Mobility-Aware Real-Time Task Allocation in the 5G-Enabled Embedded–Edge Compute Continuum

Xiaopeng Teng
Dept. Computer and Information Science
Linköping University
Sweden
xiaopeng.teng@liu.se

Soheil Samii
Dept. Computer and Information Science
Linköping University
Sweden
soheil.samii@liu.se

Johan Wibeck
Radio Access Networks
Ericsson AB
Sweden
johan.wibeck@ericsson.com

***Abstract*—Cyber-physical systems are getting increasingly connected to cellular and wireless networks. As a prominent example, the automotive industry has witnessed a rapid increase in embedded software and electronics, and lately also increased connectivity to cellular communication networks. This trend enables the deployment of applications on network edge computers, offering opportunities for reduced embedded hardware footprint and increased expandability of new applications. Put differently, the traditional automotive platform comprising embedded networked processors expands to the cellular network and edge computing services in its environment. For real-time applications, which are very common in the automotive domain, the aforementioned trend gives rise to completely new challenges in that mobility and the underlying cellular network impact the temporal properties and the design and implementation decisions of the application. In this paper, we address the problem of synthesizing task allocation of real-time applications in the embedded-edge compute continuum, with key considerations for mobility and minimization of embedded compute utilization. We present an exact SMT-based solution as well as efficient heuristics with extensive experimental evaluations.**

***Index Terms*—embedded-edge continuum, edge offloading, 5G MEC, mobility aware, real-time task allocation**

I. Introduction

Many embedded real-time and cyber-physical systems are getting increasingly connected to cellular and wireless networks, which in turn expands real-time computing from the embedded domain to a continuum of network-edge and cloud computing resources. The opportunities and challenges increase as these systems become mobile. Prominent examples can be found in the automotive domain. A vehicle today and in the future is an increasingly complex cyber-physical system with a rapid growth of features based on electronics and software. Over time, automotive technology has seen numerous advancements, with electrification, advanced driver assistance systems, and advanced infotainment. Traditionally, automobiles have functioned as standalone systems, with all computational tasks conducted on embedded local platforms, interconnected with in-vehicle networks. For conventional functions with limited data and computational demands, this on-board computing model has been effective. However, it is becoming increasingly challenging from cost, energy, and scalability perspectives as automotive features grow rapidly [1]. Extending onboard-computing capacity directly adds to the bill-of-material cost, which is and has always been subject to major constraints for automakers. It also affects energy consumption, which is an increasingly relevant challenge due to the rapid adoption of electric vehicles. Rapid scaling of features, particularly through post-sale over-the-air software updates, requires spare computing capacity. To address the aforementioned challenges, it becomes increasingly important to minimize the embedded computing utilization in vehicle platforms.

Recent advancements in reliability and availability of cellular communication and edge computing technologies have opened up the possibility to offload computation tasks. Offloading computationally intensive applications to servers reduces the onboard computing cost and energy. By reducing the amount of onboard computing utilization, we also create more room to add future features and applications. As many automotive applications have real-time constraints it is crucial to ensure that the real-time properties of cellular network and edge computing infrastructure in making offloading decisions. For cellular communication, 5G, which is being widely deployed worldwide, promises connection speeds nearly 100 times faster than 4G [2], as well as improved reliability and availability. In terms of computational offloading, Multi-access Edge Computing (MEC) has been introduced as a component of 5G infrastructure. MEC decentralizes computing and storage by positioning resources closer to the network edge, typically via localized edge servers(ES) near the user equipment(UE). This shift transforms the centralized cloud paradigm [3], [4], greatly reducing the latency associated with cloud computing and, critically, enabling latency that is both low and predictable when combined with 5G's Ultra-Reliable Low-Latency Communication (URLLC).

Despite the technical foundations provided by MEC and various 5G technologies for edge offloading in vehicles, substantial challenges remain. First, edge servers are widely distributed geographically, with some integrated within 5G base stations(gNBs), and others existing as standalone physical entities. Combined with the high complexity of the 5G system architecture–particularly for UEs accessing the network wirelessly–accurately estimating latency from a UE to each edge server presents a significant challenge. Second, unlike typical devices, vehicles are high-speed, mobile UEs whose network environments change as the vehicle moves, leading to real-time variations in latency to different edge servers. Finally, vehicles host a wide variety of applications with dependency relationships between tasks. With offloading, a higher level of flexibility is enabled to satisfy the real-time requirements of the applications. Thus, developing an effective offloading strategy that accounts for real-time latency, optimally utilizes available edge servers, ensures task completion, and accommodates task dependencies while minimizing onboard computing utilization is a formidable challenge.

Our contributions To address these challenges, this paper first introduces a detailed 5G-MEC network model, incorporating all components that significantly impact latency. We establish an accurate latency model for connections between UEs and edge servers. Next, based on this network model and the task model of automotive applications, we motivate and formulate the problem of optimizing the offloading strategy dynamically based on the real-time location of the UE. We use Satisfiability Modulo Theories (SMT) [5] to get the optimal solution, and we further propose a heuristic-based task offloading strategy that, considering task dependencies, ensures task

2325-1301/25/$31.00 ©2025 IEEE
DOI 10.1109/RTCSA66114.2025.00017

completion while minimizing the computing load of UE. Finally, we design a set of experiments with various network conditions to evaluate our proposed offloading heuristic. Compared to the baseline method, our proposed heuristic algorithm achieves at least a 12% improvement in scheduling success rate and a 36% reduction in UE computational load. Moreover, this performance gap becomes even more pronounced under varying network conditions or larger problem scales. In contrast to the SMT-based approach, our algorithm is up to 3000 times faster on small-scale problems and remains capable of delivering results within a reasonable time frame for problem sizes that SMT cannot solve efficiently.

The rest of the paper is organized as follows. Section II reviews related research on edge computing and highlights existing research gaps. Section III introduces our system model. Section IV presents our more precise network latency model, which is one of the key contributions of this paper. In Section V, we illustrate the necessity of task offloading to edge servers under an accurate network latency model through several edge computing offloading examples, emphasizing the importance of efficient task allocation and offloading strategies. Section VI formulates the problem statement. Section VII introduces the heuristic algorithm we designed to address this problem, which is another major contribution of this paper. Section VIII provides experimental evaluations of our heuristic algorithm, comparing it with other common methods to demonstrate its advantages. Finally, Section IX concludes the paper.

II. Related Work

Currently, several studies have explored edge computing and offloading. Jošilo et al. [6] and Chen et al. [7] proposed game-theoretic models to address resource competition among multiple devices in MEC systems, minimizing a linear combination of latency and energy consumption for devices. Additionally, Saleem et al. [8] considered mobility-aware task scheduling under energy constraints, aiming to minimize the total offloading delay for all devices. Similarly, in Kao et al. 's research [9], the authors proposed an algorithm named Hermes, which minimizes latency under deadlines and energy constraints. However, these studies either overlook user mobility or assume a single edge server scenario, whereas the problem discussed here involves UE mobility and multiple edge servers in the network.

Maleki et al. [10] addressed QoS optimization in 5G edge computing, focusing on network path selection in 5G-MEC environments. The study of Liu et al. [11] considered task dependency in vehicular edge computing and introduced a priority-based task-sorting algorithm to reduce the average completion time across multiple applications. The work of Yang et al. [12] investigated edge offloading in 5G, aiming to minimize overall network energy consumption while meeting latency requirements. Moreover, Ning et al. [13] introduced a health-monitoring system dedicated to IoT healthcare in 5G-MEC environments, seeking to minimize total system cost. Finally, Aissioui et al. [14] examined network architecture for vehicles in 5G-MEC environments and proposed the Follow Me Edge Cloud (FMeC) concept. These studies advance the understanding of edge offloading in 5G-MEC environments and consider the effects of UE mobility on offloading to some extent.

However, the above studies present several limitations. First, the modeling of the 5G-MEC environment is overly simplistic, leading to an excessive simplification of network latency and inapplicability to real-time applications. Second, these studies fail to jointly consider UE mobility, multi-edge server offloading, and task dependencies. All these aspects are present in real-world scenarios. This paper bridges these gaps by establishing an accurate 5G-MEC model to capture key latency contributions, while minimizing UE computing load under conditions of mobility, multiple edge servers, task dependencies, and task deadline constraints.

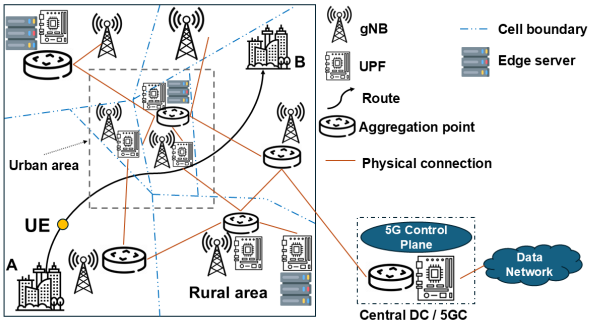

Fig. 1. The architecture of 5G-MEC environment

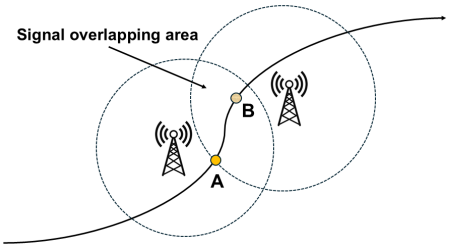

Fig. 2. Example of UE passing through the signal overlapping area

III. System Model

A. Overview of the System

We consider the 5G-MEC environment illustrated in Fig. 1. A main road connects Location A to Location B, and the UE moves along this road from A to B. The area containing Locations A and B is covered by multiple 5G base stations (gNBs), ensuring continuous signal coverage without gaps. These coverage areas overlap, enabling seamless connectivity. As the UE enters an overlapping signal region, it performs a handover from one gNB to another based on inter-gNB coordination. In the meantime, UE can do cell reselection [15] actively when no transmission happens. The boundaries naturally divide the area into multiple signal regions, where crossing a boundary (depicted as blue dashed lines in the figure) triggers a handover event or is caused by cell selection. Additionally, several edge servers are distributed within this region, serving as potential offloading destinations for UE tasks. Each edge server is closely located near a User Plane Function (UPF), which acts as the packet data unit(PDU) Session Anchor (PSA) UPF for the 5G network's edge servers. Some edge servers are also equipped with aggregation points (APs) to collect and process various network data. UPFs and APs may also be co-located near 5G base stations. Apart from the RAN wireless connection between the UE and gNBs, all other network components are interconnected via optical fiber.

As shown in Fig. 2, when the UE enters a signal overlapping region—at point A—it first retrieves necessary information from the network, including the connection speed to the next cell tower, the connectivity to available edge servers, and the associated latencies. The UE then executes a predefined scheduling algorithm. If a feasible

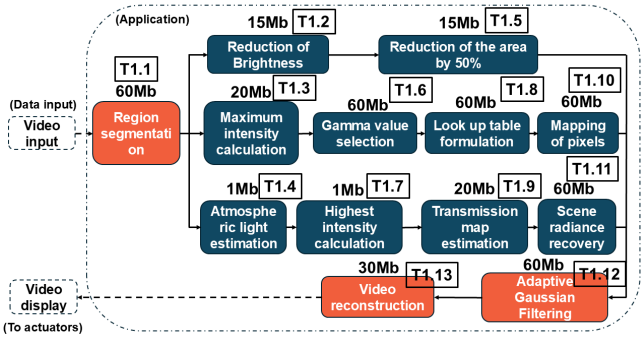

Fig. 3. Example of night vision application with data output size.

scheduling strategy is determined by the time the UE reaches point B, it will wait for all currently executing applications to complete and their data to be transmitted before performing cell reselection to connect to the next cell tower. The UE will then switch to the new scheduling strategy and continue executing tasks until it reaches the next overlapping region, where this process is repeated.

B. Edge Server Model

As previously mentioned, edge servers are geographically distributed and are connected to other network infrastructure via optical fiber. UEs within this 5G network environment can access these edge servers through the UPFs linked to them. We assume that the computational capacity of the edge servers is sufficient to support all UEs in the area without resource contention. For a given edge server E_i, F_i represents the processing speed it can provide to UEs, measured in CPU cycles per second.

C. Task Model

A UE typically runs multiple applications concurrently, each with its own period and deadline. These applications generally involve data input (e.g., sensor data) and output to actuators, and each consists of multiple interdependent tasks, where the output of one task may serve as the input for another. Tasks are the smallest units that can be executed locally or offloaded to edge servers, and while applications are periodic with deadlines, individual tasks do not have individual deadlines.

At the start of an application's period, its initial task begins, and the application completes when its final task finishes. For example, Fig. 3 illustrates a typical ADAS application, Night Vision [16], which processes video input to enhance dark areas and reduce glare from oncoming headlights. In Fig. 3, video data is first divided into regions, then processed by three parallel task chains, followed by adaptive Gaussian filtering and reassembly before output. Fig. 3 could also be translated into a task graph, ignoring the details of each task, but only showing dependencies among tasks and their data size. Here, as an instance, $T1.13$ refers to the thirteenth task in the first application.

D. Scheduling and Resource Allocation Model

The objective of task allocation is to ensure that all applications on the UE are completed before their deadlines while satisfying the dependency constraints between tasks. Each task can be executed either locally on the UE or on any edge server within the region. Since tasks have dependencies, data transmission time between tasks must also be considered. The following three cases arise when task T_a depends on T_b meaning T_b must wait for the output of T_a before execution:

Case A: If both T_a and T_b are executed on the UE or the same edge server, we consider the data transmission time to be negligible. Or it is a straightforward extension to model a worst-case intra-processor task communication delay, or consider it as a part of the task execution time, if needed.

Case B: If one task is executed on an edge server while the other runs on the UE, the transmission time for transferring data between T_a and T_b over the network must be considered.

Case C: If T_a and T_b are executed on different edge servers, the data exchange time between these edge servers must be taken into account.

It is worth noting that since the majority of automotive applications receive data input from sensors and ultimately output results to actuators, the first task of an application, which reads data from sensors, and the final task, which sends control signals to actuators, must always be executed on the UE and cannot be offloaded to an edge server. For exceptional cases where an application does not require sensor input or actuator output, virtual tasks can be added to standardize the application model. Tasks without dependencies can be executed concurrently. Additionally, we assume that both the UE and edge servers execute tasks sequentially. That is, tasks are executed in the order they are assigned, meaning tasks scheduled earlier will be executed first. Consequently, the execution platform and execution order of all tasks are determined by the specific allocation strategy applied during task assignment.

IV. Network Latency Model

An accurate network latency model is fundamental to achieving efficient task allocation and scheduling. As discussed in the previous chapter, due to task dependencies, cases B and C, as well as the scenario where the final result is transmitted back to the UE, all involve the transfer of task input and output data over the 5G network. Without a precise network model to calculate this latency, applications may fail to meet their deadline constraints. Furthermore, as a safety-critical system, a vehicle imposes strict requirements on system determinism. As previously mentioned, existing studies often rely on coarse-grained network latency models. To address this limitation, we introduce a more precise network model in this section.

A. Latency between UE and gNB

UEs are mobile and connect to nearby gNBs, such as vehicles traveling on fixed routes (e.g., highways from Location A to Location B). This wireless link contributes to end-to-end latency, with communication rate modeled using Orthogonal Frequency-Division Multiple Access(OFDMA) with MIMO and carrier aggregation. The communication rate is given by the following equation:

$$R = \sum_{j=1}^{J} [\nu_{Layers}^{(j)} \cdot Q_m^{(j)} \cdot f^{(j)} \cdot M \cdot \frac{N_{PRB}^{BW(j),\mu} \cdot 12}{T_s^{\mu}} \cdot (1 - OH^{(j)})] \quad (1)$$

where

$$T_s^{\mu} = \frac{10^{-3}}{14 \cdot 2^{\mu}},$$

where the sum operation means to calculate the data rate of each aggregated component carrier separately and accumulate them, and J means all the carriers. $\nu_{Layers}^{(j)}$ represents the number of layers in MIMO. $Q_m^{(j)}$ represents modulation order. $f^{(j)}$ represents the scaling factor (in practical settings, often 0.4, 0.75, 0.8, or 1.0) [17]. These parameters all depend on the processing capabilities and

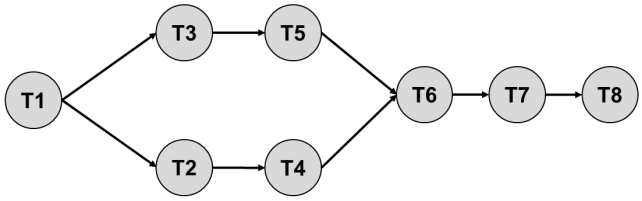

Fig. 4. A UE example application for offloading

configuration of the UE baseband unit. M represents target code rate of modulation. N represents the maximum number of Resource blocks (RBs) allocated by the cell tower to the UE. The constant 12 means that each RB has 12 subcarriers. T_s^μ represents the maximum number of Orthogonal Frequency Division Multiplexing(OFDM) symbols in each subframe ($\mu = 0, 1, 2, ...$). OH represents the overhead caused by OFDMA communication. Thus, assuming that a data packet of size p is transmitted between the UE and its connected gNB, the network transmission latency in this case is given by: $D_{UE}^p = \frac{p}{R}$.

B. Latency of UPF and AP

UPF plays a critical role in establishing PDU sessions and facilitating communication between UEs and designated data networks (DNs). We assume that UPFs are generally located near gNBs or close to edge servers and aggregation points. As illustrated in Fig. 1, when a UE sends data to a gNB, the data packet passes through one or more UPFs before reaching the DN. The UPF directly connected to the DN is termed the PDU session anchor (PSA) and is indispensable, while other UPFs can serve as relays or splitters as per SMF policies. We denote the latency incurred by each UPF during packet processing as D_{UPF}.

AP aggregates data to enhance network efficiency, each incurring a latency D_{AP}.

Except for UE-gNB communication, other network components are interconnected via optical fiber. We assume a propagation speed of $2 \times 10^8 m/s$ for signals within the fiber, with transmission distance calculated as the Euclidean distance between two points. The latency for signal transmission from point a to point b over fiber of transmitting a packet with size p is represented as $D_l^p(a, b)$.

C. End to End Latency

Based on this model, the end-to-end latency $d_{a,b}^{e2e}$ for a data packet of size p sent from node a to node b is given by:

$$d_p^{e2e}(a, b) = \frac{p}{R} \cdot \sigma(node) + \sum_{a}^{b}(D_{UPF} + D_{AP}) + D_l^p(a, b) \quad (2)$$

where $\sigma(node)$ is a function that the value equals to 1 if $node = UE$, otherwise equals to 0, such as edge servers. Here, the summation symbol indicates that, following the network path from node a to node b, the total network transmission delay is calculated by accumulating both the processing delay at each intermediate node and the transmission delay between nodes along the path.

V. Motivation Example

In this section, we illustrate the benefits of offloading vehicle tasks to edge servers through several examples. Additionally, we highlight the importance of efficient task allocation and scheduling when considering an accurate network latency model during offloading.

First, we consider a simple application as an example. As shown in Fig. 5, this application consists of eight tasks. T1 is the initial task,

TABLE I
Detailed Task Parameters

Task No.	Execution Time (ms)			Output Data Size (Mb)
	UE	ES A	ES B	
1	10	-	-	20 (1→2), 10 (1→3)
2	20	10	24	30
3	25	15	12	20
4	30	20	24	20
5	30	24	20	30
6	20	32	25	20
7	10	24	22	10
8	10	-	-	-

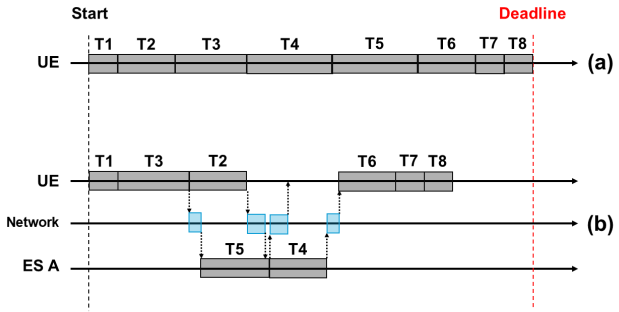

Fig. 5. Offloading examples. (a). Scheduled on UE without offloading. (b). Intuitive offloading strategy to reduce the computing load of UE.

marking the application's start. Both T2 and T3 depend on the output of T1 as their input, while T4 depends on T2, and T5 depends on T3. Task T6 depends on both T4 and T5, while T7 depends on T6, and finally, T8, the completion task, depends on T7.

Since the application interacts closely with sensor input and actuator output, T1 and T8 must be executed locally on the UE. The remaining tasks can be allocated to either the UE, ES A, or ES B. The execution time of each task on different platforms, along with output data sizes, is presented in Tab. I. The network latency introduced by data transmission is given by the following equation:

$$\begin{cases} 0.02 + \frac{Data\ size}{5}, & \text{UE} \leftrightarrow \text{Edge server A} \\ 0.1 + \frac{Data\ size}{1}, & \text{UE} \leftrightarrow \text{Edge server B} \\ 0.05 + \frac{Data\ size}{0.8}, & \text{Edge server A} \leftrightarrow \text{Edge server B} \end{cases}$$

For this application, all tasks can be executed locally on the UE,

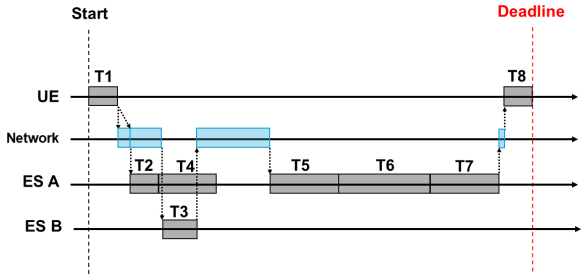

Fig. 6. Optimized offloading strategy minimizing UE computing load.

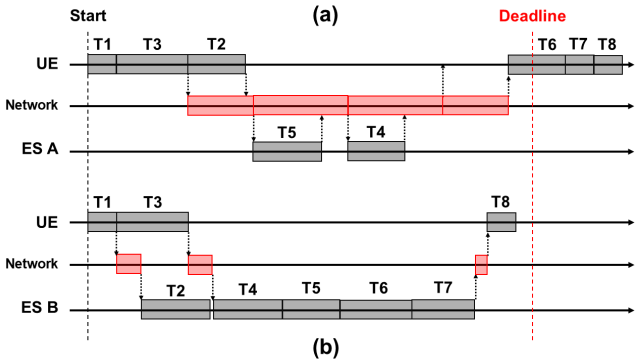

Fig. 7. Offloading examples in another scenario. (a). Failed to meet the deadline with the previous offloading strategy. (b). Feasible offloading strategy

or some may be offloaded to edge servers. The application has a period of 16 ms and a deadline of 15.5 ms, meaning that all tasks must be completed within 15.5 ms once execution begins. As shown in Fig. 5.a, if all tasks are executed locally, the application meets the deadline. However, this solution fully occupies the UE's computational resources. It is often highly desired to reduce the computing utilization as that creates room for future software upgrades and reduces energy consumption, to name a few examples. To free up computational capacity, a natural approach is to offload certain tasks to edge servers, allowing parallel execution. However, since data transmission incurs additional latency, the scheduling strategy should aim to overlap network transfer with computation.

As illustrated in Fig. 5.b, given that Edge Server B has lower performance and higher network latency compared to Edge Server A, we offload T5 and T4 to Edge Server A, while keeping the remaining tasks on the UE. The scheduling order is adjusted to maximize the overlap between data transmission and computation. This results in a 20 percent reduction in UE computing load, while satisfying the deadline constraint. Although this offloading strategy effectively leverages parallel execution and overlapping data transfer, further reduction in UE computation load can be achieved by considering a different offloading strategy. Towards this, Fig. 6 presents a revised scheduling plan where the UE's computing load is further minimized, while the application completes precisely at the deadline, making full use of the available execution window. Even though Edge Server B has lower performance and higher network latency compared to Edge Server A, the strategy ensures network latency overlaps with task execution, maintaining a feasible schedule. Additionally, placing T6 and T7 on the same edge server reduces inter-server communication delays. This demonstrates that minimizing UE computation is a complex design problem and requires a well-designed scheduling strategy.

While the above examples illustrate the optimal scheduling strategy, that strategy is limited to the UE location being in a given cell boundary. As UEs are mobile and transition from one cell boundary to another, the network conditions change and, therefore, also the communication delays among the UE and the edge servers. The offloading strategy in Fig. 6 is only effective in a given region along the UE's route. For example, in Fig. 7, if the UE moves into a different area, it may still offload tasks to Edge Servers A and B, but the network latency changes. Here, the UE to Edge Server A network conditions deteriorate significantly, while latency to Edge Server B worsens slightly. The latency between Edge Servers A and B remains unchanged. If we continue using the Fig. 5.b strategy, as shown in Fig. 7.a, the application now fails to meet its deadline and needs a different offloading strategy that is customized for the real-time network performance in the UE's new region.

To adapt to the new UE location, Fig. 7.b presents an alternative offloading strategy. Given the severe degradation in UE to Edge Server A latency (compared to the previous UE location), communication between the UE and Edge Server A is now minimized. Although Edge Server B has lower performance, it becomes the better choice in the new UE location (due to improved network conditions). With appropriate adjustments, the new offloading strategy ensures that the application meets its deadline while keeping the UE's computing load as low as possible.

From this example, it is clear that offloading strategies are highly influenced by UE mobility and must be adaptively optimized to maintain performance under varying network conditions. Combined with the previous examples, this highlights that for a mobile UE, a fixed offloading strategy is sufficient only within a portion of the end-to-end route of the UE. As the UE moves along its route, the offloading strategy needs to change. Knowing the cell boundaries, a UEs route can be divided into segments such that, in a given segment, a fixed offloading strategy meets the deadline constraints. The UE will dynamically switch the offloading strategy as it moves along the route. We will now formulate the scheduling and task offloading problem to be solved for a given segment of the UE's route.

VI. Problem Formulation

In this paper, we aim to minimize the computing load of a UE over the hyper-period of all its applications. We first define a decision variable:

$$\omega_{m,i} = \begin{cases} 0, & \text{scheduled on UE local;} \\ 1, 2, 3, ..., & \text{scheduled on edge server,} \end{cases}$$

where $\omega_{m,i}$ indicates the platform on which the i-th task of the m-th application is assigned. A value of 0 denotes execution on the UE itself, while each edge server has a fixed identifier representing offloading to a specific edge server. Additionally, we introduce a function $\varepsilon(m, i, \omega_{m,i})$, which provides the execution time of the i-th task of the m-th application on platform $\omega_{m,i}$. Lastly, we define an indicator:

$$\psi(x, y) = \begin{cases} 0, & x = y; \\ 1, & x \neq y. \end{cases}$$

whose value is 1 if platform x is the same to platform y, otherwise 0. As we assume that the CPU frequency of the UE is constant, the computing load of the UE is defined as the time used by all tasks executed on the UE in a hyperperiod. Thus, our problem can be represented by the following mathematical formulation:

Algorithm 1: EADDS Algorithm - Overview

Input: A: a set of applications
P: a set of platforms
$G_a = (V_a, E_a)$: Each appliocation $a \in A$ is represented by as task graph
T_a, D_a: period and relative deadline of each appliction
$d_p^{e2e}(p,q)$: for each pair $(p,q) \in P^2$, the end-to-end latency by transmitting a packet with data size p
λ: UE penalty factor
μ: Edge saturation adjustment parameter
α, β: Priority weight parameters

1 Preprocessing stage
2 Scheduling main loop

Output: A schedule for each task instance t^j: Start time $S(t^j)$, finish time $F(t^j)$, and assigned platform $p(t^j)$

$$\text{minimize} \quad \sum_{w} \omega_w(t),\ w \in W_0 \tag{3}$$

$$\begin{aligned}\text{subject to} \quad & t_{T_{m,i}} \geq t_{T_{m,h}} + d_p^{e2e}(a,b) \cdot \psi(\omega_{m,h}, 0) \\ & + d_p^{e2e}(a,b) \cdot \psi(0, \omega_{m,i}) \\ & + \varepsilon(m,h,\omega_{m,h}), h \in H\end{aligned} \tag{4}$$

$$d_m \geq t_{T_{m,-1}} + \varepsilon_{m,-1} \tag{5}$$

$$a, b = f(\omega_{m,i}) \tag{6}$$

where $\omega_w(t)$ means the execution time of task w, and W_0 is the set including all the tasks allocated on UE. $t_{T_{m,i}}$ is the start time of the i-th task of the m-th application. $T_{m,h}$ is the predecessor dependency task of $T_{m,i}$. d_m is the deadline constraint of the m-th application. Eq. (4) is the relative task dependency constraint, indicating that for two dependent tasks, the start time of the subsequent task must not be earlier than the start time of the preceding task plus its execution time, along with the network transmission delay for the output data if any. H is the set including all the tasks with predecessors. Eq. (5) is the application deadline constraint, indicating that the completion time of the final task in an application must not exceed the application's deadline. Here, $t_{T_{m,-1}}$ is the start time of the last take in application m. $\varepsilon_{m,-1}$ means the execution time of the last task in application m on its assigned platform. Eq. (6) indicates that the specific values of platforms a and b are determined by the decision variables, that is, the concrete scheduling result of each task.

Based on the above problem formulation, we implemented an SMT solver using Z3 [5] to theoretically obtain the optimal solution for this problem. Since the SMT solver directly aligns with the problem formulation, we do not elaborate on its details here. In the following section, we introduce a heuristic algorithm specifically designed to address this problem, aiming to overcome the slow execution speed of traditional SMT solvers while maintaining sufficient performance. Furthermore, in Section VIII, we compare the performance of the SMT solver with our proposed heuristic algorithm.

VII. Heuristic Task Assignment Algorithm

Since the scheduling problem is a combinatorial optimization problem known for its high computational complexity and the difficulty of

Algorithm 2: EADDS Algorithm - Preprocessing phase

Initialize execution time $\omega_p(t)$, estimated communication time from task t to task s $c(t,s)$
Initialize hyperperiod H, release time $R(t^j)$, absolute deadline $D(t^j)$
Initialize upward rank $rank_{up}(t)$, priority weight $P(t)$, priority queue Q
for *Each task t and platform $p \in P$* **do**
  $\omega_p(t) \leftarrow \frac{compute\_load(t)}{capacity(p)}$
endfor
for *Each application $a \in A$* **do**
  for *Each task $t \in V_a$* **do**
    if *task t has successor* **then**
      $c(t,s) \leftarrow \overline{d_{t.output\_data\_size}^{e2e}(p,q)}$, $p, q \in P^2$
    end
  endfor
endfor
Decide C according to $\overline{\omega_p(t)}$ and $\overline{c(t,s)}$
if $C > 1$ **then**
  for *Each application instance $a \in A$, given the topologically sorted list V_a of task instances* **do**
    for *i from 0 to V_a in steps of C* **do**
      Set a macro task $M \leftarrow$ $V_a[i], V_a[i+1], ...V_a[min(i+C-1, |V_a|-1)]$
    endfor
    Update dependency graph G_a with M
  endfor
end
$H \leftarrow lcm\{T_a : a \in A\}$
for *each application $a \in A$ and task instance index $j = 1, ..., \frac{H}{T_a}$* **do**
  for *each task $t \in V_a$* **do**
    Set a task instance t^j.
    Copy meta data from t to t^j
    $R(t^j) \leftarrow (j-1) \times T_a$; $D(t^j) \leftarrow R(t^j) + D_a$
    Initialize the unsatisfied predecessor count for t^j according to its incoming edges
  endfor
endfor
for *Each task t^j in reverse topological order* **do**
  if *t^j has no successor* **then**
    $rank_{up}(t^j) \leftarrow \omega_{UE}(t^j)$
  else
    $rank_{up}(t^j) \leftarrow \omega_{UE}(t^j) + max_{s \in succ(t^j)}\{c(t^j,s) + rank_{up}(s)\}$
    $P(t^j) \leftarrow \alpha \cdot Normalize(rank_{up}(t^j)) + \beta \cdot \frac{1}{max(D(t^j)) - rank_{up}(t^j), \varepsilon}$
  end
endfor
for *Each platform $p \in P$* **do**
  Initialize $CPU\_free(p) \leftarrow 0$, $SEND\_free(p) \leftarrow 0$, $RECV\_free(p) \leftarrow 0$, $Load(p) \leftarrow 0$
endfor
Sort $P(t^j)$ in descending order **for** *Each t^j* **do**
  if *t^j with zero unsatisfied predecessors and $R(t^j) \leq 0$* **then**
    Append t^j in Q; $t^j.state \leftarrow ready$
  end
endfor

finding exact solutions in a reasonable amount of time, as previously mentioned, the SMT-based solution can theoretically provide the optimal result. However, its runtime grows rapidly, making it impractical to obtain solutions within an acceptable time frame as the problem size increases. To address this issue, we propose a heuristic algorithm named Edge-Aware Distributed Deadline Scheduler (EADDS), which aims to produce solutions in a significantly shorter time while achieving results that are as close as possible to the theoretical optimum.

A. Overview

The structure of EADDS is shown as Algo. 1. First, the input to EADDS consists of the following components: a set of applications A, where each application $a \in A$ contains a task graph represented as a DAG, $Ga = (V_a, E_a)$. These task graphs include all relevant task information such as computational load, output data size, and dependency relationships. Each application also has its own period T_a and relative deadline D_a. The platform set p includes at least one UE and potentially several edge servers (possibly zero), with known computational capacities for each platform. The end-to-end network latency $d_p^{e2e}(a, b)$ is given for transmitting a data packet of size p between any two platforms in the network. Additionally, EADDS takes four groups of tunable parameters related to scheduling performance: the UE penalty factor λ, the edge server load adjustment factor μ, task weight coefficients α and β, and an aggregation factor C.

The algorithm consists of two main phases. The first phase is the preprocessing stage. In this phase, EADDS aggregates the tasks within each application based on the aggregation factor C, forming new macro-tasks and a modified task graph. Then, all applications' multi-period task graphs are unfolded over their hyper-period and aligned on a unified timeline to facilitate scheduling. After that, EADDS computes and sorts the priorities of all tasks within the hyper-period. Finally, the system state is initialized to prepare for scheduling.

The second phase is the scheduling phase. In this stage, the algorithm iteratively evaluates all feasible scheduling options for the next task, taking into account the task's properties, platform status, and current system state. It selects the best candidate among the feasible options, performs the scheduling, and updates the system state. This process repeats until all tasks have been scheduled, marking the end of the algorithm.

At the end, EADDS outputs a complete schedule that includes the start time, end time, and execution platform for every task within the hyper-period across all applications.

B. Preprocessing Phase

The preprocessing phase of EADDS is detailed in Algo. 2. The algorithm begins by initializing all variables required in later stages. Lines 4 to 22 handle task aggregation. First, the execution time of each task on every platform, as well as the average network latency for data transfer between any two dependent tasks, will be computed. The value of the aggregation factor C is determined based on the average task execution time and the average network latency. When the average execution time is close to or greater than the average network latency, a small C, or even $C = 1$, is preferred. In contrast, when the average network latency is significantly higher than the average execution time, a larger C should be used. If the aggregation factor C provided as input is greater than 1, task aggregation is performed. Starting from the initial task of each application, the algorithm traverses along dependency paths (i.e., edges in the DAG)

Algorithm 3: EADDS Algorithm - Scheduling main loop

Initialize variables: current time $t_{curr} \leftarrow 0$, candidate start time S_p, scheduled start and finish time of task $S(t)$, $F(t)$, the time when data from x reaches platform p $T_{comm}(x, p)$, indictor $I_{\{p\}}$, edge server load threshold θ
while *Q is not empty or unscheduled tasks remain* **do**
  if *Q is empty* **then**
    advance t_{curr} to the smallest release time among unscheduled tasks and insert those tasks into Q
  end
  Select the hightest-priority task t from Q
  for *Each platform $p \in P$* **do**
    $S_p \leftarrow max\{t_{curr}, CPU\_free(p), R(t^j)\}$
    for *Each predecessor x of t^j* **do**
      if $p(x) \neq p$ **then**
        $T_{comm}(x, p) \leftarrow max\{F(x), SEND\_free(p(x)), RECV\_free(p)\} + d^{e2e}_{x.output\_data\_size}(p(x), p)$
      end
      Update $S_p \leftarrow max\{S_p, T_{comm}(x, p)\}$
    endfor
    $F_p \leftarrow S_p + \omega_p(t^j)$
    if *$F_p > D(a^j)$, where $t^j \in a^j$* **then**
      break
    else
      $Cost(p) = F_p + I_{\{p=UE\}} \cdot \lambda \cdot \omega_{UE}(t) + I_{\{p=edge\}} \cdot \mu \cdot max\{Load(p) - \theta, 0\}$
    end
  endfor
  Choose the platform $p*$ with minimum cost ($Cost(p)$) among the feasible platforms.
  Set $p* \leftarrow UE$ as a fallback if no platform is feasible
  $S(t^j) \leftarrow S_{p*}$
  $F(t^j) \leftarrow F_{p*}$
  $p(t^j) \leftarrow p*$
  $CPU\_free(p*) \leftarrow F(t^j)$
  $Load(p*) \leftarrow Load(p*) \leftarrow Load(p*) + \omega_{p*}(t^j)$
  for *Each predecessor x with $p(x) \neq p*$* **do**
    $SEND\_free(p(x)) \leftarrow max\{SEND_free(p(x)), F(x)\} + d^{e2e}_{x.output\_data\_size}(p(x), p*)$
    $RECV\_free(p(x)) \leftarrow max\{RECV_free(p(x)), SEND_free(p(x))\} + d^{e2e}_{x.output\_data\_size}(p(x), p*)$
  endfor
  for *Each successor y of t^j* **do**
    decrement its unsatisfied predecessor count
    if *unsatisfied predecessor count is 0 and $R(y) \leq F(t^j)$* **then**
      insert y into Q
    end
  endfor
  $t_{curr} \leftarrow S(t^j)$
end

and merges every C consecutive dependent tasks into a new macro-task. The computational load of a macro-task is the sum of the merged tasks, and it inherits all relevant dependencies. This process continues until all tasks in each application are aggregated, forming a new application that retains the original period and relative deadline.

The main motivation behind task aggregation is to improve scheduling performance, especially under conditions of high network latency and lightweight computation. When dependent tasks are assigned to different platforms, data must be transferred over the network. In scenarios where network delay dominates task execution time, grouping dependent tasks onto the same platform helps mitigate the negative impact of transmission delays. Additionally, when the number of applications and tasks becomes large, aggregation helps reduce the scheduling complexity and improve execution speed.

Lines 22 to 31 compute the hyper-period of all applications and unfold each application into multiple instances aligned on a unified timeline. These instances inherit metadata from their original application. Notably, the start and deadline times of each instance are converted into absolute values on the hyper-period timeline.

Lines 32 to 39 calculate the priority of each unfolded task instance t^j. Task priority is determined based on two factors: the upward rank ($rank_{up}$) and the slack. If a task has no successors (i.e., it is a terminal task), its $rank_{up}$ is simply its execution time on the UE. For other tasks, the $rank_{up}$ is computed as the execution time on the UE plus the maximum critical path length to a terminal task, including network transfer delays. Slack reflects the urgency of a task with respect to its application's deadline—the smaller the slack, the higher the urgency and the earlier it should be scheduled. The final task priority is a weighted combination of the normalized $rank_{up}$ (scaled by coefficient α) and the normalized slack (scaled by coefficient β).

Finally, lines 40 to 48 initialize the system state for each platform, including CPU usage, data transmission and reception channels, and current load. All tasks are then sorted by priority, and tasks whose dependencies have been satisfied are added to the ready queue.

C. Scheduling Main Loop

The scheduling main loop of EADDS is described in Algorithm 3. Similar to the preprocessing phase, the algorithm begins by initializing all variables that will be used later. The scheduling time indicator variable t_{curr} is set to 0, indicating that scheduling starts from the beginning of the hyper-period. The loop continues until all tasks have been scheduled.

The algorithm first checks the ready queue Q. If no tasks are ready—either because no task at the current time t_{curr} has all its dependencies satisfied, or because the current time is earlier than the start time of any upcoming task—then t_{curr} is advanced to the earliest start time of any unscheduled task. At that time, all newly ready tasks are added to the queue Q. From this queue, the task with the highest priority (precomputed during preprocessing) is selected for scheduling.

The core scheduling process proceeds as follows. For each platform p, the algorithm computes the cost of assigning the selected task to that platform. As shown in lines 8 to 15, the earliest possible start time S_p on platform p is calculated as the maximum of three values: the current time t_{curr}, the earliest time the platform's CPU is free, and the task's earliest allowed start time.

Next, for each predecessor task x of the selected task, if x was assigned to a platform different from p, then the communication arrival time t_{comm} is determined. This is calculated as the maximum of: the finish time of task x, the availability of the sending channel on x's platform, and the receiving channel on p, plus the network transmission time between the two platforms. The start time S_p is then updated to the larger of its current value and T_{comm}.The finish time F_p is calculated as S_p plus the execution time of the task on platform p. If F_p exceeds the application's deadline, platform p is deemed infeasible for this task. Otherwise, the platform's cost function is computed.

TABLE II
SETTING OF SMALL SCALE AND LARGE SCALE

	Small Scale	**Large Scale**
Number of applications	10	100
Number of tasks	5	50
Network delay range	20-40 ms	40-60 ms
Number of edge servers	3	5
Execution time of ES range	10-30 ms	30-50 ms

As shown in line 19, the cost function includes three components: $I_{\{p\}} = 1$ if p is the UE, otherwise 0. This term, weighted by the penalty coefficient λ, increases the cost when the task is assigned to the UE, encouraging offloading to edge servers to reduce local load. $Load(p)$ is the platform's current load, μ is the load adjustment coefficient, and θ is a predefined load threshold. This term increases the cost for overloaded platforms, helping avoid excessive concentration of tasks on a single edge server. After computing the cost for all feasible platforms, the algorithm selects the platform with the minimum cost and assigns the task accordingly.

Lines 23 to 39 handle system state updates. Once a task is assigned, the corresponding platform's CPU, send, and receive channels are updated. Any tasks whose dependencies are now satisfied are added to the ready queue Q. Finally, t_{curr} is advanced to the next event time, and the process repeats until all tasks are scheduled.

VIII. Performance Evaluation

In this section, we conduct a comprehensive experimental evaluation of the performance of the proposed EADDS algorithm. Our evaluation focuses on three key metrics: algorithm runtime, scheduling success rate, and UE computational load in successful scheduling cases. As previously mentioned, although the SMT-based approach can provide a theoretically optimal solution, its runtime is prohibitively long, and increases exponentially with the problem scale. In our model, the UE is expected to execute the scheduling algorithm and complete the pre-scheduling process within a short period of time (e.g., while passing through the overlapping coverage area of two gNBs). Therefore, it is essential to ensure that the algorithm's runtime falls within an acceptable and practical range. The scheduling success rate is also a critical metric. We aim for the algorithm to consistently produce a feasible schedule—otherwise, task offloading from the UE becomes impossible. Finally, minimizing the UE's computational load remains our primary optimization goal. We expect the algorithm to reduce the UE's computing burden as much as possible while still ensuring successful task scheduling.

A. Experimental Setup

To evaluate the three key metrics, we designed a series of experiments. Regarding the scale of the scheduling problem, we defined two categories: small-scale and large-scale problems. The specific parameter configurations are as follows: $(\alpha, \beta) = (0.5, 0.5)$, $\lambda = 0.5$, and $\mu = 0.25$. In small-scale problems, both the number of applications and tasks is limited, resulting in relatively lower scheduling complexity. In contrast, large-scale problems involve a significantly higher number of applications and tasks, making the

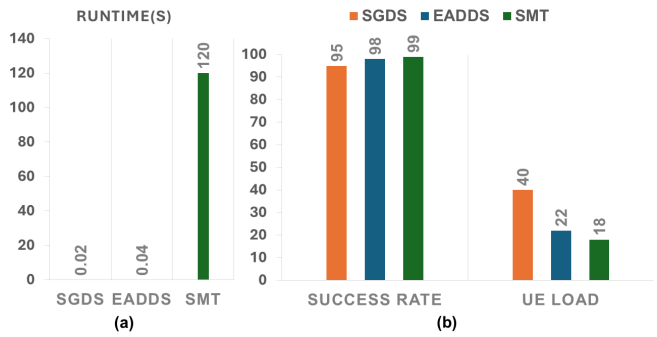

Fig. 8. Comparison under small-scale problem among SGDS, EADDS, and SMT. (a). Execution time. (b). Success rate and UE load

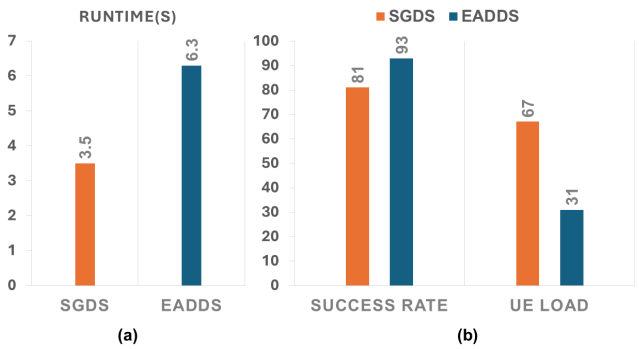

Fig. 9. Comparison under large-scale problem between SGDS and EADDS. (a). Execution time. (b). Success rate and UE load.

scheduling process more challenging. All other parameters, including task dependencies and application deadlines, are randomly generated. For small-scale problems, due to the manageable number of tasks, we use the SMT-based solution as a baseline. The SMT results serve as the theoretical optimum for comparison, enabling a direct evaluation of EADDS performance. For large-scale problems, since their size far exceeds what an SMT solver can handle within a reasonable time frame, we instead introduce a simple, intuitive greedy algorithm as a baseline for performance comparison. This baseline method is referred to as Simple Greedy Deadline-based Scheduling (SGDS). It consists of the following steps: 1. Unfold all DAGs across the hyper-period. 2. Sort all tasks based on their start times and schedule them in order. 3. For each task, calculate its earliest finish time on all platforms, prioritizing edge servers, and only consider UE as a last resort to minimize UE computation. 4. Assign the task to the platform that yields the earliest completion time and repeat the process until all tasks are scheduled.

In addition to evaluating EADDS across different problem scales, we also assess its robustness under other conditions that may influence performance. First, we consider network latency, which significantly affects scheduling outcomes and increases complexity. To test EADDS under varying network conditions, we conduct a set of experiments with both small- and large-scale task sets, varying only the network latency from low to high, while keeping all other parameters constant. This allows us to observe both the algorithm's performance and its stability under different network environments. We also evaluate the impact of the number of edge servers, which, like network latency, influences scheduling complexity and outcomes. In this experiment, we vary the number of edge servers while keeping all other parameters fixed to assess the algorithm's adaptability. Finally, we examine how EADDS performs as the problem size continually increases. In this experiment, we gradually scale up the problem size to identify the limits within which EADDS can still deliver acceptable performance.

For all experiments mentioned above, except for the fixed configurations in Tab. II, other task parameters are randomly generated. To ensure statistical significance and generality, for each experimental setting, we generate 50 different task sets, and report the average results across these sets. The complete list of parameter values used for EADDS in the experiments is shown in Table Tab. I.

B. Result and Analysis

As shown in Fig. 8, we first compare the performance of SGDS, EADDS, and SMT under small-scale problems. It is evident that

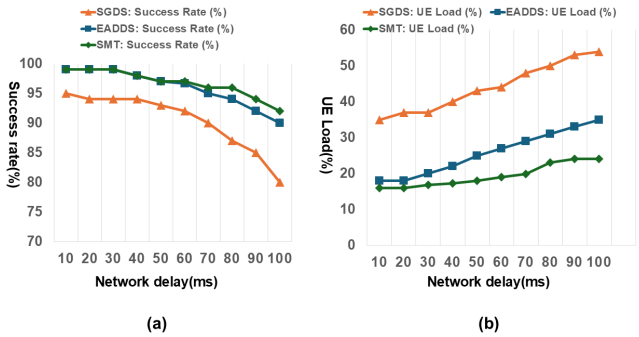

Fig. 10. Comparison under small-scale problem with different network delay among SGDS, EADDS, and SMT. (a). Success rate. (b). UE load.

both SGDS and EADDS, which do not involve global search, exhibit very short runtimes and significantly outperform SMT in terms of speed. Regarding scheduling success rate (Fig. 8.b), since the problem scale is relatively small and less complex, all three algorithms achieve high success rates, with EADDS closely approaching the theoretical optimum given by SMT. In terms of UE computational load, SGDS performs poorly as it does not include any optimization mechanisms, resulting in high UE load. In contrast, EADDS achieves much lower UE load, with results close to the optimal values produced by SMT. For large-scale problems, as shown in Fig. 9, the runtime of both SGDS and EADDS increases with problem size. However, because their growth trends are approximately linear, the runtime remains

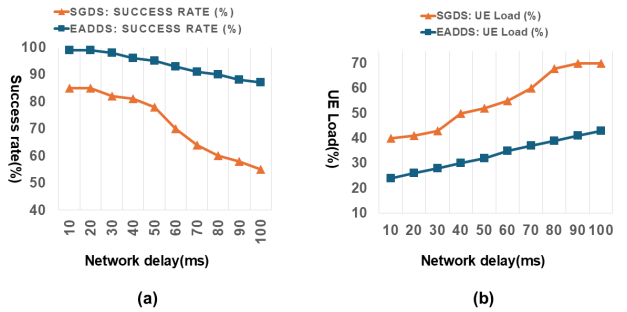

Fig. 11. Comparison under large-scale problem with different network delay between SGDS and EADDS. (a). Success rate. (b). UE load.

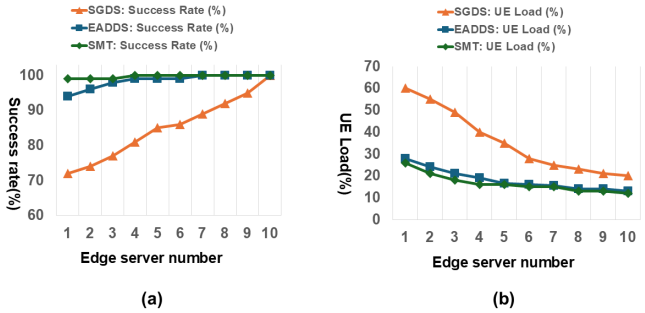

Fig. 12. Comparison under large-scale problem with different ES number among SGDS, EADDS, and SMT. (a). Success rate. (b). UE load.

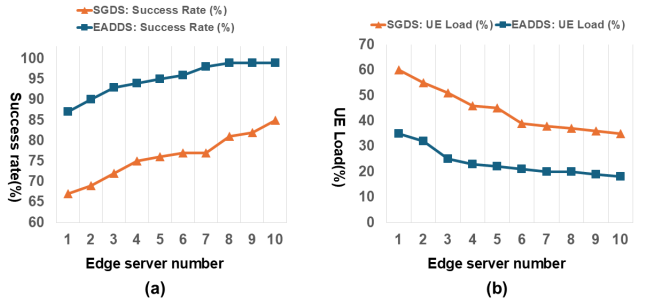

Fig. 13. Comparison under large-scale problem with different ES number between SGDS and EADDS. (a). Success rate. (b). UE load.

within an acceptable range. Due to additional optimization steps, EADDS takes slightly longer to run than SGDS. Nonetheless, in terms of scheduling success rate and UE computational load, EADDS significantly outperforms SGDS, demonstrating that its optimization mechanism is effective. Despite the increased problem size, EADDS maintains a high success rate and manages to prevent excessive growth in UE load.

Figures Fig. 10 and Fig. 11 illustrate how average network latency affects algorithm performance under different problem sizes. For small-scale problems, EADDS performs well and remains close to SMT, and under all conditions, EADDS consistently outperforms SGDS. This demonstrates EADDS's adaptability and robustness across varying levels of network latency and problem size. It is worth noting that under small-scale settings, EADDS's UE load

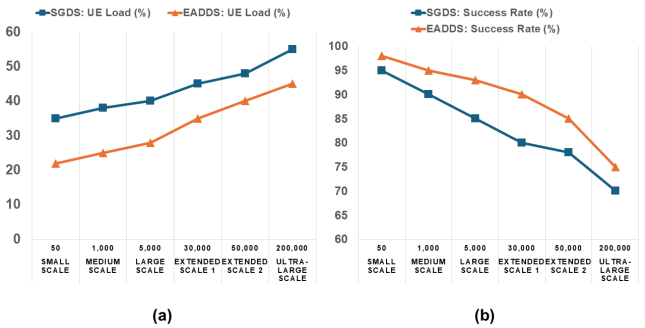

Fig. 14. Comparison under different problem scales between SGDS and EADDS. (a). UE load. (b). Success rate.

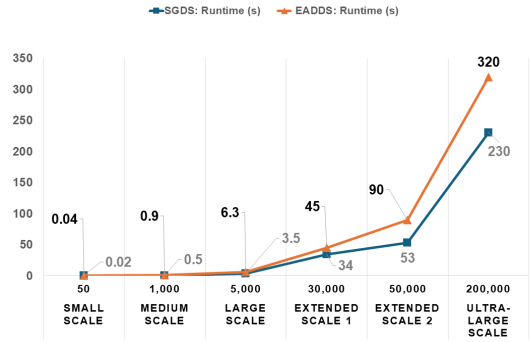

Fig. 15. Comparison of execution time under different problem scales between SGDS and EADDS.

increases faster than SMT's as network latency grows. A reasonable explanation is that, when latency becomes too large, the penalty coefficient λ, used to discourage UE allocation, becomes insufficient to offset the increased communication cost of using edge servers. As a result, more tasks are assigned to the UE. In such cases, increasing λ can effectively reduce UE load and improve performance.

Figures Fig. 12 and Fig. 13 show how the number of edge servers impacts performance. The results demonstrate that EADDS performs consistently well. SGDS shows inferior performance compared to both EADDS and SMT across all scenarios. In small-scale settings, when the number of edge servers is greater than two, EADDS trails SMT by only 2% in scheduling success rate and 3% in UE load. In contrast, SGDS consistently shows a significant performance gap. In large-scale scenarios, EADDS outperforms SGDS by 15%–20% in scheduling success rate and reduces UE load by 20%–25%.

Finally, Fig. 14 and Fig. 15 evaluate algorithm performance as the problem size scales up. As expected, UE computational load increases for both SGDS and EADDS as the number of tasks grows. This is because, in real-world scenarios, the number of edge servers cannot increase indefinitely, while the number of tasks can scale up significantly. Accordingly, in our experiments, edge server count is only increased linearly rather than proportionally to task count. This mismatch leads to insufficient offloading capacity, resulting in increased UE load and reduced scheduling success rate. Nonetheless, EADDS continues to deliver significantly better performance than SGDS under these conditions. In terms of runtime, although EADDS is slightly slower than SGDS, the difference remains small. For instance, when the task count is 5,000, both algorithms complete scheduling in a few seconds. Even at 500,000 tasks, both algorithms are able to complete within tens of seconds. Such runtime remains acceptable for the scenarios considered in this paper. Therefore, EADDS demonstrates strong scalability and can generate scheduling results within a reasonable time frame, even under ultra-large-scale workloads.

IX. Conclusion

This paper proposes an algorithm, EADDS, to address the task scheduling problem in the embedded-edge system continuum for real-time automotive applications. The algorithm effectively accounts for mobility while minimizing the computing load of the vehicle's embedded platform. Furthermore, experimental evaluations demonstrate EADDS's advantages over other algorithms and its robust performance across different problem scales, achieving a balance between solution quality and runtime.

ACKNOWLEDGMENT

This research has been funded by the Swedish Foundation for Strategic Research (SSF) under the project "Adaptive Software for the Heterogenous Edge-Cloud-Continuum," as well as by ELLIIT (Excellence Center at Linköping-Lund in Information Technology.

REFERENCES

[1] "In-Vehicle Computing for AI-Defined Cars", May 2021, [online] Available: https://www.nvidia.com/en-us/self-driving-cars/adas/.

[2] X. Huang, L. He, L. Wang and F. Li, "Towards 5G: Joint optimization of video segment caching transcoding and resource allocation for adaptive video streaming in a multi-access edge computing network", IEEE Trans. Veh. Technol., vol. 70, no. 10, pp. 10909-10924, Oct. 2021.

[3] D. Bhatta and L. Mashayekhy, "A bifactor approximation algorithm for cloudlet placement in edge computing", IEEE Trans. Parallel Distrib. Syst., vol. 33, no. 8, pp. 1787-1798, Aug. 2022.

[4] W. Ma and L. Mashayekhy, "Quality-aware video offloading in mobile edge computing: A data-driven two-stage stochastic optimization", Proc. IEEE 14th Int. Conf. Cloud Comput., pp. 594-599, 2021.

[5] De Moura, Leonardo, and Nikolaj Bjørner. "Z3: An efficient SMT solver." International conference on Tools and Algorithms for the Construction and Analysis of Systems. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[6] S. Jošilo and G. Dán, "Computation Offloading Scheduling for Periodic Tasks in Mobile Edge Computing," in IEEE/ACM Transactions on Networking, vol. 28, no. 2, pp. 667-680, April 2020

[7] X. Chen, L. Jiao, W. Li and X. Fu, "Efficient Multi-User Computation Offloading for Mobile-Edge Cloud Computing," in IEEE/ACM Transactions on Networking, vol. 24, no. 5, pp. 2795-2808, October 2016

[8] U. Saleem, Y. Liu, S. Jangsher, Y. Li and T. Jiang, "Mobility-Aware Joint Task Scheduling and Resource Allocation for Cooperative Mobile Edge Computing," in IEEE Transactions on Wireless Communications, vol. 20, no. 1, pp. 360-374, Jan. 2021

[9] Y. -H. Kao, B. Krishnamachari, M. -R. Ra and F. Bai, "Hermes: Latency Optimal Task Assignment for Resource-constrained Mobile Computing," in IEEE Transactions on Mobile Computing, vol. 16, no. 11, pp. 3056-3069, 1 Nov. 2017

[10] E. F. Maleki, W. Ma, L. Mashayekhy and H. J. La Roche, "QoS-Aware Content Delivery in 5G-Enabled Edge Computing: Learning-Based Approaches," in IEEE Transactions on Mobile Computing, vol. 23, no. 10, pp. 9324-9336, Oct. 2024

[11] Y. Liu et al., "Dependency-Aware Task Scheduling in Vehicular Edge Computing," in IEEE Internet of Things Journal, vol. 7, no. 6, pp. 4961-4971, June 2020

[12] L. Yang, H. Zhang, M. Li, J. Guo and H. Ji, "Mobile Edge Computing Empowered Energy Efficient Task Offloading in 5G," in IEEE Transactions on Vehicular Technology, vol. 67, no. 7, pp. 6398-6409, July 2018

[13] Ning Z, Dong P, Wang X, et al. Mobile edge computing enabled 5G health monitoring for Internet of medical things: A decentralized game theoretic approach[J]. IEEE Journal on Selected Areas in Communications, 2020, 39(2): 463-478.

[14] Aissioui, A. Ksentini, A. M. Gueroui and T. Taleb, "On Enabling 5G Automotive Systems Using Follow Me Edge-Cloud Concept," in IEEE Transactions on Vehicular Technology, vol. 67, no. 6, pp. 5302-5316, June 2018

[15] Yamamoto, Toshiaki, Toshihiko Komine, and Satoshi Konishi. "Mobility load balancing scheme based on cell reselection." Cell 7 (2012): 7UEs.

[16] Mandal, Gouranga, Diptendu Bhattacharya, and Parthasarathi De. "Real-time automotive night-vision system for drivers to inhibit headlight glare of the oncoming vehicles and enhance road visibility." Journal of Real-Time Image Processing 18.6 (2021): 2193-2209.

[17] 3GPP TS 38.306 version 18.1.0 Release 18, "https://www.etsi.org/deliver/etsi_ts/138300_138399/138306/18.01.00_60/ts_138306v180100p.pdf"