Multi-user computation offloading in mobile edge computing

Shuai Yu

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

Doctor of Philosophy
Sorbonne Université

Specialization

COMPUTER SCIENCE
(École Doctorale Informatique, Télécommunication et Électronique "EDITE de Paris")

presented by

Mr Shuai Yu

Submitted for the degree of

Doctor of Philosophy of Sorbonne Université

Title:

Multi-user Computation Offloading in Mobile Edge Computing

Defense: 30^{th} April 2018

Committee:

Mr Nadjib Aitsaadi	**Reviewer**	**Professor, ESIEE Paris - France**
Mr Paolo Bellavista	**Reviewer**	**Professor, University of Bologna - Italy**
Mr Sidi-Mohammed Senouci	**Examiner**	**Professor, University of Bourgogne - France**
Mr Stefano Paris	**Examiner**	**Senior Researcher, Huawei - France**
Mr Stefano Secci	**Examiner**	**Associate Professor/HDR, Sorbonne University - France**
Mr Rami Langar	**Supervisor**	**Professor, Université Paris Est Marne-la-Vallée - France**

Acknowledgements

This thesis is dedicated to the memory of my father, who contributed to this thesis in ways he will never know.

First, I would like to express my deepest gratitude to my supervisor Pr. Rami Langar who has given me the opportunity to do my thesis and to be involved in the projects of PODIUM and European FP7 MobileCloud where I have learnt a lot.

A very special thanks goes out to my former advisor Pr. Li Wang in BUPT, for her excellent guidance, assistance, patience and motivation.

I would also like to thank all the jury members for reading my thesis, for offering their valuable time and for their constructive feedbacks.

I am grateful to Pr. Zhu Han from University of Houston, for his collaboration, expertise and precious advices that helped me to learn about how to publish journal papers.

I must acknowledge as well my friends and colleagues in the PHARE team who have been so supportive and helpful along the way of doing my thesis. A particular thanks to Boutheina Dab, Xin Wang, Longbiao Chen and Lyazidi Mohammed Yazid for their constructive collaboration.

I would like to express my heartfelt gratitude to my friends Xin Ma, Zijian Li, Fan Bai Yanan Li and Jia Guo who gave me their help and time in Paris.

I would also like to thank China Scholarship Council (CSC) for supporting my study in France.

Last, I would especially like to thank my amazing family for their love, support, and constant encouragement I have gotten over the years. In particular, I am highly indebted to my mother. Without her faith in me, I can not go through the most difficult moments in my life, and this thesis would never have been achieved.

Abstract

Mobile Edge Computing (MEC) is an emerging computing model that extends the cloud and its services to the edge of the network. Consider the execution of emerging resource-intensive applications in MEC network, computation offloading is a proven successful paradigm for enabling resource-intensive applications on mobile devices. Moreover, in view of emerging mobile collaborative application (MCA), the offloaded tasks can be duplicated when multiple users are in the same proximity. This motivates us to design a collaborative computation offloading scheme for multi-user MEC network. In this context, we separately study the collaborative computation offloading schemes for the scenarios of MEC offloading, device-to-device (D2D) offloading and hybrid offloading, respectively.

In the MEC offloading scenario, we assume that multiple mobile users offload duplicated computation tasks to the network edge servers, and share the computation results among them. Our goal is to develop the optimal fine-grained collaborative offloading strategies with caching enhancements to minimize the overall execution delay at the mobile terminal side. To this end, we propose an optimal offloading with caching-enhancement scheme (OOCS) for femto-cloud scenario and mobile edge computing scenario, respectively. Simulation results show that compared to six alternative solutions in literature, our single-user OOCS can reduce execution delay up to 42.83% and 33.28% for single-user femto-cloud and single-user mobile edge computing, respectively. On the other hand, our multi-user OOCS can further reduce 11.71% delay compared to single-user OOCS through users' cooperation.

In the D2D offloading scenario, we assume that where duplicated computation tasks are processed on specific mobile users and computation results are shared through Device-to-Device (D2D) multicast channel. Our goal here is to find an optimal network partition for D2D multicast offloading, in order to minimize the overall energy consumption at the mobile terminal side. To this end, we first propose a D2D multicast-based computation offloading framework where the problem is modelled as a combinatorial optimization problem, and then solved using the concepts of from maximum weighted bipartite matching and coalitional game. Simulation results show that our algorithm can significantly reduce the energy consumption, and guarantee the battery fairness among multiple users at the same time.

We then extend the D2D offloading to hybrid offloading with social relationship consideration. In this context, we propose a hybrid multicast-based task execution framework for mobile edge computing, where a crowd of mobile devices at the network edge leverage network-assisted D2D collaboration for wireless distributed computing and outcome sharing. The framework is social-aware in order to build effective D2D links. A key objective of this framework is to achieve an energy-efficient task assignment policy for mobile users. To do so, we first introduce the social-

aware hybrid computation offloading system model, and then formulate the energy-efficient task assignment problem by taking into account the necessary constraints. We next propose a monte carlo tree search based algorithm, named, *TA-MCTS* for the task assignment problem. Simulation results show that compared to four alternative solutions in literature, our proposal can reduce energy consumption up to 45.37%.

Last but not least, the deployment problems require combinatorial optimization and are NP-hard. To overcome the great complexity involved, we formulate the offloading decision problem as a multi-label classification problem and develop the Deep Supervised Learning (DSL) method to minimize the computation and offloading overhead. As the number of fine-grained components n of an application grows, the exhaustive strategy suffers from the exponential time complexity $O(2^n)$. Fortunately, the complexity of our learning system is only $O(mn)^2$, where m is the number of neurons in a hidden layer which indicates the scale of the learning model.

Key Words

Mobile Edge Computing, computation offloading, edge caching, mobile collaborative applications, coalitional game, wireless distributed computing, multicast communication, social-awareness, monte carlo tree search, deep supervised learning.

Table of contents

Chapter 1

Introduction

Contents

Nowadays, the advances in hardware technology has leaded to more powerful mobile devices in terms of memory, processing speed and network connectivity. This development has pushed to the pervasive computing era, in which different mobile devices, ranging from smartphones, tablets and laptops to low-power sensors have widely penetrated to our everyday life. Accompanied by the emergence of near-to-eye display technologies, such as Google Glass, a variety of Mobile Collaborative Applications (MCA) [1–4] are developed to meet the user's requirements, such as augmented

reality (AR) [1, 3], collaborative gaming [5, 6] and mobile crowd sensing applications [7, 8]. These applications make use of complex algorithms for camera tracking, image processing and pattern recognition which are resource-intensive and, hence, beyond the capabilities of current mobile devices.

A potential solution to address the challenges is through mobile cloud computation offloading [9, 10]. However, when considering 1 ms to 5ms end-to-end latency required by 5G for a class of applications (called the "Tactile Internet" [11]), the traditional cloud may not be suitable for code offloading due to the high and variable latency to distant datacenters, Mobile Cloud Computing (MCC) is thus not adequate for a wide-range of emerging mobile applications that are latency-critical. Presently, new network architectures are being designed to better integrate the concept of Cloud Computing into mobile networks. On the other hand, caching most popular contents at the network edge can reduce latency and improve user's quality-of-experience (QoE) [12–15]. Thus, a promising approach to tackle this challenge is to design new network architectures that bring cloud infrastructure (computation and storage abilities) closer to the end users [16].

Motivated by above facts, the European project TROPIC (distributed computing, storage and radio resource allocation over cooperative femtocells) and the European Telecommunications Standards Institute (ETSI) proposed femto-cloud [17] and mobile edge computing (MEC) [18], respectively. In the proposed architectures, substantial compute and storage resources are placed at the edge of the Internet, in close proximity to mobile devices, sensors, end users, and Internet of Things devices. This physical proximity improves latency, bandwidth, trust, and survivability, thus allowing a large class of state-of-the-art applications, like Big Data and the Internet of Things, to be deployed in a very effective way.

In this context, this thesis addresses the problem of multi-user computation offloading and caching for 5G mobile edge computing. In what follows, we provide a description of the context and motivation in section 1.1, and outline the architecture of the MEC system in section 1.2. Then, section 1.3 introduce the key issues of MEC system. Section 1.4 and 1.5 respectively addresses the thesis problematic and the contributions. Finally, section 1.6 presents the organization of this manuscript.

1.1 Context and Motivations

MEC has emerged as a key enabling technology for realizing the IoT and 5G visions. MEC research lies at the intersection of mobile computing and wireless communications, where the existence of many research opportunities has resulted in a highly active area. In recent years, researchers from both academia and industry have investigated a wide-range of issues related to MEC, including system and network modeling, optimal control, multiuser resource allocation, implementation and standardization.

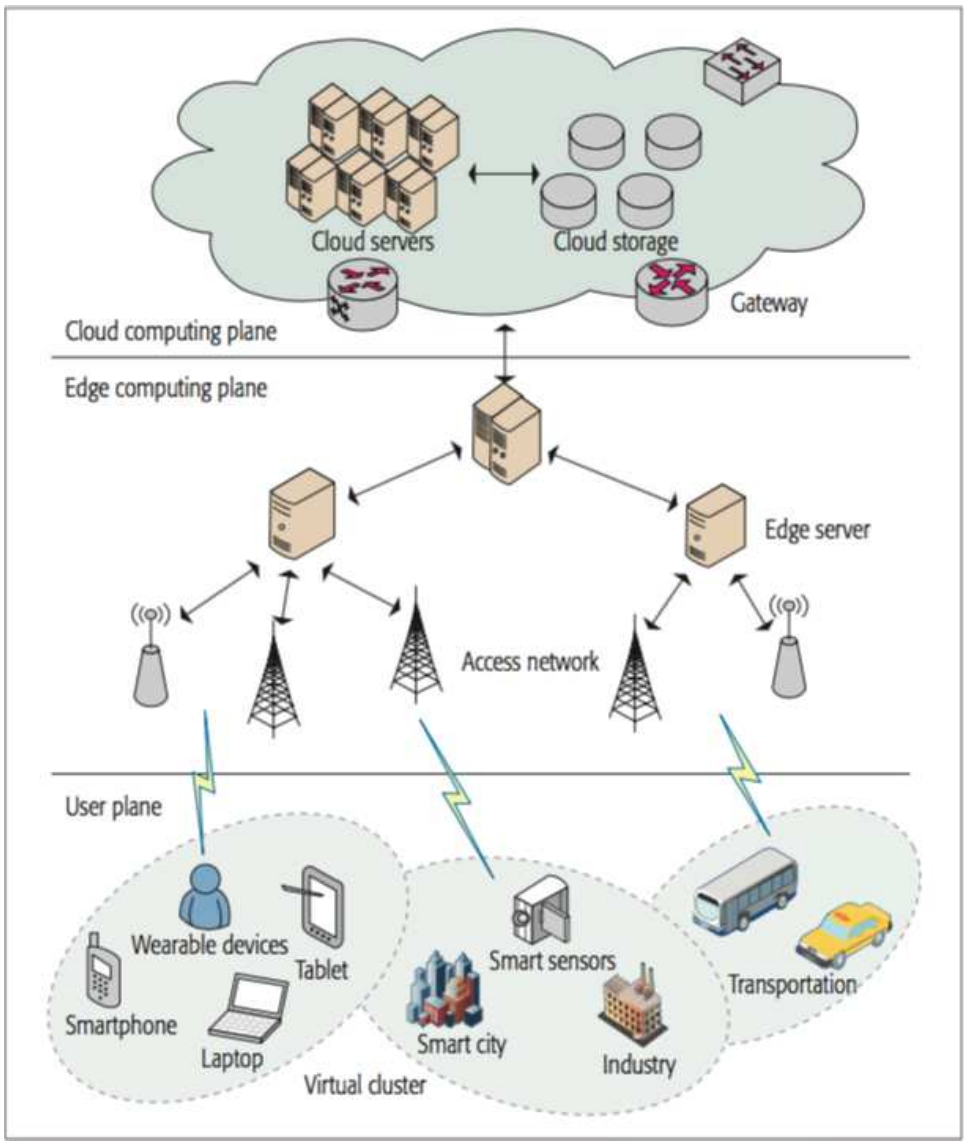


Figure 1.1 – MEC-enabled cloud computing network [19].

In view of exist works, there still lack of following concerns:

1.1.1 Muti-user computation offloading with edge caching consideration in MEC:

There is an increasing number of emerging mobile applications that will benefit from MEC, by offloading their computation-intensive tasks to the MEC servers for cloud execution. For certain types of emerging Mobile Collaborative Applications (MCAs), such as mobile crowd sensing applications and AR applications Multiple users in the same neighborhood typically look at the same

scene, track the same environment, and need to recognize the same objects, so they can benefit from collaboration and computation/data sharing. Moreover, many emerging mobile applications involve intensive computation based on data analytics, thus caching parts of computation results that are likely to be reused by others can further boost the computation performance of the entire MEC system. This motivates us to design optimal multi-user computation offloading strategies leveraging edge caching, as proposed in this thesis.

1.1.2 Socially-aware MEC:

Exploiting the social relationship of mobile users have emerged recently as a new topic for network design and optimization. In MEC network, a large number of users interacts with different operators, social service providers, and application gateways. It is thus extremely important to provide a trustworthy computation offloading environment. Based on the above concern, we will design socially-aware computation offloading strategies for MCA execution in MEC.

1.1.3 Learning-based computation offloading strategies in MEC:

In view of computation offloading in MEC, one important fact is that benefits and drawbacks from offloading computation-intensive portions of mobile applications can be influenced by various internal and external factors, such as application requirements, network conditions, and computing capabilities of mobile and external devices.

Prior studies have primarily focused on core mechanisms for offloading. Yet, adaptive scheduling in such systems is important because offloading effectiveness can be influenced by varying network conditions, workload requirements, and load at the target device.

We will present in this thesis a study on the feasibility of applying deep learning techniques to address the adaptive scheduling problem in mobile offloading framework.

1.2 MEC architecture

Mobile edge computing, which equips the radio access networks with cloud computing capabilities and hosts the application at the mobile network edge, has the potential to offer an improved user experience. Fig. 1.1 illustrates the basic architecture of MEC-enable cloud computing network. We can see that there are three basic components in MEC architecture: i) Mobile devices such as smart phones, smart sensors and intelligent vehicular, ii) MEC servers and iii) Wireless base stations (e.g., 3G/4G/5G, macro base stations and Wi-Fi access points) through which the mobile devices connect to the MEC servers.

Regarding MEC servers, they provide computing resource, storage capability and connectivity. They can either handle a user request and respond directly to the user equipment (UE) or forward the request to remote data centers and content distribution networks (CDNs). The server can be

deployed at high-end or low-end proximate in the small cell based edge cloud network [20]. The high-end deployment is a current focus of the ETSI Mobile-Edge Computing standard, as shown in Fig. 1.2 (b). In this case, typically a high-end standard server is located in the access network that is an aggregation point of a set of small cell base stations (SCeNB i.e., Small Cell e-Node B). The server is also regarded as a central coordinator that is required to design a cache placement strategy. On the other hand, in the low-end case, application servers can be deployed in low-end devices which can be routers, access points, or home-based nodes. Femto-cloud ([17, 21–23]) is a typical low-end deployment. The idea is to endow small cell base stations, with cloud functionalities (computation, storage and server), thus providing mobile UEs with proximity access to the mobile edge cloud. The novel base stations are called small cell cloud-enhanced e-Node B (SCceNB), as illustrated in Fig. 1.2 (a) , which are deployed with high-capacity storage units but have limited capacity backhaul links. In this case, the popular contents can be cached in a distributed manner among the SCceNBs without a central coordinator. With respect to the high-end deployment, this deployment brings three advantages: i) strong reduction of latency with respect to centralized clouds, because small cells are the closest points in the mobile network to the mobile users with only one wireless hop, and therefore with minimum latency, ii) storage of large amounts of local contents. Indeed, data can be temporarily stored (local caching) in the nearby SCceNB, with very low latency, and iii) no central coordinator is required to collect the information of the whole network, which significantly saves signalling overhead.

Compared to traditional centralized cloud computing architecture, the MEC architecture has several advantages listed below:

Reduced latency: Since the processing and storage capabilities are in proximity to the end users, the communication delay can be reduced significantly. Authors in [24] present experimental results from Wi-Fi and 4G LTE networks showing that offloading to cloudlets[1] can improve response times by 51% compared to cloud offloading.

Reduced bandwidth: Authors in [25] proved that through deploying at network edge, we can save operation cost by up to 67% for the bandwidth-hungry applications and compute intensive applications. Research results also show that backhaul savings can be up to 22% exploiting proactive caching scheme [26]. Higher gains are possible if the storage capability of edge servers are increased.

Energy efficiency: Experimental results in [24] showed that the energy consumption in a mobile device can be reduced by up to 42% through offloading to cloudlets compared to cloud offloading.

Proximity services: The architecture of mobile edge networks has great advantages in pro-

[1] A cloudlet is a mobility-enhanced small-scale cloud datacenter that is located at the edge of the Internet. The main purpose of the cloudlet is supporting resource-intensive and interactive mobile applications by providing powerful computing resources to mobile devices with lower latency.

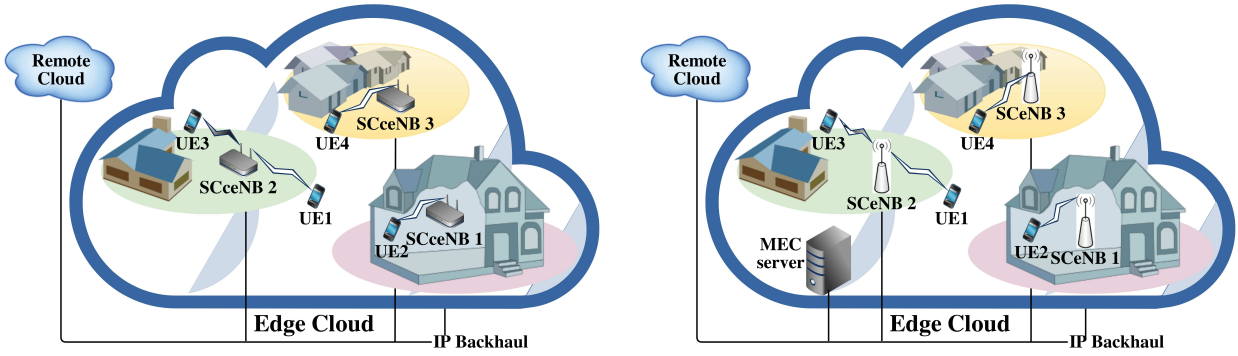


Figure 1.2 – The low-end and high-end deployments for 5G mobile edge cloud.

viding proximity services since the edge servers are closer to end users and D2D communication technology can be exploited. Therefore, the traffic load on radio access network can be reduced.

1.3 Key issues in 5G MEC

The key issues in 5G MEC are the following:

Emerging mobile collaborative applications: The new applications are the main driven force for the evolvement of network architecture. The emerging applications need mobile devices not only with considerable energy, memory size and computational resources, but also with collaboration capacity. For example, mobile collaborative applications require a close collaboration between the participants. The number of participants typically varies between only a few and up to a hundred. The collaboration takes place in a decentralized closed group, whereby each participant has a comprehensive knowledge of all the others. Therefore, these new types of applications bring several challenges to edge computing system deployment, particularly in terms of design, implementation, use, and evaluation.

Computation offloading in MEC: One of the main purposes of edge computing is computation offloading to break the limitations of mobile devices such as computational capabilities, battery resources and storage availability. When, where and how to offload the computation tasks is a hard problem. Various approaches have been proposed to tackle this problem under many kinds of scenarios such as single user case, multi user case etc. For the single user case, the common design objective is to save energy for the mobile device. Whereas in the multi user case, the optimal offloading problem is usually NP-hard. In next generation heterogeneous networks, the computation tasks can not only be offloaded to servers but also to devices by utilizing D2D communication (i.e., D2D computation offloading). Therefore, we can offload computation tasks to nearby mobile

devices, MEC servers and remote cloud servers in the MEC system, thus bring new challenges to offloading decision making.

Edge caching in MEC: The future mobile networks will be heterogeneous due to dense deployment of different types of base stations. Thus, cache can be deployed at various places in the mobile networks. In legacy cellular system, the content requested by users has to be fetched from the Internet CDN node far away from the mobile networks. Then, caching content at the mobile core network is implemented. However, the backhaul links are still constrained. In addition, with the evolvement of base station and low cost storage unit, deploying cache at macro base stations and small base stations become feasible. In the future 5G networks, D2D communication enables the storage unit at user devices to be exploited for content sharing according to the social relationship among users [27]. Thus storage resources in mobile devices can be exploited.

Socially-aware MEC: Exploiting the social relationship of mobile users have emerged recently as a new topic for network design and optimization. In MEC network, a large number of users interacts with different operators, social service providers, and application gateways. It is thus extremely important to provide a trustworthy environment. When considering the computation offloading in MEC, the trustworthy environment must provide reliable, secure, and personalized connectivity between resource (including computation and storage) seekers and resource providers. Trust management allows exhaustive utilization of network services by diversified users with mutual consent over social behaviours especially focusing on user privacy and confidentiality. Thus, it becomes important to consider trust-management for MEC.

1.4 Problem statement

In this thesis, we address the issue of multi-user computation offloading in the MEC. This challenging problem has been motivated by:

1. Resource intensive feature of the emerging mobile applications.
2. Collaborative requirements of the mobile collaborative applications.
3. Edge caching ability of the MEC servers that can boost the computation performance for mobile users.
4. Security and privacy consideration when mobile users make/receive offloading requirements.
5. Low complexity requirements for offloading decision making algorithm.

Consequently, collaborative computation offloading strategies are needed in order to enhance the mobile users' experience with edge caching and social relationship consideration. More specifically, the objective is to **minimize the cost** (i.e., delay and energy consumption for mobile application execution) within multiple mobile users in MEC network.

The optimal offloading decision making problem in hand is highly complex and is proved to be NP-hard [28]. Hence, the generation of the optimal solution within a polynomial time for a large-scale network is not possible. In view of this, we exploit the concepts of coalitional game, minimum weighted bipartite matching, monte carlo tree search and deep supervised learning for the algorithms design.

1.5 Thesis contributions

In this section, we summarize the significant contributions of this thesis:

1.5.1 Proposed an optimal offloading with caching-enhancement scheme (OOCS) for femto-cloud scenario and mobile edge computing scenario, respectively.

As a first contribution, we propose a fine-grained collaborative computation offloading and caching strategy that optimizes the offloading decisions on the mobile terminal side with caching enhancement. The objective is to minimize the overall execution latency for the mobile users within the network. Most of previous works either focus on the offloading decisions [10, 21–23, 28–33] or caching placement strategy [13, 34–36]. To the best of our knowledge, our work is the first of its kind that optimizes offloading decisions, while considering data caching.

Based on the concept of the call graph [37], we propose in this thesis the concept of the collaborative call graph to model the offloading and caching relationship for the Mobile Collaborative Applications (MCA) execution, and then compute the delay and energy overhead for each single user. We first evaluate our algorithm in a single-user scenario for both femto-cloud and MEC networks. Moreover, to further reduce the execution delay in the multi-user case, we explore the concept of the coalition formation game for the distributed caching scenario. We compare our approach with six benchmark policies from the literature and discuss the associated gains in terms of required cpu cycles per byte (cpb) for applications and content's popularity factor.

This contribution is the object of a conference publication in the International conference on communications (ICC) [38] and a submitted journal in IEEE Trans. on Vehicular Tech.

1.5.2 Proposed a device-to-device (D2D) multicast-based computation offloading strategy

As a second contribution, we propose in this thesis a flexible D2D multicast computation offloading framework in which the role of offloadees (servers) and offloaders (users) are negotiated within multiple mobile devices according to their channel state and battery level.

Our objective is to reduce the overall energy consumption at the mobile terminal side under delay and energy constraints. To this end, we modelled the optimal offloading framework as a com-

binatorial optimization problem, and then solved using the concepts of from maximum weighted bipartite matching and coalitional game.

This contribution is the object of a conference publication in the IEEE Global Communications Conference (GLOBECOM) [39]

1.5.3 Proposed a socially-aware hybrid (D2D/MEC) computation offloading (SAHCO) strategy

As a third contribution, we propose here a new fine-grained task assignment mechanism for MCA execution in Wireless Distributed Computing (WDC) with social trust consideration. The objective is to minimize the overall energy consumption at the mobile terminal side. Most of existing D2D offloading works either lack of social tie consideration [29, 30, 40], or consider the social tie in D2D communications ([41, 42], i.e., D2D data/traffic offloading). To the best of our knowledge, our work is the first of its kind that optimizes task assignment, while considering the social relationship. We propose in this thesis a new framework of social-aware D2D computation offloading (named *SAHCO*) and then compute the energy overhead for each application cluster. In order to enhance the performance, we solve the computation offloading problem for all the components of an application at the same time. In fact, while the sequential processing for components, minimizes the computation energy, it may fail to ensure an efficient optimized offloading decision. To this end, we propose a new optimal task assignment approach based on Monte-Carlo search tree (MCTS), named *TA-MCTS*. Our proposed solution achieves an optimized computation offloading policy. We compare our approach with the related benchmark policies from the literature, and discuss the associated gains in terms of mobile user density, social trust threshold and CPU structure, respectively.

This contribution is the object of a journal submission in IEEE Trans. on Mobile Computing.

1.5.4 Proposed a deep learning based low complexity computation offloading strategy

Last but not least, we propose a fine-grained computation offloading framework that minimizes the offloading cost for the MEC network. The offloading actions taken by a mobile user consider the local execution overhead as well as varying network conditions (including wireless channel condition, available communication and computation resources). Most of previous machine learning-based offloading works consider either coarse-grained computation offloading [43, 44], or assume an infinite amount of available communication or computation resources in cloudlet.

In this context, we formulate and build a Deep Supervised Learning (DSL) model to obtain an optimal offloading policy for the mobile user. Our method provides a pre-calculated offloading solution which is employed when a certain level of knowledge about the application and network conditions is achieved. We formulate the continuous offloading decision problem as a multi-label

classification problem. This modelling strategy largely benefits from the emerging deep learning methods in the artificial intelligence field. Our method approaches the optimal solution obtained by the exhaustive strategy performance with a very subtle margin. As the number of fine-grained components n of an application grows, the exhaustive strategy suffers from the exponential time complexity $O(2^n)$. Fortunately, the complexity of our learning system is only $O(mn)^2$, where m is the number of neurons in a hidden layer which indicates the scale of the learning model. In fact, the artificial intelligence problem with both large input/output dimension and large number of examples such as image classification is proved experimentally feasible using similar technique [45]. To the best of our knowledge, our work is the first of its kind that achieves optimal offloading policy through Deep Learning approach.

This contribution is the object of a conference publication in the International symposium on personal, indoor and mobile radio communications (PIMRC) [46].

1.6 Thesis outline

This thesis is organized as follows. In Chapter 2, we outline the background and review of literature review. Then, we the models that are used in this thesis. In Chapter 3, we consider both high-end and low-end MEC offloading scenarios. We propose an optimal offloading with caching-enhancement scheme (OOCS) for femto-cloud scenario and mobile edge computing scenario, respectively. In Chapter 4, we first focus on the D2D offloading scenario. To this end, we propose a D2D multicast-based computation offloading framework where the problem is modelled as a combinatorial optimization problem, and solved using the concepts of from maximum weighted bipartite matching and coalitional game. Then, we extend our work to hybrid offloading scenario. We formulate a fine-grained task assignment mechanism for MCA execution in Wireless Distributed Computing (WDC) with social trust consideration. We propose a monte carlo tree search based algorithm, named, *TA-MCTS* for the task assignment problem. Then in Chapter 5, we formulate the offloading decision problem as a multi-label classification problem and develop a Deep Supervised Learning (DSL) method to solve it. Finally, Chapter 7 concludes this thesis and presents our short and long term future work in the field.

Chapter 2

Background and Literature Review

Contents

2.1 Introduction

In order to save the cost at the mobile terminal side in the MEC network, designing an efficient computation offloading strategy is the first and foremost task. In this chapter, we will give an overview of the different aspects that should be concerned when designing an efficient computation offloading strategy. Specifically, we will summarize the related computation offloading strategies

and review the computation offloading strategise in MEC. Thereafter, we will outline the application scenario. We take mobile collaborative applications and mobile crowd sensing applications as examples. Section 2.4 deals with the related edge caching deployment strategies. For this, we classify the edge caching strategies into two groups: i) service caching, and ii) data caching. Finally, section 2.6 concludes this chapter.

2.2 Computation offloading

In order to meet the low latency and improved QoE requirements of emerging applications, we need code offloading and local caching strategies that empower mobile applications with resourceful cloud equipments. Offloading mechanisms have been studied extensively [31–33, 47–51] and generally fallen into two categories: coarse-grained/full offloading [31] and fine-grained/partial (typically at a method-level granularity) offloading [32, 33, 47–51].

2.2.1 Coarse-grained/full offloading

Coarse-grained offloading also refers to the full offloading or total offloading in which full task is migrated to the cloud. This approach does not require estimating the computation overhead prior to the execution. For example, authors in [31] proposed a computational offloading policy to decide whether an entire application should be offloaded to remote cloud or executed locally to reduce energy consumption on the mobile terminal side. The main objective of the exist works focused on the coarse-grained offloading decision are i) minimize an execution delay, ii) minimize energy consumption at the UE while predefined delay constraint is satisfied and iii) find a proper trade-off between both the energy consumption and the execution delay.

One of the advantages introduced by the computation offloading to the MEC is a possibility to reduce the execution delay. In case of the computation offloading to the cloudlet, the execution delay incorporates three following parts: i) transmission duration of the offloaded data to the cloudlet, ii) computation/processing time at the cloudlet, and 3) time spent by reception of the processed data from the cloudlet. To minimize execution delay, Liu et al. proposed a full offloading strategy in [52]. This is accomplished by one-dimensional search algorithm, which finds an optimal offloading decision policy according to the application buffer queuing state, available processing powers at the UE and at the MEC server, and characteristic of the channel between the UE and the MEC server. The computation offloading decision itself is done at the UE by means of a computation offloading policy module. This module decides, during each time slot, whether the application waiting in a buffer should be processed locally or at the MEC while minimizing the execution delay.

The computation offloading decision minimizing the energy consumption at the UE while satisfying the execution delay of the application is proposed in [53]. The optimization problem is formulated as a constrained Markov decision process (CMDP). To solve the optimization problem,

two resource allocation strategies are introduced. The first strategy is based on an online learning, where the network adapts dynamically with respect to the application running at the UE. The second strategy is pre-calculated offline strategy, which takes advantage of a certain level of knowledge regarding the application (such as arrival rates measured in packets per slot, radio channel condition, etc.). The numerical experiments show that the pre-calculated offline strategy is able to outperform the online strategy by up to 50% for low and medium arrival rates (loads).

The computation offloading decision for the multi-user multi-channel environment considering a trade-off between the energy consumption at the UE and the execution delay is proposed in [28]. Whether the offloading decision prefers to minimize energy consumption or execution delay is determined by a weighing parameter. The main objective of the paper is twofold; i) choose if the UEs should perform the offloading to the MEC or not depending on the weighing parameter and ii) in case of the computation offloading, select the most appropriate wireless channel to be used for data transmission. To this end, the authors present an optimal centralized solution that is, however, NP-hard in the multi-user multi-channel environment. Consequently, the authors also propose a distributed computation offloading algorithm achieving Nash equilibrium.

2.2.2 Fine-grained/partial offloading

Fine-grained offloading (partial offloading or dynamic offloading) dynamically transmits as little code as possible and offloads only the computation-hungry parts of the application. We classify the research on exist works focused on i) minimization of the energy consumption at the UE while predefined delay constraint is satisfied and ii) works finding a proper trade-off between both the energy consumption and the execution delay.

Authors in [33] investigated the problem of collaborative task execution by strategically offloading task to the cloud. They proposed a fine-grained offloading strategy to reduce energy consumption under a latency constraint. Despite the introduced burden on the application programmers and additional computational overhead, the proposed approach can reduce unnecessary transmission overhead, achieving a reduced latency and energy consumption. Cuervo et al in [47] propose a framework called MAUI, which focuses on energy saving. It uses a profiler which measures energy consumption and a solver which decides whether to offload or not a method based on the measurement provided from the profiler. They evaluate MAUI using three mobile applications and revealed that the computation offloading not only saves energy, but also that it allows applications to run faster.

Chun et al in [48] propose CloneCloud, which partitions the application binary with a set of execution points. The execution points are determined so that the resulting partitions are executed in the most efficient execution environment. As a result, CloneCloud can determine the most efficient execution points and execution configuration for each partition. Kemp et al in [49] propose Cuckoo, a simple offloading framework that always offloads a method when the remote server is

Table 2.1 – Comparison between state of the art computation offloading frameworks

Framework	Main goal	Fine-grained	Code profiler	Intrusiveness	OS/Language
MAUI [47]	Energy saving	Yes	Manual annotations	Low	Win/C#
CloneCloud [48]	Transparent code migration	Yes	Automated process	Runtime modification	Android/Java
Cuckoo [49]	maximize computation speed or minimize energy usage	No	Manual annotations	Development in AIDL	Android/Java
ThinkAir [50]	Scalability	No	Manual annotations	Runtime modification	Android/Java
COSMOS [51]	Both improve offloading performance and reduce the monetary cost	No	Automated process	Code modification	Android/Java
ULOOF [54]	Minimize remote execution overhead	No	Automated process	Low	Android/Java

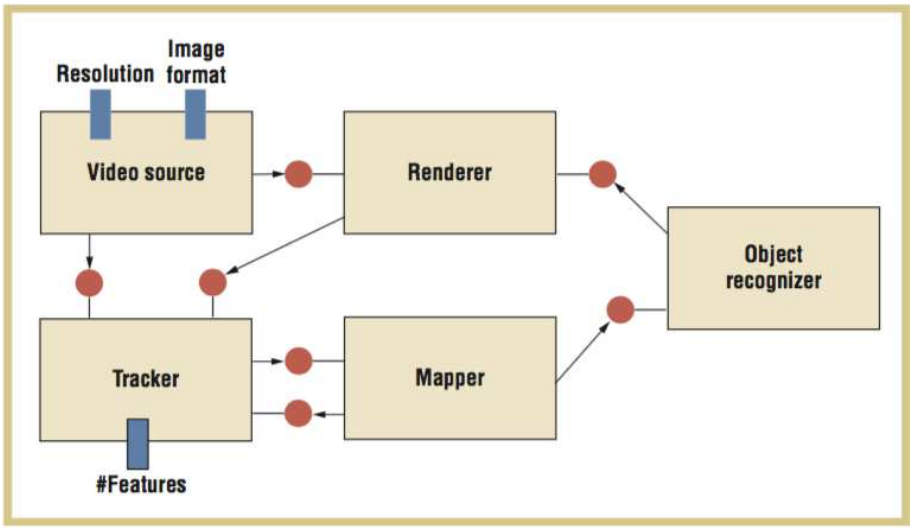


Figure 2.1 – Identifying loosely coupled software components in an example immersive application [1].

available. Cuckoo implements a library to manage the communication between the mobile device and a communication middleware. Kosta et al in [50] propose ThinkAir. It generates a wrapper for methods to be offloaded so that an execution controller decides whether to offload the method based on execution time, energy and cost, or not. They modelled the execution time and energy using historical data from previous executions. A client handler manages the connection to the remote Virtual Machine (VM) and is also responsible for managing the VM configuration. Shi et al in [51] propose the COSMOS framework; it determines the benefit of offloading based on argument size, upload bandwidth, result size and download bandwidth with a predefined threshold. The result of each offloaded execution is then adjusted when the upload and download bandwidth information is updated. Table.2.1 summarises the state of the art computation offloading frameworks.

2.3 Mobile application model

Mobile collaborative applications (MCA) [1–4], such as mobile audio/video conferences, collaborative writing, social networks, and mobile gaming, are already important parts of human interaction. MCA play an increasing role in mobile communications in order to enable cooperation among mobile and/or stationary participants - often using the device-to-device (D2D) group communication paradigm.

In this section, we first introduce the call graph. Then, to better understand the functionalities of this type of applications, let us present two kinds of typical MCA, named mobile augment reality application (MARA) and mobile crowd sensing (MCS) application.

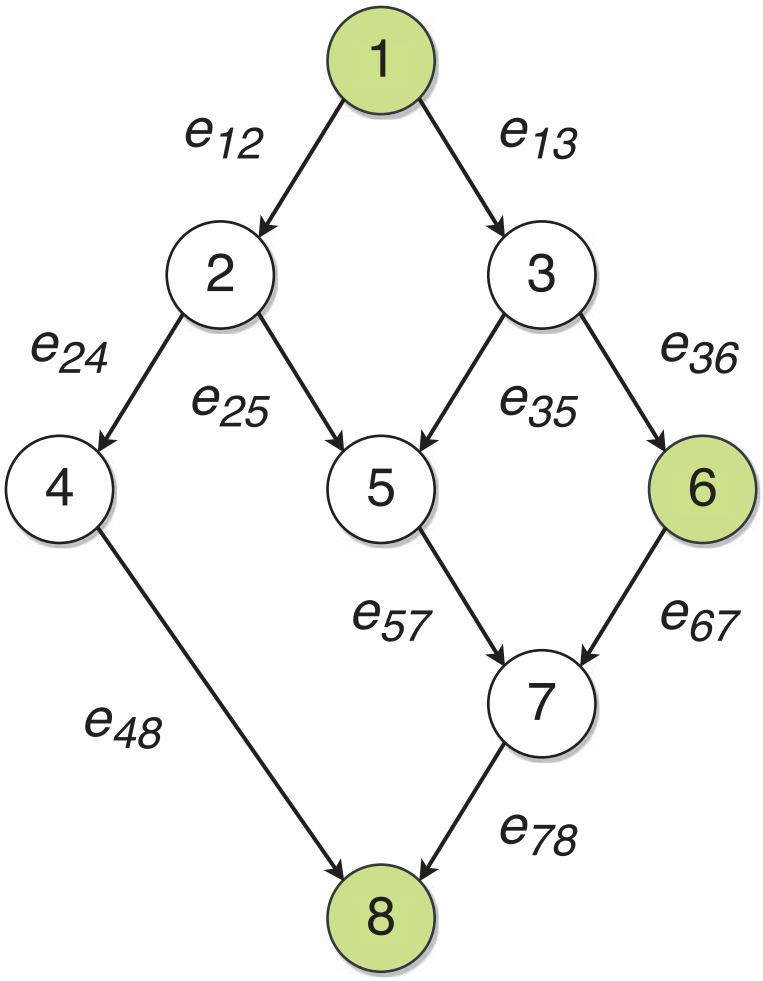


Figure 2.2 – An example of task graph for mobile wearable application [56].

2.3.1 Call graph

A call graph (also known as task graph) is a control flow graph, which represents calling relationships between components/subtasks in a mobile application. Typically, it consists in modeling the relationship between components as a weighted directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ denotes the set of components, and $\mathcal{E}$ represents the data dependencies between components. Each edge $\mathcal{E}_{u,v} \in \mathcal{E}$ indicates the data communication (i.e., computation result) required by component v to be fed from component u. In addition, there are three typical dependency models of components, namely i) sequential, ii) parallel and iii) general dependency [55]. To meet the requirements of most emerging mobile applications, we consider the general dependency model in this thesis.

Fig. 2.2 illustrates a task graph of a mobile wearable application [56] with general dependency. Note that the dependency among different components cannot be ignored as it significantly affects

Figure 2.3 – By sharing components, multiple devices can track and expand the same map and recognize the same objects [1].

the procedure of execution and computation offloading. In fact, the outputs of some components may be used as inputs for others, and hence the execution order of functions can not be arbitrarily chosen. For example, as shown in Fig. 2.2, component 8 can not be executed until it receives the outputs of components 4 and 7 (i.e., $e_{4,8}$ and $e_{7,8}$).

On the other hand, due to either software or hardware constraints, some components can be offloaded to the server for remote execution, while other ones can only be evaluated locally. For example, components 1, 6 and 8 in Fig. 2.2 denote the user input, picture capture and result display, respectively, that must be executed on wearable devices [56].

2.3.2 Mobile augment reality applications

A mobile augmented reality application (MARA) is a type of mobile application that incorporates and complements built-in components in a mobile phone and provides a specialized application to deliver reality-based services and functions. MARA uses the architectural composition of a mobile phone to deliver applications that add value to the physical world through virtual data and services. In multi-user MARA scenario, multiple users in the same neighborhood typically look at the same scene, track the same environment, and need to recognize the same objects, so they can benefit from collaboration and data/resource sharing [2]. For example, authors in [1,3] proved that MARA have the unique property that the applications of different users share part of the computational tasks and of the input and output data. Authors in [1] proposed a component-based offloading framework that optimizes application-specific metrics. They split an immersive application into several

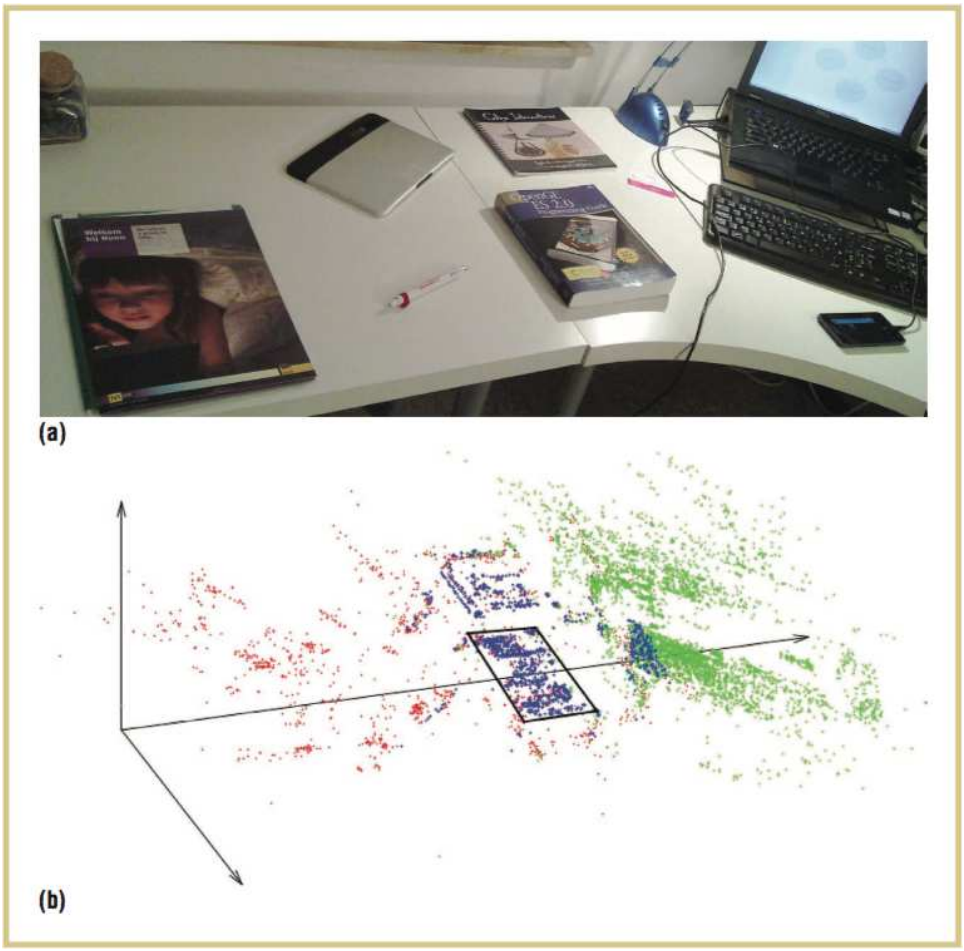


Figure 2.4 – Three devices explore (a) the top of a desk, resulting in (b) mapped feature points and a recognized object. by sharing the detected feature points of all devices, more feature points from multiple viewpoints are added to the map, leading to more robust tracking and faster map expansion [1].

loosely coupled software components as shown in Fig. 2.1. Each components with its dependency, configuration parameters and constraints can be offloaded and shared among multiple users. By adopting the component-based approach, collaborative scenarios can easily be constructed by sharing application components. In the parallel tracking and mapping use case, for example, the Mapper component can be shared between multiple mobile devices in the same physical environment. This way, all devices receive the same world model to track the camera position. Because the model is updated and refined using camera images from multiple devices, the model will often be more accurate than one created by just one device. For example Fig. 2.3 shows an immersive application, where two devices share the same Mapper and Object Recognizer, which are offloaded to the laptop. The map of the environment is generated twice as fast (because both devices send

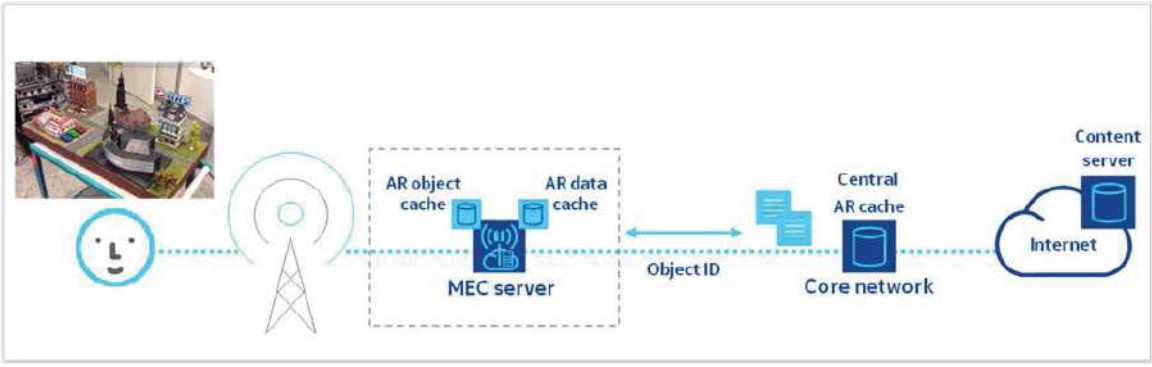


Figure 2.5 – MEC-enabled augmented reality scenario [18].

updates to the Mapper), and processing is performed only once (instead of twice-once for each device). Using multiple devices to add feature points to the map also leads to a more complete map of the environment. Each device now has a complete overview of the environment, whereas in the noncollaborative case, the map is limited to the device's own viewpoint. Figure 2.4 shows a map of a desk environment generated with three devices. Each device observes the desk from a different viewpoint, so each maps different (overlapping) parts of the desk. Through this component-based framework, the cloudlet agent allows users to not only save computational resources but also to gain information from the input of others.

In addition, the use of a MEC server is highly advantageous since augmented reality information is highly localized. Fig. 2.5 illustrates a ETSI MEC-enable augmented reality scenario [18], the processing of user location or camera view can be performed on a MEC server rather than on a more centralized server. There may be a need to update information at a fast rate, depending on how the user moves, and the context in which the augmented reality service is used (e.g. in an art gallery, exhibits are positioned only a few metres apart and each piece is supplemented with additional text on the artist, the interpretation of the artwork, etc.). In other words, augmented reality data requires low latency and a high rate of data processing in order to provide the correct information to the user's device depending on the location and orientation of the user.

2.3.3 Mobile crowd sensing applications

Mobile crowd sensing (MSC) is another resource-intensive application, where individuals with sensing and computing devices collectively share data and extract information to measure and map phenomena of common interest [57]. MSC applications can be classified into two categories: i) personal sensing and ii) community sensing, based on the type of phenomena being monitored. In personal sensing applications, the phenomena is pertaining to an individual. For example, the monitoring of movement patterns (e.g. running, walking, exercising) of an individual for personal

record-keeping or healthcare reasons. Whereas community sensing pertains to the monitoring of large-scale phenomena that cannot be easily measured by a single individual. For example, intelligent transportation systems may require traffic congestion monitoring and air pollution level monitoring.

Currently, a typical MCS application has two application specific components, one on the device (for sensor data collection and propagation) and the second in the MCS server (or cloud) for the analysis of the sensor data to drive the MCS application. When mobile devices process MSCs, the OS of mobile devices allows applications to access the sensors and extract raw sensing data from them. However, depending on the nature of the raw data and the needs of applications, the physical readings from sensors may not be suitable for the direct consumption of applications. Many times, some local analytics performing certain primitive processing of the raw data on the device are needed. The motivation of such local analytics are based on the following three reasons. First, the kind of processing performed leads to appropriately summarized data, thus consuming lesser energy and bandwidth than transmitting the raw sensor readings. This is a well-known local execution in conventional computation offloading scenario: using computation to save energy/bandwidth. Second, it reduces the amount of processing that the MCS server has to perform. Further, if the mobile devices in a societal scale deployment transmit raw sensor data, the MCS server can easily be overwhelmed. Finally, some applications are delay sensitive and transmitting raw sensor data on intermittently connected channels can be time consuming as compared to that of sending processed sensor data.

2.4 Edge caching in MEC

Nowadays, wireless caching (i.e., caching content within the wireless access network) is gaining interest, especially in ultra-dense networks where many connected devices try to access various network services under latency, energy efficiency, and/or bandwidth limitation constraints. On the other hand, many emerging mobile applications involve intensive computation based on data analytics, thus caching parts of computation-result data that is likely to be reused by others can further boost the computation performance of the entire MEC system [55]. One typical example is mobile cloud gaming [6]. Note that certain game rendered videos, e.g., gaming scenes, can be reused by other players, caching these computation results would not only significantly reduce the computation latency of the players with the same computation request, but also ease the computation burden for edge servers. Consider the novel cache-enabled MEC system [55], the MEC server can cache several application services and their related database, called service caching and data caching, respectively, and handle the offloaded computation from multiple users. In the high-end MEC case, massive data are cached in a centralized manner. Thus the key problem is how to balance the trade-off between massive database and limited storage capacity of MEC server. Whereas in the low-end

MEC case, the design principle is how to cache the massive data in the distributed low-end MEC servers. Furthermore, it is also essential to establish a practical database popularity distribution model that is able to statistically characterize the usage of each database set for different MEC applications. In this section, we will introduce the service caching and data caching, respectively. At last, we will formulate our caching model that is used in this thesis.

2.4.1 Service caching

Unlike the central cloud server that is always equipped with huge and diverse resources (computation and storage), the current edge server has much less resources, making it unable to accommodate all users' computation requests. On the other hand, different mobile services require different resources. Such a mismatch between resource and demand introduces a key challenge on how to allocate heterogeneous resources for service caching. To this end, authors in [55] proposed two possible approaches for the resource allocation problem in cache-enabled MEC system. The first one is spatial popularity-driven service caching, referring to caching different combinations and amounts of services in different MEC servers according to their specific locations and surrounding users' common interests. This idea is motivated by the fact that users in one small region are likely to request similar computing services. For example, visitors in a museum tend to use mobile augment reality (AR) application for better sensational experience. Thus, it is desirable to cache multiple AR services at the MEC server of this region for providing the real-time service. The second approach is temporal popularity-driven service caching. The main idea is to exploits the popularity information in the temporal domain, since the computation requests also depend on the time period. One example is that users are apt to play mobile cloud gaming after dinner. This kind of information will suggest MEC operators to cache several gaming services during this typical period for handling the huge computation loads. One disadvantage of this temporal-based approach is the additional server cost resulted from frequent cache-and-tear operations since popularity information is time-varying and MEC servers possess finite resources.

2.4.2 Data caching

Data caching refers to caching related data (libraries/databases) of application services in the edge server, e.g. MEC-enabled Base Station (BS), enabling corresponding computation tasks to be executed. Take virtual reality (VR) application [58] as an instance. It creates an imaginary environment similar to the real world by generating realistic images, sounds and other sensations for enhancing users' experience. Achieving this end is nontrivial as it requires the MEC server to finish multiple complicated processes within the ultra-short duration (e.g., 1ms), such as recognizing users' actions via pattern recognition, understanding users' requests via data mining, as well as rendering virtual settings via video streaming or other sensation techniques [164]. All the above data-analytics based

techniques should be supported by comprehensive database, which, however, imposes extremely heavy burden on the edge server storage. This challenge can be relieved by intelligent data caching that only reserves frequently-used database. Therefore, for MEC data caching at a single edge server, one key problem is how to balance the tradeoff between massive database and finite storage capacity.

2.5 Summary and discussion

This chapter first gave an overview of current mobile computation offloading environments. Based on the fact that mobile hardware remains restricted due to form factor constraints. This motivates our work on computation offloading. We began with a discussion of the coarse-grained/full offloading scenario. We next described the fine-grained/partial offloading scenario, and discussed the exist works. Then, we investigated the application model for mobile computation offloading and discussed two typical application scenario. At last, we discussed the role of edge caching in MEC and investigated the service caching and data caching, respectivly. We found that most of previous works either focus on the offloading decisions [10, 21–23, 28–33] or caching placement strategy [13, 34–36]. This motivates us to optimizes offloading decisions, while considering edge caching.

2.6 Conclusion

This chapter provided an overview of the key issues in offloading decision making in MEC network. First, we summarized the state of the art computation offloading strategies. Then, we outlined the application model and introduced two kinds of typical MCA. At last, we deal with the related edge caching deployment strategies.

Chapter 3

Computation Offloading with Caching-Enhancement for Mobile Edge Computing

Contents

3.1 Introduction

As stated in the previous chapter, computation offloading is a proven successful paradigm for enabling resource-intensive applications on mobile devices. Moreover, in view of emerging applications, the offloaded tasks can be duplicated when multiple users are in the same proximity, which motivates to cache the popular computation results. In this chapter, we will investigate collaborative computation offloading with the data caching enhancement strategy for the MCA execution in femto-cloud and MEC, respectively. Our objective is to reduce the average execution delay for the mobile users within the network. Our proposal can dramatically reduce the MCA's execution latency based on the following two facts:

- Multiple mobile users in the same MCA execution environment can share computation and outcome results. Indeed, through sharing the same components (e.g., Mapper and Objective Recognizer as in [1]) and computation results (e.g., detected feature points of environment), the environment will be generated faster and more complete. Through this kind of cooperation and sharing, mobile users can not only save computational resources but also to gain information from the input of others.

- For cache-enabled MEC (e.g., mobile cloud gaming), caching parts of computation results that are likely to be reused by others can further boost the computation performance of the entire system [55]. This idea is motivated by the fact that users in one small region are likely to request similar computing services. For example, visitors in a museum tend to use Augmented Reality (AR) for better sensational experience. Thus, it is desirable to cache multiple AR services and output data at the MEC server of this region to provide the real-time services.

To this end, we propose an optimal offloading with caching-enhancement scheme (OOCS) for femto-cloud scenario and mobile edge computing scenario, respectively.

This chapter is organized as follows. We first introduce our system model in Section 3.2. Then, in Section 3.3, we introduce the cooperative call graph and formulate the optimaztion problems, followed by a description of our proposed algorithm in Section 3.4. Simulation results are presented in Section 3.5. Finally, conclusions are drawn in Section 3.6.

3.2 System Model

In this section, we present the model of our optimal offloading with caching-enhancement scheme (OOCS).

3.2.1 Network Model

In this thesis, we consider both high-end and low-end deployments of small cell-based mobile edge cloud. In the case of low-end deployment, we consider an LTE femto-cloud network composed of N_l small cells with computation and memory enhancement (i.e. SCceNBs in LTE terminology). In the case of high-end deployment, we consider an edge cloud network consisting of N_h traditional small cell base stations (SCeNBs) and a centralized high-end server. The number of mobile users for both network types is M. We also consider both of the network deployments are based on the orthogonal frequency division multiple-access (OFDMA) in which M users within the same SCeNB/SCceNB are separated in the frequency domain. Note that, using such a transmission scheme for the uplink offloading implies that the users do not interfere with one another.

Let p_u and p_s denote the transmit power for the UEs and small cells (i.e., SCeNBs and SCceNBs), respectively. The maximum achievable rate [59, 60] (in bps) over an additive white Gaussian noise (AWGN) channel for user m ($m \in \mathcal{M}$) to offload its application to small cell n ($n \in \mathcal{N}_h$ or $n \in \mathcal{N}_l$) can be expressed as follows:

$$r_{n,m}^{ul} = B \log_2 \left(1 + \frac{p_u |h_{ul}|^2}{\Gamma(g_{ul}) d^{\beta} N_0} \right), \tag{3.2.1}$$

where B is the bandwidth, d is the distance between UE and SCeNB/SCceNB. In this paper, we consider the Rayleigh-fading environment, and h_{ul} and h_{dl} are the channel fading coefficient for uplink and downlink, respectively. N_0 denotes the noise power and β is the path loss exponent. Note that $\Gamma(BER) = -\frac{2log(5BER)}{3}$ represents the SNR margin introduced to meet the desired target bit error rate (BER) with a QAM constellation [61]. g_{ul} and g_{dl} are the target BER for uplink and downlink, respectively.

Similarly, the maximum achievable rate (in bps) for a user m receiving its computation results from small cell n ($n \in \mathcal{N}_h$ or $n \in \mathcal{N}_l$) is given by:

$$r_{n,m}^{dl} = B \log_2 \left(1 + \frac{p_s |h_{dl}|^2}{\Gamma(g_{dl}) d^{\beta} N_0} \right). \tag{3.2.2}$$

3.2.2 Application Model

We assume that a mobile application can be split into multiple components [1] which in the granularity of either method [47–51] or thread [31] (i.e., a fine-grained partitioning). We then exploit the concept of the call graph [37], which consists in modeling the relationship between components as a weighted directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ denotes the set of components, and $\mathcal{E}$ the data dependencies between components. Fig. 3.1 represents an example of two call graphs for an immersive application [1], which shows two users (UE1 and UE2) offloading their components to the same edge server individually. The immersive application can be split into different components.

From the example illustrated in Fig. 3.1, there are 5 different components: Interface, Tracker, Mapper, Recognizer, and Renderer [1]. The Tracker component denotes the input camera frames with delay constraints. Such a component is used to estimate the camera position. Note that, some of these components cannot be offloaded such as the user's interface and renderer in Fig. 3.1, and must necessarily be evaluated locally. The components in the edge server denote the component clones for the components of mobile users.

Here, we consider M UEs equipped with the same mobile device, so the local energy consumption and latency are same among the UEs when executing the same component. As stated earlier, $\mathcal{E}$ denotes the set of weight for all the edges, we assume each edge $\mathcal{E}_{u,v}$ ($\mathcal{E}_{u,v} \in \mathcal{E}$) represents the data communication (computation result) between two components. We let ϕ_v ($v \in \mathcal{V}$) denotes the weight of component v, which specifies the workload (CPU cycles) for the component v. For a given input data size $\mathcal{E}_{u,v}$, ϕ_v can be derived from [62, 63] as $\phi_v = \omega \cdot \mathcal{E}_{u,v}$, where the ω in CPU cycles/byte (cpb) [64] indicates the number of clock cycles a microprocessor will perform per byte of data processed in an algorithm. The parameter depends on the nature of the component, e.g., the complexity of the algorithm. The estimation of this value has been studied in [65, 66] which is thus beyond the scope of our work.

3.2.3 Caching Model

As stated earlier, a library consists of $|\mathcal{E}|$ computation results, denoted by $\mathcal{E} = \{\mathcal{E}_{u,v}, (u, v \in \mathcal{V})\}$, is cached at one or multiple edge servers which can be either in high-end or low-end deployment. Let $|\mathcal{E}_{u,v}|$ denotes the size (in bytes) of $\mathcal{E}_{u,v}$. The computation result's popularity distribution conditioned on the event that user m makes an offloading request for its current component v. We also assume that each edge server has a finite-capacity storage storing part of the popular computation results. Specifically, we denote by Q_h the storage capacity of the high-end server, and by Q_l the storage capacity of the low-end SCceNB. In this work, we consider the popularity-based caching policy [34, 36] that the edge servers store computation results based on their highest popularity until the storage capacity is achieved. Thus, offloading requests is ranked from the most popular to the least, such that the request probability for a computation result $\mathcal{E}_{u,v}$ is:

$$f(\mathcal{E}_{u,v}, \delta, |\mathcal{E}|) = \frac{1}{\mathcal{E}_{u,v}^{\delta}} \sum_{n=1}^{|\mathcal{E}|} \frac{1}{n^{\delta}} \tag{3.2.3}$$

where δ models the skewness of the popularity profile. For $\delta = 0$, the popularity profile is uniform over files, and becomes more skewed as δ grows larger [34].

3.2.4 Execution Model

As stated earlier, we focus in this paper on both low-end and high-end deployments for small cell-based mobile edge cloud. For low-end deployment, each component can be executed either on the

mobile device or offloaded to a SCceNB. Whereas for high-end deployment, each component can be executed either on the mobile device or offloaded to the centralized server through a nearby SCeNB. The offloading decision is based on the workload of components $\mathcal{V}$, data communication $\mathcal{E}$, data rate $r^{dl}_{n,m}$ and $r^{ul}_{n,m}$. In this context, we consider the model of local execution, low-end execution and high-end execution, respectively.

3.2.4.1 Local Execution

If the component v is executed on mobile device, the completion time is $t^{local}_v = \phi_v \cdot f_m^{-1}$, and the corresponding energy consumption of the mobile device is $E^{local}_v = p_c \cdot t^{local}_v$, where f_m denotes the CPU rate of mobile devices (Million Instructions Per Second, MIPS), p_c represents the computing power for mobile device, ϕ_v denotes the number of instructions to be executed for component v.

3.2.4.2 Low-end Execution

If component v is offloaded to a Virtual Machine (VM) [24] in a SCceNB, the mobile device is idle before receiving the computation results. We denote $t^{low}_v = \frac{q_l \cdot \phi_v}{f_s}$ as the completion time of component v executed on the SCceNB, where f_s denotes the CPU rate of SCceNB ($f_s > f_m > 0$). Due to the limited computation capacity of SCceNBs, we assume that each SCceNB can serves at most q_l mobile users. The corresponding energy consumption of the mobile device during the remote execution is $E^{low}_v = p_i \cdot t^{low}_v$, where p_i is the idle power for mobile devices.

When component v is offloaded to SCceNB and the input data $\mathcal{E}_{u,v}$ from its previous component u is stored locally (i.e., stored in the mobile device), $\mathcal{E}_{u,v}$ must be sent to SCceNB before the execution of component v. We define this procedure as **Sending Input Data (SID)** for component v. Therefore, the time for UE m sending input data from component u to component v in low-end SCceNB n is $t\_s\_low^{u,v}_{n,m} = \frac{|\mathcal{E}_{u,v}|}{r^{ul}_{n,m}}$, and the corresponding energy consumption for UE m during the **SID** time is $E\_s\_low^{u,v}_{n,m} = p_u \cdot t\_s\_low^{u,v}_{n,m}$.

Conversely, if component v is executed locally, and its previous component u is executed in SCceNB, the output data $\mathcal{E}_{u,v}$ of component u must be sent back to mobile device before the execution of component v. We define this procedure as **Receiving Output Data (ROD)** for component v. Therefore, the delay for UE m receiving output data from component u to component v in low-end SCceNB n is $t\_r\_low^{u,v}_{n,m} = \frac{|\mathcal{E}_{u,v}|}{r^{dl}_{n,m}}$ and the corresponding energy consumption for UE m during the **ROD** time is $E\_r\_low^{u,v}_{n,m} = p_i \cdot t\_r\_low^{u,v}_{n,m}$.

3.2.4.3 High-end Execution

Then, we analyze the task execution for high-end deployment. When component v is offloaded to a VM in a centralized high-end server. We denote by $t^{high}_v = \frac{q_h \cdot \phi_v}{f_c}$ the completion time of component v executed on the centralized high-end server, where f_c denotes the CPU rate of the

high-end server ($f_c > f_s > f_m > 0$) and q_h the maximum number of UEs that the server can serve. The corresponding energy consumption for the mobile device during the remote execution is $E_v^{high} = p_i \cdot t_v^{high}$.

Similarly, when component v is offloaded to the high-end server whereas the input data $\mathcal{E}_{u,v}$ from its previous component u is stored locally, the **SID** delay for UE m sending input data from component u to component v in high-end deployment is $t\_s\_high_m^{u,v} = \frac{|\mathcal{E}_{u,v}|}{r_{n,m}^{ul}} + t_e$, where t_e is the end-to-end delay from SCeNBs to the high-end server. The corresponding energy consumption for UE m during the **SID** time is given as follows:

$$E\_s\_high_m^{u,v} = p_u \cdot \frac{|\mathcal{E}_{u,v}|}{r_{n,m}^{ul}} + p_i \cdot t_e. \tag{3.2.4}$$

Conversely, if component v is executed locally and its previous component u is executed in the server, the **ROD** delay for component v of UE m receiving output data from component u in high-end deployment is given by

$$t\_r\_high_m^{u,v} = t_e + \frac{|\mathcal{E}_{u,v}|}{r_{n,m}^{dl}}, \tag{3.2.5}$$

and the corresponding energy consumption for UE m during the **ROD** time is $E\_r\_high_m^{u,v} = p_i \cdot t\_r\_high_m^{u,v}$.

Note that, when components u and v are processed on the same side (i.e., both processed locally or remotely), the **SID** and **ROD** delay as well as the energy consumption are equal to zero in both of the deployments. This is based on the fact that the parameters passing on the same side do not involve wireless communication, and hence the overhead is not significant. In addition, we do not consider the transmission energy consumption from SCeNB/SCceNB to UE. We only consider the idle energy consumption for mobile users when its serving SCeNB/SCceNB sending the computation results back. In this work, the latency and energy consumption for decoding at the server side are assumed to be negligible for the following reasons: (i) compared with mobile devices, low-end and high-end servers are usually equipped with more powerful processors, and (ii) the servers are usually located at fixed areas with stable power supply.

3.3 Proposed collaborative call graph and problem formulation

In this section, we first propose a concept of collaborative call graph to model the cooperative offloading and caching problem within multiple mobile users, and then formulate the optimization problems for high-end and low-end deployments, respectively.

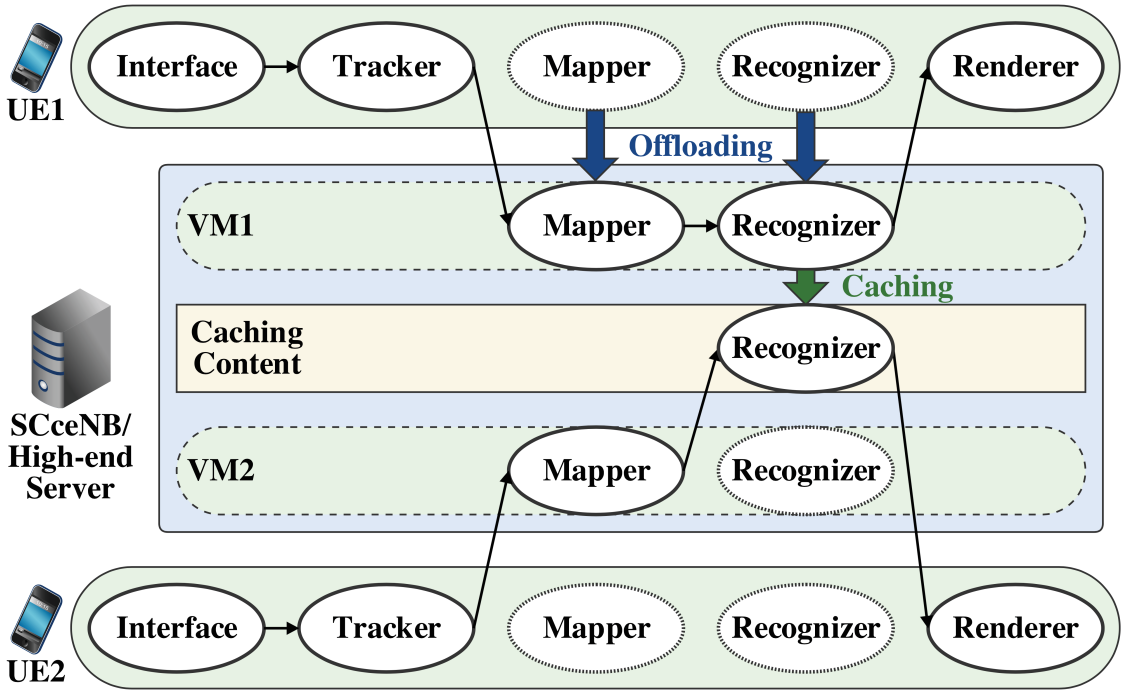


Figure 3.1 – Collaborative call graph with caching enhancement.

3.3.1 Collaborative call graph with caching enhancement

We propose a concept of the collaborative call graph for multi-user MCA execution scenario, as shown in Fig. 3.1. When a group of UEs connect to the same edge server (high-end server or SCceNB), they can cooperate through sharing their input data and computation results in the server. For example, Fig. 3.1 shows that UE1 and UE2 offload their computation to the edge server and share the corresponding computation results. They can benefit from the fact that the result of component "Recognizer" is already cached in the edge server to reduce the execution latency. Or if there is no such result cached in the edge server, they can collaboratively execute the component in the edge server for one time, instead of two times separately.

3.3.2 Problem formulation for Low-end MEC deployment

3.3.2.1 Single-user Scenario

We first consider the single-user low-end deployment case, where UE m ($m \in \mathcal{M}$) offloads its components to a low-end server (i.e., SCceNB) n ($n \in \mathcal{N}_l$). In addition, we define $T^l_{n,m,v}$ as the offloading execution delay for component v which is given by the sum of **SID/ROD** period of its previous component u and its execution period. Whereas $E^l_{n,m,v}$ is the corresponding energy con-

$$T^l_{n,m} = \sum_{v\in\mathcal{V}}\sum_{\mathcal{E}_{u,v}\in\mathcal{E}} T^l_{n,m,v} = \underbrace{\sum_{v\in\mathcal{V}}(1-I^l_{n,m,v})\cdot t^{local}_v}_{\text{local execution delay}} + \underbrace{\sum_{\mathcal{E}_{u,v}\in\mathcal{E}}[\ \overbrace{(t\_s\_low^{u,v}_{n,m} + t^{low}_v\cdot K^l_{n,m,v})\cdot I^l_{n,m,v}}^{\text{uplink transmission delay and SCceNB execution delay}} +}_{\text{offloading overhead}}$$

$$\underbrace{\overbrace{(t\_r\_low^{u,v}_{n,m} + t\_d_{u,v})\cdot I^l_{n,m,u}}^{\text{downlink transmission delay and decoding delay}} -(t\_s\_low^{u,v}_{n,m} + t\_r\_low^{u,v}_{n,m} + t\_d_{u,v})\cdot I^l_{n,m,v}\cdot I^l_{n,m,u}]}_{\text{offloading overhead}}, \tag{3.3.6}$$

$$E^l_{n,m} = \sum_{v\in\mathcal{V}}\sum_{\mathcal{E}_{u,v}\in\mathcal{E}} E^l_{n,m,v} = \underbrace{\sum_{v\in\mathcal{V}}(1-I^l_{n,m,v})\cdot E^{local}_v}_{\text{local energy consumption}} + \underbrace{\sum_{\mathcal{E}_{u,v}\in\mathcal{E}}\overbrace{[(E\_s\_low^{u,v}_{n,m} + E^{low}_v\cdot K^l_{n,m,v})\cdot I^l_{n,m,v}}^{\text{uplink transmission and idling energy consumption}}}_{\text{offloading overhead}}$$

$$+\underbrace{\overbrace{(E\_r\_low^{u,v}_{n,m} + E\_d^{u,v}_{n,m})\cdot I^l_{n,m,u}}^{\text{idling and local decoding energy consumption}} -(E\_s\_low^{u,v}_{n,m} + E\_r\_low^{u,v}_{n,m} + E\_d^{u,v}_{n,m})\cdot I^l_{n,m,v}\cdot I^l_{n,m,u}]}_{\text{offloading overhead}}, \tag{3.3.7}$$

sumption at the mobile terminal side. In this case, the total latency $T^l_{n,m}$ and energy consumption $E^l_{n,m}$ for UE m to execute the application $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ can be expressed by (3.3.6) and (3.3.7), respectively. Where $t\_d_{u,v}$ is the time needed by the UE m to decode the computation result (from component u to v) transmitted back by SCceNB n, and $E\_d^{u,v}_{n,m}$ denotes the corresponding energy consumption to decode the computation results transmitted back by SCceNB n.

Note that $I^l_{n,m,v}$ and $K^l_{n,v}$ are the binary indicator variables. Specifically, $I^l_{n,m,v} \in \mathcal{I}^l_{n,m}$ is the offloading decision variable, which is equal to one, if component v is processed remotely, or zero, if the component is executed locally. The UE's offloading requests are normalized such that $\sum_{n=1}^{N_l} I^l_{n,m,v} = 1$. $K^l_{n,v} \in \mathcal{K}^l_n$ is the computation results caching variable: which is equal to one, if UE m cannot find the computation results of component v cached in SCceNB n, and zero, if UE m can find the results in the corresponding SCceNB n through local caching.

Note also that a delay cost to transfer the data between UE m and SCceNB n occurs in (3.3.6) only if the two components u and v are processed at different locations, as illustrated in Table 3.1 (a).

Specifically, when $I^l_{n,m,u} = 0$ and $I^l_{n,m,v} = 1$ (i.e., the **SID** period of component v), UE m must send the results of component u to the SCceNB n. In this case, $T^l_{n,m,v}$ denotes the delay cost as shown in (3.3.6), where: (i) $t\_s\_low^{u,v}_{n,m}$, represents the **SID** delay cost for component v; and (ii)

$T^l_{n,m,v}$	$I^l_{n,m,v}=0$	$I^l_{n,m,v}=1$
$I^l_{n,m,u}=0$	t_v^{local}	$t\_s\_low^{u,v}_{n,m}+t_v^{low}\cdot K^l_{n,m,v}$
$I^l_{n,m,u}=1$	$t\_r\_low^{u,v}_{n,m}+t\_d_{u,v}+t_v^{local}$	$t_v^{low}\cdot K^l_{n,m,v}$

(a)

$E^l_{n,m,v}$	$I^l_{n,m,v}=0$	$I^l_{n,m,v}=1$
$I^l_{n,m,u}=0$	E_v^{local}	$E\_s\_low^{u,v}_{n,m}+E_v^{low}\cdot K^l_{n,m,v}$
$I^l_{n,m,u}=1$	$E\_r\_low^{u,v}_{n,m}+E\_d^{u,v}_{n,m}+E_v^{local}$	$E_v^{low}\cdot K^l_{n,m,v}$

(b)

Table 3.1 – All the possible values for $T^l_{n,m,v}$ and $E^l_{n,m,v}$ in (3.3.6) and (3.3.7), respectivelly.

t_v^{low}, represents the delay cost when SCceNB executes component v (if no results of v cached in SCceNB).

Whereas, when $I^l_{n,m,u}=1$ and $I^l_{n,m,v}=0$ (i.e., the **ROD** period of component v), SCceNB n must send the computation results from component u back to UE m. In this case, $T^l_{n,m,v}$ will be equal to the sum of the following three delay costs: (i) $t\_r\_low^{u,v}_{n,m}$, the **ROD** delay for UE m to receive the results back firstly; (ii) $E\_d^{u,v}_{n,m}$ the latency for UE m to decode the computation results from u; and (iii) t_v^{local} the delay for the UE m to process the component v locally.

On the other hand, if the two components u and v are processed at the same location, e.g, on the UE side, when $I^l_{n,m,u}=0$ and $I^l_{n,m,v}=0$, $T^l_{n,m,v}$ corresponds to the local execution delay t_v^{local}; or on the SCceNB side, when $I^l_{n,m,u}=1$ and $I^l_{n,m,v}=1$, $T^l_{n,m,v}$ corresponds to the remote execution delay t_v^{low} (if no results of v are cached in SCceNB).

Similarly, according to different values of $I^l_{n,m,u}$ and $I^l_{n,m,v}$, the total energy cost for component v in (3.3.7) is shown in Table 3.1 (b).

Note that the objective of data caching is to save CPU cycles for repeated computation, and thus the best data caching policy is to cache more popular computation results with higher ω (cpu cycles per byte). The optimal data caching policies $K_n^{l,*}$ $(n=1,2,...,N_l)$ for the low-end (distributed caching) scenario is given as follows:

$$\begin{aligned} K_n^{l,*} &= \underset{K^l_{n,v}\in\mathcal{K}^l_n}{\operatorname{argmin}} K^l_{n,v}\cdot|\mathcal{E}_{v,w}|\cdot\omega(v), \\ \textit{s.t.} &\sum_{v,w\in\mathcal{V}} K^l_{n,v}\cdot|\mathcal{E}_{v,w}|\le Q_l, \\ & K^l_{n,v}\in\{0,1\}, \\ & n=1,2,...,N_l. \end{aligned} \tag{3.3.8}$$

In order to minimize the average delay for UEs, we formulate the problem as an optimal offloading strategy under given caching lists of the SCceNBs, which is a simple 0-1 programming problem that aims at selecting the optimal number of components to be offloaded at SCceNB n for UE m. This can be formulated as follows:

When UE m makes an offloading decision to SCceNB n for its current application $\mathcal{G}$, we define

$$T^h_{n,m} = \sum_{v\in\mathcal{V}}\sum_{\mathcal{E}_{u,v}\in\mathcal{E}} T^h_{n,m,v} = \underbrace{\sum_{v\in\mathcal{V}}(1-I^h_{n,m,v})\cdot t^{local}_v}_{\text{local execution delay}} + \underbrace{\sum_{\mathcal{E}_{u,v}\in\mathcal{E}} [\overbrace{(t\_s\_high^{u,v}_{n,m} + t^{high}_v \cdot K^h_{m,v})\cdot I^h_{n,m,v}}^{\text{uplink transmission delay and server execution delay}} +}_{\text{offloading overhead}}$$

$$\underbrace{\overbrace{(t\_r\_high^{u,v}_{n,m} + t\_d_{u,v})\cdot I^h_{n,m,u}}^{\text{downlink transmission delay and local decoding delay}} -(t\_s\_high^{u,v}_{n,m} + t\_r\_high^{u,v}_{n,m} + t\_d_{u,v})\cdot I^h_{n,m,v}\cdot I^h_{n,m,u}]}_{\text{offloading overhead}}, \tag{3.3.22}$$

$$E^h_{n,m} = \sum_{v\in\mathcal{V}}\sum_{\mathcal{E}_{u,v}\in\mathcal{E}} E^h_{n,m,v} = \underbrace{\sum_{v\in\mathcal{V}}(1-I^h_{n,m,v})\cdot E^{local}_v}_{\text{local energy consumption}} + \underbrace{\sum_{\mathcal{E}_{u,v}\in\mathcal{E}} [\overbrace{(E\_s\_high^{u,v}_{n,m} + E^{high}_v \cdot K^h_{m,v})\cdot I^h_{n,m,v}}^{\text{uplink transmission and idling energy consumption}}}_{\text{offloading overhead}}$$

$$\underbrace{\overbrace{+(E\_r\_high^{u,v}_{n,m} + E\_d^{u,v}_{n,m})\cdot I^h_{n,m,u}}^{\text{idling and decoding energy consumption}} -(E\_s\_high^{u,v}_{n,m} + E\_r\_high^{u,v}_{n,m} + E\_d^{u,v}_{n,m})\cdot I^h_{n,m,v}\cdot I^h_{n,m,u}]}_{\text{offloading overhead}}, \tag{3.3.23}$$

---

the optimal offloading decision $I\_low^*_{n,m} = \{I^l_{n,m,v}, v\in\mathcal{V}\}$ ($I\_low^*_{n,m} \in \mathcal{I}\_low^*_m$) under caching list $K^l_{n,v}$ as **Optimization Problem 1:**

$$I\_low^*_{n,m}(K^{l,*}_n, \mathcal{G}) = \underset{I^l_{n,m,v}}{\operatorname{argmin}}\, T^l_{n,m} \tag{3.3.9}$$

$$\textit{s.t.} \quad T^l_{n,m}(I\_low^*_{n,m}) \le \sum_{v\in\mathcal{V}} t^{local}_v, \tag{3.3.10}$$

$$E^l_{n,m}(I\_low^*_{n,m}) \le \sum_{v\in\mathcal{V}} E^{local}_v, \tag{3.3.11}$$

$$\sum_{n=1}^{N_l} I^l_{n,m,v} = 1, \;\; m = 1,2,...,M, \;\; v\in\mathcal{V}, \tag{3.3.12}$$

where $\mathcal{I}\_low^*_m$ represents the set of the optimal offloading decisions between the m^{th} UE and all the available nearby SCceNBs in the network (i.e., the ones that meet the constraints). Note that constraint (3.2.10) indicates that the time it takes to execute the application through offloading action $I\_low^*_{n,m}$ must less than the local execution delay, while constraint (3.2.11) ensures that the total energy consumption for a feasible offloading action $I\_low^*_{n,m}$ is less than the total energy consumption of local execution.

$T^h_{n,m,v}$	$I^h_{n,m,v}=0$	$I^h_{n,m,v}=1$
$I^h_{n,m,u}=0$	t_v^{local}	$t\_s\_high^{u,v}_{n,m}+t_v^{high}\cdot K^h_{m,v}$
$I^h_{n,m,u}=1$	$t\_r\_high^{u,v}_{n,m}+t\_d_{u,v}+t_v^{local}$	$t_v^{high}\cdot K^h_{m,v}$

(a)

$E^h_{n,m,v}$	$I^h_{n,m,v}=0$	$I^h_{n,m,v}=1$
$I^h_{n,m,u}=0$	E_v^{local}	$E\_s\_high^{u,v}_{n,m}+E_v^{high}\cdot K^h_{m,v}$
$I^h_{n,m,u}=1$	$E\_r\_high^{u,v}_{n,m}+E\_d^{u,v}_{n,m}+E_v^{local}$	$E_v^{high}\cdot K^h_{m,v}$

(b)

Table 3.2 – All the possible values for $T^h_{n,m,v}$ and $E^h_{n,m,v}$ in (3.3.22) and (3.3.23), respectivelly.

3.3.2.2 Multi-user scenario

Then, we focus on the reduction of average latency for MCA collaborative execution within multiple users. Note that in the multi-user multi-small cell scenario, different SCceNBs have different data caching contents. Therefore, the caching policies $K_n^{l,*}$ are different if one user attach to different SCceNBs, which can change their offloading decisions. On the other hand, users' offloading request can affect the local caching content of its serving SCceNB, and thus affect the offloading decision of other users who attach to the same SCceNB. Therefore, when UEs make offloading decisions in a collaborative manner, and we can minimize the average delay for the UEs as **Optimization Problem 2:**

$$\min \ \frac{1}{M}\cdot\sum_{m=1}^{M} T^l_{n,m}\left(I\_low^*_{n,m}, K_n^{l,*}\right) \tag{3.3.13}$$

$$\textit{s.t.} \quad m\in\mathcal{M}, \tag{3.3.14}$$

$$n\in\mathcal{N}_l, \tag{3.3.15}$$

$$I\_low^*_{n,m}\in\mathcal{I}\_low^*_m, \tag{3.3.16}$$

$$T^l_{n,m}(I\_low^*_{n,m})\le\sum_{v\in\mathcal{V}} t_v^{local}, \tag{3.3.17}$$

$$E^l_{n,m}(I\_low^*_{n,m})\le\sum_{v\in\mathcal{V}} E_v^{local}, \tag{3.3.18}$$

$$\sum_{n=1}^{N_l} I^l_{n,m,v}=1, \ \ m=1,2,...,M, \ \ v\in\mathcal{V}, \tag{3.3.19}$$

where (3.2.17) and (3.2.18) are the delay constraint and energy constraint for the optimal offloading decision, respectively. (3.2.19) denotes each mobile user offloads its component to a single SCceNB.

3.3.3 Problem formulation for High-end MEC deployment

3.3.3.1 Single-user Scenario

Then, we consider the single-user high-end deployment case, where UE m ($m \in \mathcal{M}$) offloads its components to an aggregation point (i.e., the centralized high-end server) through its nearby SCeNB n ($n \in \mathcal{N}_h$). Similar with the low-end case, the total latency $T^h_{n,m}$ and energy consumption $E^h_{n,m}$ for UE m to execute the application can be expressed by (3.3.22) and (3.3.23), respectively. Here $T^h_{n,m,v}$ (given by Table 3.2 (a)) is the execution delay of component v in high-end case, and $E^h_{n,m,v}$ (illustrated in Table 3.2 (b)) is the corresponding energy consumption at the mobile terminal side. $I^h_{n,m,v}$ and K^h_v are the indicator variables. Specifically, $I^h_{n,m,v} \in \mathcal{I}^h_{n,m}$ is the offloading decision variable, which is equal to one, if component v is processed remotely, or zero, if it is executed locally. Whereas $K^h_v \in \mathcal{K}^h$ is the computation results caching variable: which is equal to one, if the UE m can not find the computation results of component v shared in the high-end server (through local caching), and zero, if UE m can find the results in the server. The optimal data caching policies $K^{h,*}$ for the high-end (centralized caching) scenario is given as follows:

$$\begin{aligned} K^{h,*} &= \underset{K^h_v \in \mathcal{K}^h}{\operatorname{argmin}} K^h_v \cdot |\mathcal{E}_{v,w}| \cdot \omega(v), \\ \textit{s.t.} &\sum_{v,w \in \mathcal{V}} K^h_v \cdot |\mathcal{E}_{v,w}| \leq Q_h, \\ & K^h_v \in \{0,1\}. \end{aligned} \tag{3.3.22}$$

Different from the low-end case, each user in the high-end deployment can be served by one or multiple SCeNBs, depending on the employed transmission scheme (such as CoMP [67]). The optimal offloading decision problem for high-end deployment is thus given by **Optimization Problem 3:**

$$I\_high^*_{n,m}(K^{h,*}, \mathcal{G}) = \underset{I^h_{n,m,v}}{\operatorname{argmin}} T^h_{n,m} \tag{3.3.23}$$

$$\textit{s.t.} \quad T^h_{n,m}(I\_high^*_{n,m}) \leq \sum_{v \in \mathcal{V}} t^{local}_v, \tag{3.3.24}$$

$$E^h_{n,m}(I\_high^*_{n,m}) \leq \sum_{v \in \mathcal{V}} E^{local}_v, \tag{3.3.25}$$

$$\sum_{n=1}^{N_h} I^h_{n,m,v} \geq 1, \;\; m = 1, 2, ..., M, \;\; v \in \mathcal{V}, \tag{3.3.26}$$

where $I\_high^*_{n,m}$ denotes the optimal offloading decision between UE m and SCeNB n that can minimize the total delay for processing the application $\mathcal{G}$. (3.2.24) and (3.2.25) are the delay and energy consumption constraints for the optimal offloading decision, respectively.

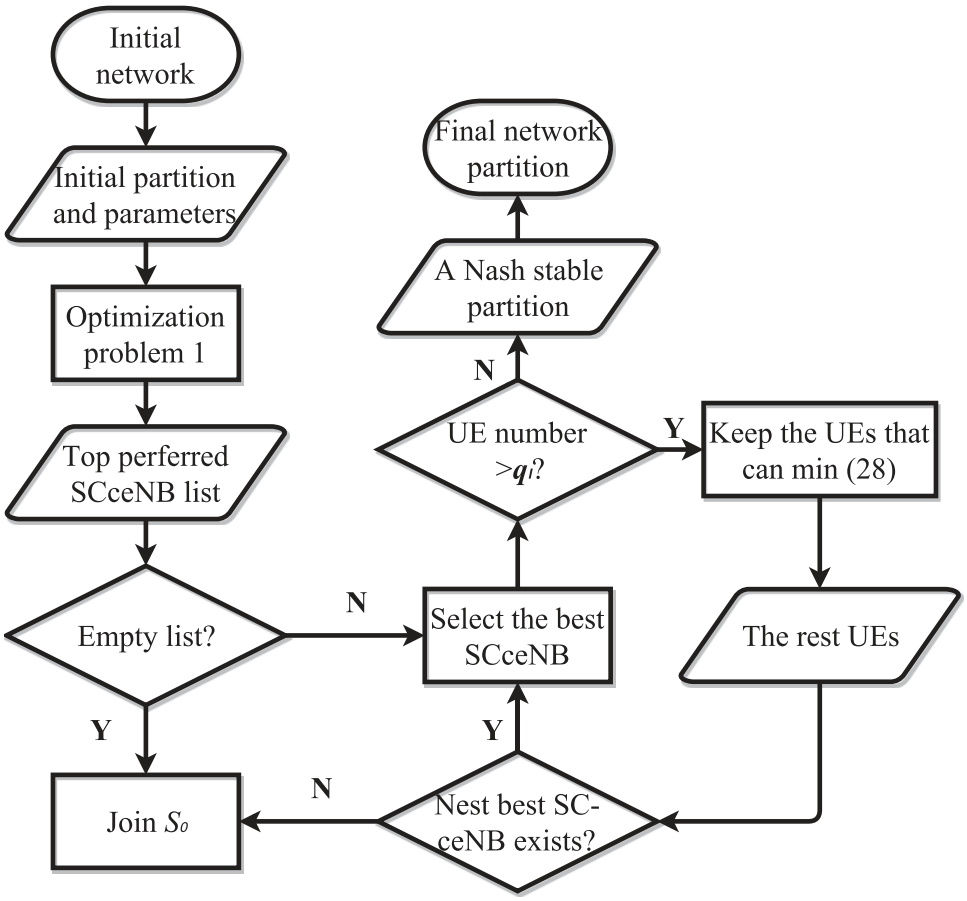


Figure 3.2 – The flowchart of *Algorithm 1*

3.3.3.2 Multi-user scenario

Note that in the high-end deployment, all the mobile users offload the components to the high-end server, which means that the users make offloading decisions according to the same caching factor $K^h_{m,v}$ in a non-cooperative manner. Each user can thus select its nearest SCeNB (i.e., the SceNB with highest $r^{dl}_{n,m}$ and $r^{ul}_{n,m}$) as its serving small cell. We can reduce the average delay through minimizing each single user's execution latency as formulated in **Optimization Problem 3**.

3.4 Proposal: OOCS scheme

In this section, we present our proposed optimal offloading with caching-enhancement scheme (OOCS). We give the details how the optimal offloading decisions for all users in the network are made by computing first a list of the optimal number of components to be offloaded to nearby small cells for each user (using **Optimization Problems 1** and **3**, respectively). Then we define

the optimal partitions of users (or coalitions) for low-end deployment. In order to identify which SCceNB can serve the attached users and execute the offloading decisions, we explore the concept of coalition formation game [68, 69] to solve **Optimization Problem 2**.

3.4.1 Game Formulation

Indeed, in the low-end deployment case, the attachment of mobile users to a particular SCceNB can be seen as a coalition formation game in partition form with transferable utility. Specifically, let M UEs be players, and π be the set of existing users in the network. We assume that UEs in each coalition connect to a single SCceNB and form a coalition. Let S_n denote the set of UEs that are served by SCceNB n ($n \in \mathcal{N}_l$), and S_0 denote the set of UEs that execute the application locally, i.e. without offloading.

The payoff of UE m in the coalition S_n (in terms of execution latency) can be expressed as follows

$$V_m(S_n, \pi) = \begin{cases} T^l_{n,m}, & n \neq 0, \\ \sum\limits_{v \in \mathcal{V}} t^{local}_v, & n = 0. \end{cases} \tag{3.4.27}$$

We define $V_{S_n}(\pi)$, as the sum of the payoff of all players within the coalition, i.e. total execution delay of all UEs in the coalition S_n, and is given by:

$$V_{S_n}(\pi) = \sum_{m \subseteq S_n} V_m(S_n, \pi). \tag{3.4.28}$$

Based on this, the optimal network partition (coalition formation) for multi-user low-end deployment is given by *Algorithm 1*. The corresponding flowchart is shown in Fig.5.3. Note that the UEs merge through SCceNB keeping the top q_l UEs that can minimize (3.4.28). The rejected users will re-send their location and component information to the next SCceNB of their top preferred lists. Each SCceNB updates its list of serving users, and then repeats the previous process of coalition formation until all UEs are allocated to their nearby SCceNB. Note that user m has an incentive to move from its current coalition S_a to another coalition S_b in π^* if the following split rule in (3.4.29) is satisfied.

$$v(S_a \backslash \{m\}) + v(S_b \bigcup \{m\}) < v(S_a) + v(S_b). \tag{3.4.29}$$

Algorithm 1: Optimal Network Partition for Multi-user Low-end Deployment

Initial network:
initial network partition for the UEs: $\{\{\emptyset\}, \{1\}, \{2\}, ..., \{M\}\}$.

Step 1: Component Offloading Decision
UEs work in a Non-cooperative manner
Input of Step 1: Parameters M, N_l, $K^l_{m,n,v}$, Q_l, t_v^{local}, $\mathcal{G}$, $t\_s\_low_{n,m}$, t_v^{low}, $t\_r\_low^{u,v}_{n,m}$, $t\_d_{u,v}$.
1) Each UE builds a top preferred SCceNB list $I\_low^*_{n,m}$ according to **Optimization Problem 1**.
2) Each UE selects its best preferred SCceNB as its serving SCceNB and submit its offloading requests.
3) For the UEs whose list is empty, they join S_0.
Output of Step 1: $I\_low^*_{n,m}$.

Step 2: Coalition Formation
UEs work in a cooperative manner
Input of Step 2: Parameters $I\_low^*_{n,m}$, q_l.
4) Each SCceNB receives the requests. Due to the limited computation capacity of SCceNBs, each SCceNB keeps the top q_l UEs that can minimize (3.4.28), and reject the rest.
5) The rejected UEs will re-apply to their next best SCceNB of their list $I\_low^*_{n,m}$, and each SCceNB updates its serving UEs list.
6) Repeat 5), until convergence to a final Nash-stable partition π^*. For the UEs who cannot be allocated to a SCceNB, they execute the application locally and join S_0.
Output of Step 2: π^*

It is worth noting that UEs who can not be allocated to a SCceNB will execute their application locally and join the coalition S_0. The final network partitions will be thus given as $\pi^* = \{S_0, S_1, ..., S_{N_l}\}$.

Proposition 3.4.1 *Starting by any initial partition π_{ini} from step 4) in Algorithm 1, the coalition formation for transfers is guaranteed to converge to a final stable partition π^*.*

Proof. For a partition Π_{ini} from step 4) in *Algorithm 1*, the coalition formation process can be seen as a sequence of transfer operations that transform the network's partition,

$$\Pi_l = \Pi_{ini} \rightarrow \Pi_1 \rightarrow \Pi_2 \rightarrow ..., \tag{3.4.30}$$

where $\Pi_l = \{S_0, S_1..., S_{N_l}\}$ is a partition which composed of at most $N_l + 1$ coalitions (N_l coalitions forms at N_l given SCceNBs and one local execution coalition S_0) that is formed after l transfers. User m has an incentive to move from its current coalition S_a to another coalition S_b in π^* if

$$v(S_a \backslash \{m\}) + v(S_b \bigcup \{m\}) < v(S_a) + v(S_b). \tag{3.4.31}$$

As a transfer between two SCceNBs a and b in a partition in a partition Π_l, does not affect the total utility generated by the other coalitions in $\Pi_l \setminus \{S_a, S_b\}$ (since UEs use orthogonal channels, we do not consider intra-cell interference in this thesis), every transfer $\Pi_l \rightarrow \Pi_k$ forms an order such that

$$\Pi_l \rightarrow \Pi_k \Leftrightarrow \sum_{S_m \in \Pi_k} v(S_m) < \sum_{S_m \in \Pi_l} v(S_p) \tag{3.4.32}$$

which is transitive and irreflexive. Therefore, for any two partitions, we have $\Pi_l \neq \Pi_k$, $l \neq k$. As the number of partitions of a set is finite and equal to the Bell number, the sequence in (3.4.30) is guaranteed to converge to a final partition π^*. ■

3.4.2 Algorithm Description

To sum up, our algorithm consists of three main phases: (i) Initial phase, selection of possible offloaded components, along with the best preferred SCeNB for each single UE in high-end deployment case, or a list of top preferred SCceNBs for each single UE in the low-end deployment case. In the latter case, the optimal network partition shown in *Algorithm 1* is executed, (ii) Cache placement phase, where we determine the cache results at each edge server. Specifically, in the high-end deployment case, the central high-end server make caching decision through collecting the offloading requests of all the UEs around the network. Whereas in the low-end deployment case, each SCceNB caches the results according to its nearby UEs' offloading requests in a distributed manner, and (iii) Delivery phase, which allows the edge servers to deliver the requested results to the UEs over wireless channels. In what follows, we describe these three phases.

First of all, each UE observes the current network states to identify the accessible nearby small cells over a control channel, and computes the optimal number of components to be offloaded at each identified nearby small cell by resolving **Optimization Problem 1** (low-end) or **Optimization Problem 3** (high-end). A top preferred SCeNB (high-end case) or a list of top preferred SCceNBs (low-end case) will be thus created for every user according to the server's previous caching content $K^l_{n,m,v}$ or $K^h_{m,v}$. The list is sorted in ascending order according to the total latency.

In the high-end case, each user attaches to their best preferred SCeNB and submit its location and offloading requests to the high-end server. The server makes current cache placement decision $K^h_{m,v}$ according to its previous caching content and the current offloading requests.

Whereas in the low-end case, each user first selects its best preferred SCceNB (the one from its list with minimum delay) as its serving SCceNB and submit its location and offloading requests to the SCceNB. Then, the optimal network partition is derived using *Algorithm 1*. After each user attaching to the optimal small cell, the servers deliver the requested results to the UEs over wireless channels. At last, each SCceNB makes the optimal cache placement decision $K^{l,*}_n$ (as shown in (3.3.8)) according to its serving users' offloading requests and previous caching content.

3.4.3 Complexity analysis

The optimal caching placement problems formulated in (3.3.8) and (3.3.22) are the simple 0-1 Knapsack problems [70]. The problem is NP-hard [71], but the suboptimal solutions can be obtained in pseudo-polynomial time by dynamic programming approach [72]. Thus, the complex-

Table 3.3 – Network Parameters

Parameter	Value	Parameter	Value	Parameter	Value
λ_s^l	10^{-3}	λ_s^h	10^{-3}	λ_u	5×10^{-3}
g_{ul}	10^{-3}	g_{dl}	10^{-3}	N_0	5×10^{-5}
t_e	0.02 s	t_d	0.2 s	p_c	0.9 W
p_u	0.01 W	p_s	0.1 W	p_i	0.3 W
f_m	10^9 cyclse/s	f_s	10^{10} cyclse/s	f_c	10^{11} cyclse/s
Q_h	4 Gbits	Q_l	2 Gbits	β	-2
q_h	50	q_l	6	r	100m

ities of our suboptimal caching placement algorithms are $\mathcal{O}(|\mathcal{E}_{v,w}|Q_l)$ for the low-end case and $\mathcal{O}(|\mathcal{E}_{v,w}|Q_h)$ for the high-end case. Note that both problems are solved at the server side after the execution of MCAs. Thus, the overheads (delay) for solving both optimization problems have no impact on the execution latency in (3.3.6) and (3.3.22). In addition, Optimization Problem 1 and Optimization Problem 3 are 0-1 programming problems that can be solved through the branch and bound algorithm. Their corresponding complexities are both $\mathcal{O}(2^{(|\mathcal{V}|)})$ in the worst case, but can be reduced through pruning schemes, such as alpha-beta pruning [73]. Optimization Problem 2, on the other hand, can be viewed as a college admission game [74]. The complexity for the solution is $\mathcal{O}(N_l)$ in the worst case.

3.5 Performance evaluation

In this section, we evaluate the efficiency of our proposal for both low-end and high-end deployments under single-user and multi-user scenarios. We assume that the geographical distributions of both the small cells and UEs follow an independent Homogeneous Poisson Point Process (HPPP) [75], which is suitable for describing the distribution of small devices. The geographical distributions of both small cells and UEs are thus given by:

$$P(n,r) = \frac{[\lambda L(r)]^n}{n!} e^{-\lambda L(r)} \tag{3.5.33}$$

where $L(r) = r^2$ denote the area of the network of radius r. λ is the mean density (intensity) of points. We define λ_u, λ_s^l and λ_s^h as the density of UEs, SCceNBs and SCeNBs, respectively.

For the computing components, we assume that each application consists five random components, both data dependencies $\mathcal{E}_{u,v}$ and required number of CPU cycles per byte (cpb) for components ω follow the uniform distribution with $\mathcal{E}_{u,v} \in [100, 500]$ KB and $\omega \in [4000, 12000]$ cpb as in [62]. All random variables are independent for different components. The size of computation results library $|\mathcal{E}_{u,v}| = 1000000$.

We compare the benefits of our optimal offloading with caching-enhancement scheme (OOCS) with respect to six benchmark policies, namely:

- *No Offloading Scheme (NOS):*
  Local execution, which means that applications are executed on smartphones, by letting offloading decision variables $\mathcal{I}^l_{n,m}$ and $\mathcal{I}^h_{n,m}$ equal to 0 in (3.3.6) and (3.3.22), respectively.

- *Optimal Offloading without Caching Scheme (OOS):*
  Traditional offloading strategies [21–23] without caching enhancement, i.e., let caching decision variables $\mathcal{K}^l_{n,m}$ and $\mathcal{K}^h$ equal to 0 in (3.3.6) and (3.3.22), respectively.

- *Total Offloading without Caching Scheme (TOS):*
  Coarse offloading strategies [31,47] without caching enhancement, which means that offload all the components to the server side, i.e., let offloading decision variables $\mathcal{I}^l_{n,m}$ and $\mathcal{I}^h_{n,m}$ equal to 1 in (3.3.6) and (3.3.22), respectively.

- *Total Offloading and Caching Scheme (TOCS):*
  Coarse offloading with caching enhancement.

- *D2D Offloading without Caching Scheme (DOS):*
  D2D offloading strategies [29,30] without caching enhancement, which means resource-poor devices can utilize other users' vacant computing resources.

- *D2D Offloading and Caching Scheme (DOCS):*
  D2D offloading strategies with D2D caching [12, 76, 77] enhancement.

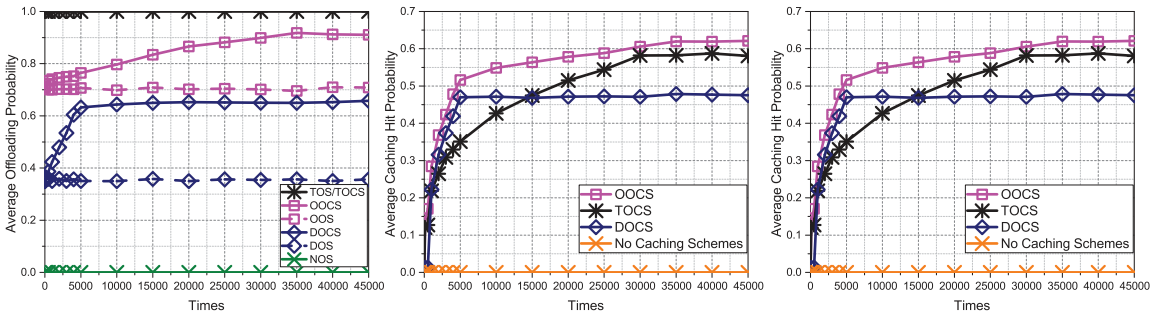
Note that in the D2D offloading scenario, we assume that the offloadee mobile devices [30] (servers) are equipped with more powerful processors than the offloader mobile devices (clients), the cpu rate of offloadee devices $f_{ms} = 1.5 \times 10^9$ cpu cycles per second. Each offloadee device allocates 0.5 Gbits of its memory to D2D caching. The parameters used in our simulations are reported in Table 6.1.

3.5.1 Performance evaluation of OOCS in single-user scenario

We first illustrate the performance of our proposed OOCS scheme for the single-user scenario. This is done by minimizing the execution delay for a single user through **Optimization Problem 1** and **Optimization Problem 3**.

To do so, we generate 45000 independent applications and run the applications continuously. Let the required number of cpu cycles per byte for the components $\omega = 5900$ cpb, which corresponds to the workload of processing the English main page of Wikipedia [63]. Assume that offloading requests are modeled as the Zipf distribution and the popularity factor $\delta = 1$ (shown in (3.2.3)). The simulation results are averaged over 2000 Monte-Carlo topology realizations.

Fig. 3.3(a), 3.3(b), 3.3(c) show the curves of average offloading probability, average caching

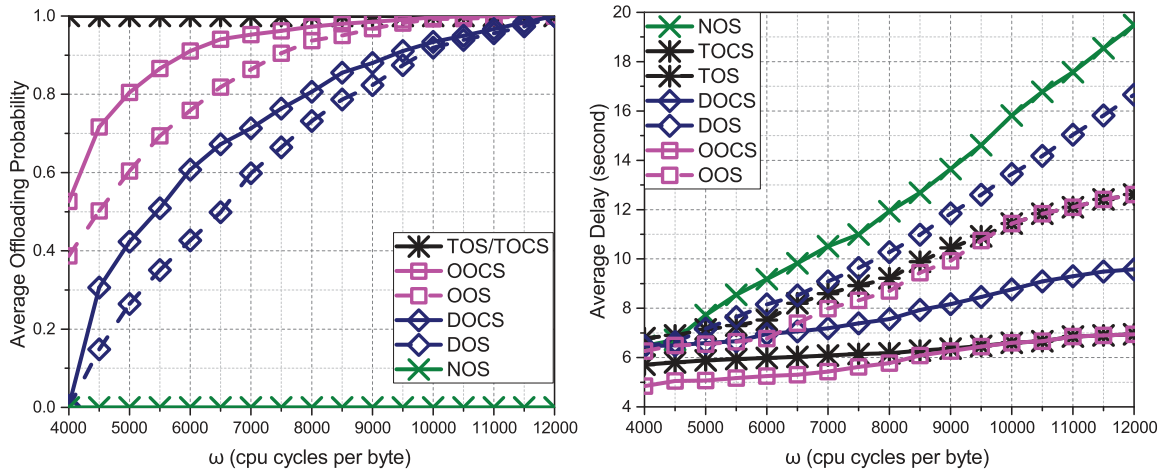


(a) Average offloading probability vs. execution times (b) Average caching hit probability vs. execution times (c) Average application execution delay vs. execution times

Figure 3.3 – High-end performance vs. application execution times.

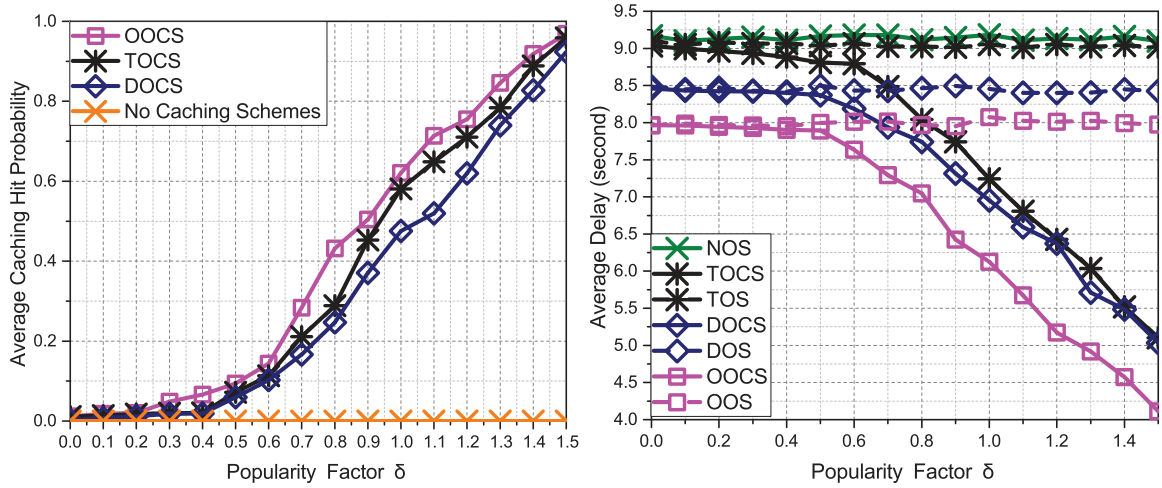
hit probability and average application execution delay versus application execution times, respectively. Several observations can be made. First of all, for all the non-caching schemes (TOS, NOS, DOS, OOS), execution times have no influence on the average offloading probability, average caching hit probability and the average execution delay. The reason is that without caching enhancements, the users make offloading decision independently at the beginning of each execution time. Secondly, for the schemes allowing caching enhancement (OOCS, DOCS), the average offloading probability and average caching hit probability increase considerable with the execution time. We observe a peak after the servers' storage capacities are achieved. Specifically, the storage capacity of high-end server is achieved after about 35000 times of execution and storage capacity of offloadee is achieved after about 5000 times of execution. This suggests that as the execution time grows, more popular computation results are cached in the server. Thus, the user prones to offload the components to the server to reduce the execution delay. Last but not least, it can be observed from Fig. 3.3(c) that after the high-end server's storage capacity is achieved, our high-end OOCS algorithm can reduces 28.04%, 33.28%, 32.31%, 15.42%, 29.19% and 11.89% execution delay compared with OOS, NOS, TOS, TOCS, DOS and DOCS, respectively.

Figs. 3.4 and 3.5 plot the high-end performance versus ω (required cpb for the components) and the popularity factor δ, respectively. It can be seen from the figures that ω has tremendous influence on the average offloading probability (as shown in Fig. 3.4(a)), whereas δ plays a key role in the average caching hit probability (as shown in Fig. 3.5(a)). It is evident that the local execution overhead increases as ω grows. Thus, the mobile user prefers to offload the components rather than local execution. Moreover, higher ω and δ can bring larger reduction of execution delay (as shown in Figs. 3.4(b) and 3.5(b)), and our OOCS algorithm still performs better than the other schemes in the delay reduction.



(a) Average offloading probability vs. ω (b) Average execution delay vs. ω

Figure 3.4 – High-end performance vs. the required number of cpu cycles per byte for the components.

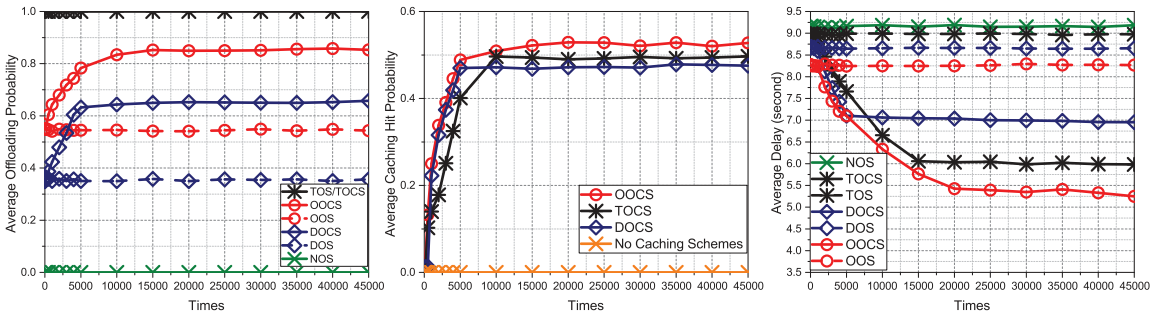


(a) Average caching hit probability vs. δ (b) Average execution delay vs. δ

Figure 3.5 – High-end performance vs. popularity factor.

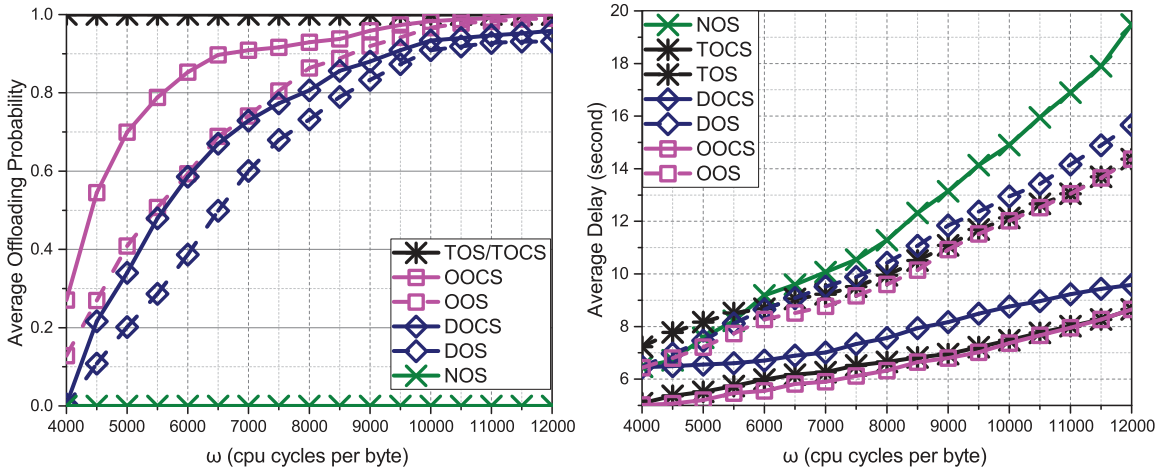
3.5.1.1 Low-end deployment scenario

Similar results are given in Figs. 3.6, 3.7 and 3.8 for low-end deployment scenario. Specifically, Fig. 3.6 reports the low-end performance versus the application execution times. Note that the storage capacity of SCceNB is achieved after about 20000 times of execution. Figs. 3.7 and 3.8 plot the low-end performance versus ω and δ, respectively. From the figures, we can see clearly that



(a) Average offloading probability vs. execution times (b) Average caching hit probability vs. execution times (c) Average application execution delay vs. execution times

Figure 3.6 – Low-end performance vs. application execution times.

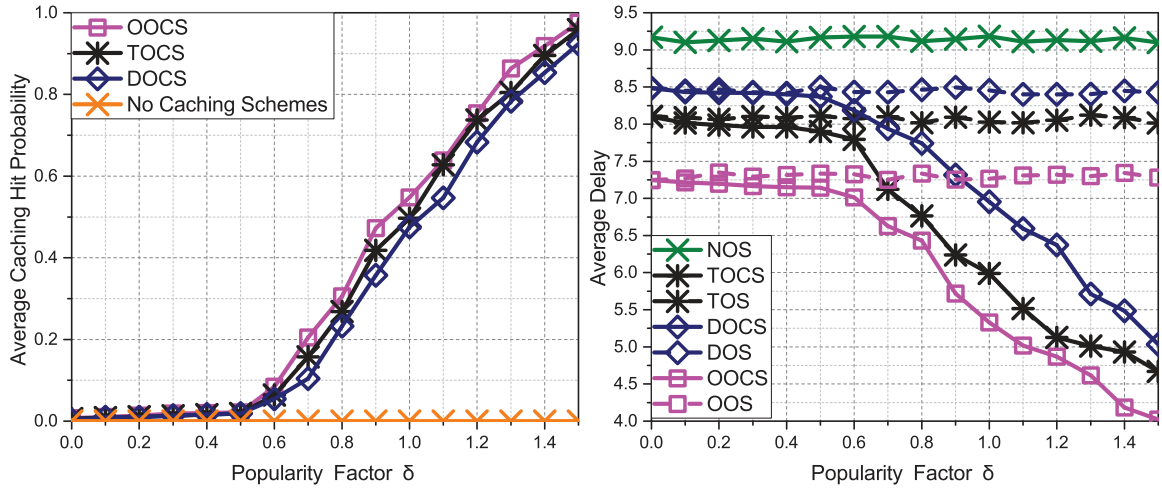


(a) Average offloading probability vs. ω

(b) Average execution delay vs. ω

Figure 3.7 – Low-end performance vs. required number of cpu cycles per byte for the components.

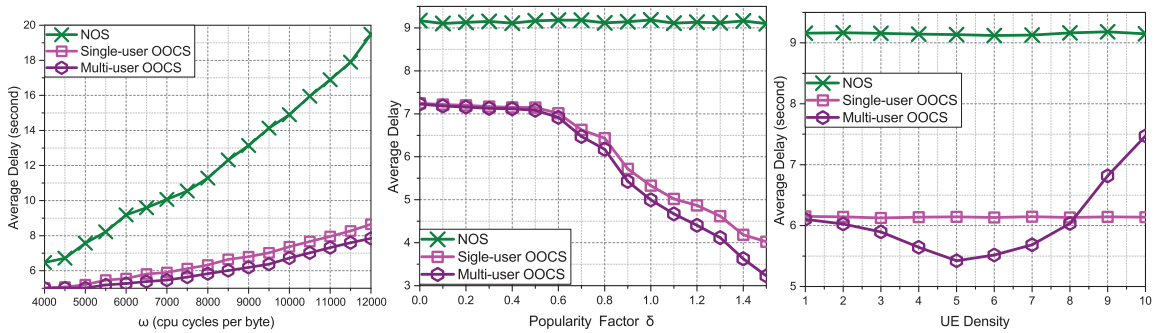
ω and δ still play key roles in the average offloading probability and average caching hit probability, respectively. Note that the average offloading probability of OOCS scheme increases considerable from 28% to 97%, when ω grows from 4000 to 9000, as shown in Fig. 3.7(a). When $\omega > 9000$, the average offloading probability of our OOCS will be more than 99%. This suggests that the performance of our OOCS is similar with the performance of total offloading schemes (TOS and TOCS) when $\omega > 9000$. The situation is illustrated by Fig. 3.7(b) (b). On the other hand, when δ grows from 0 to 0.5, there was hardly any change in the value of average caching hit probability. This suggests that when $\delta < 0.5$, the schemes with caching enhancements (TOCS, OOCS, DOCS) performs similarly with respect to their non-caching schemes (TOS, OOS, DOS). The trend is given by Fig. 3.8(b). In addition, it can be observed from Fig. 3.6(c) that after SCceNB's storage capacity



(a) Average caching hit probability vs. δ

(b) Average execution delay vs. δ

Figure 3.8 – Low-end performance vs. popularity factor.



(a) Average application execution delay vs. ω

(b) Average application execution delay vs. δ

(c) Average application execution delay vs. UE density λ_u (10^{-3})

Figure 3.9 – Low-end (Femto-cloud) performance of multi-user scenario.

achieved, our low-end OOCS algorithm can reduce 36.49%, 42.83%, 41.51%, 12.25%, 39.33% and 24.50% execution delay compared to OOS, NOS, TOS, TOCS, DOS and DOCS, respectively.

3.5.1.2 High-end/Low-end performance comparison

As stated earlier, the high-end and low-end scenarios perform quasi similarly. However, there are still several differences between the two scenarios in the following aspects: (i) The high-end deployment performs better in the average offloading probability and average caching hit probability, whereas the low-end deployment performs better in the average delay reduction. The reason is that the high-end server is equipped with more powerful processors and larger storage sizes (i.e.

larger caching list) than the SCceNB, and thus mobile users have more incentive to offload the components to the high-end server. On the other hand, the high-end deployment performs worse in delay reduction due to the large end-to-end delay from SCeNBs to the high-end server. (ii) The performance thresholds between schemes are different in the two deployments. For example, as shown in Figs. 3.4(a) and 3.7(a), the CPB thresholds between OOCS and TOCS are 7500 and 9000 for high-end and low-end deployments, respectively. In addition, the popularity factors thresholds between OOCS and OOS are 0.4 and 0.5 for high-end and low-end deployments, respectively (as shown in Figs. 3.5(a) and 3.8(a)).

3.5.2 Performance evaluation of OOCS in multi-user scenario

We then illustrate the performance of our OOCS scheme for the multi-user scenario. This is done by minimizing the average execution delay for multiple users through **Optimization Problem 3**. Fig. 3.9 show the multi-user delay performance versus ω, δ and UE density λ_u, respectively. Note that our multi-user OOCS performs better in delay reduction when ω grows as shown in Fig. 3.9(a). The reason is that the offloading probability increases with ω since more users can reduce their delay through joining our coalition game performed in OOCS. Similarly, as δ grows, more users can find their required results cached in their serving SCceNB, and thus reducing their execution delay, as shown in Fig. 3.9(b). Finally, Fig. 3.9(c) shows that our multi-user OOCS performs better as UE density grows from 10^{-3} to 8×10^{-3}. A peak is observed when $\lambda_u = 5 \times 10^{-3}$, which corresponds to 11.71% and 40.61% delay reduction, compared to single-user OOCS and NOS, respectively. When $\lambda_u > 8 \times 10^{-3}$, single-user OOCS performs better. The reason is that, when λ_u grows larger ($> 5 \times 10^{-3}$), the SCceNBs cannot afford such many UEs (we assume that each SCceNB can handle 6 UEs in our simulations), and thus a large number of UEs will be rejected and run the application locally. As a result, the average delay increases and will be close to the local execution delay.

3.6 Conclusion

In this chapter, we studied the computation offloading problem with caching enhancement for mobile edge cloud networks. We proposed an optimal offloading with caching-enhancement scheme (OOCS) to minimize the execution delay for mobile users in two kinds of edge cloud scenarios. First, we proposed a concept of cooperative call graph to formulate the offloading and caching relationships within multiple components. Then, in order to minimize the overall delay for multi-user femto-cloud networks, we formulated the problem of users' allocation as a coalition formation game. We proved the existence of convergence. Compared to six alternative solutions in literature, our proposed approach achieves the best performance. Specifically, our high-end OOCS algorithm can reduces 28.04%, 33.28%, 32.31%, 15.42%, 29.19% and 11.89% execution delay compared

with OOS, NOS, TOS, TOCS, DOS and DOCS, respectively. Whereas our low-end OOCS algorithm can reduce 36.49%, 42.83%, 41.51%, 12.25%, 39.33% and 24.50% execution delay compared to OOS, NOS, TOS, TOCS, DOS and DOCS, respectively. Moreover, when consider the multiuser scenario, our multiuser OOCS can reduce 11.71% delay comparing with the single-user OOCS.