge and Cloud Environments," in *Proc. PerCom Workshops*. IEEE, 2018.

[124] M. Breitbach, D. Schäfer, J. Edinger, and C. Becker, "Context-Aware Data and Task Placement in Edge Computing Environments," in *Proc. PerCom*. IEEE, 2019.

[126] D. Schäfer, J. Edinger, M. Breitbach, and C. Becker, "Workload Partitioning and Task Migration to Reduce Response Times in Heterogeneous Computing Environments," in *Proc. ICCCN*. IEEE, 2018.

[128] D. Schäfer, J. Edinger, C. Becker, and M. Breitbach, "Writing a Distributed Computing Application in 7 Minutes with Tasklets," in *Proc. Middleware Posters and Demos*. ACM, 2016.

[129] J. Edinger, D. Schäfer, M. Breitbach, and C. Becker, "Developing Distributed Computing Applications with Tasklets," in *Proc. PerCom Workshops*. IEEE, 2017.

[132] M. Breitbach, J. Edinger, D. Schäfer, and C. Becker, "DataVinci: Proactive Data Placement for Ad-Hoc Computing," in *Proc. IPDPS Workshops*. IEEE, 2021.

[166] J. Edinger, M. Breitbach, N. Gabrisch, D. Schäfer, C. Becker, and A. Rizk, "Decentralized Low-Latency Task Scheduling for Ad-Hoc Computing," in *Proc. IPDPS*. IEEE, 2021.

[198] M. Breitbach, J. Edinger, S. Kaupmees, H. Trötsch, C. Krupitzer, and C. Becker, "Voltaire: Precise Energy-Aware Code Offloading Decisions with Machine Learning," in *Proc. PerCom*. IEEE, 2021.

[229] M. Breitbach, J. Edinger, S. Kaupmees, H. Trötsch, C. Krupitzer, and C. Becker, "Artifact: Voltaire: Precise Energy-Aware Code Offloading Decisions with Machine Learning," in *Proc. PerCom Workshops*. IEEE, 2021.

C. Curriculum Vitae

Since 08/2018	Research Assistant Chair of Information Systems II Universität Mannheim
09/2016 – 06/2018	Master of Science Mannheim Master in Management Universität Mannheim
09/2013 – 06/2016	Bachelor of Science Business Informatics Universität Mannheim