Title Scheduling in Serverless Computing for Solar Powered IoT Networks: A Two-Phase Approach

Student Name Junfei Zhan

Student No. 2020101900

Major Information and Computing Science

Supervisor Tengjiao He

Date 20/04/2024

暨 南 大 学

本科生毕业论文

论文题目 太阳能物联网网络无服务器计算中的调度:两阶段方法

学　　院 暨南大学伯明翰大学联合学院

学　　系

专　　业 信息与计算学

姓　　名 占俊飞

学　　号 2020101900

指导教师

2024 年 4 月 20 日

Statement of Originality

I hereby declare that the thesis presented is the result of research performed by me personally, under guidance from my supervisor. This thesis does not contain any content (other than those cited with references) that has been previously published or written by others, nor does it contain any material previously presented to other educational institutions for degree or certificate purpose to the best of my knowledge. I promise that all facts presented in this thesis are true and creditable.

Signed: 马俊飞 Date: 2024/04/20

诚 信 声 明

我声明，所呈交的毕业论文是本人在老师指导下进行的研究工作及取得的研究成果。据我查证，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得其他教育机构的学位或证书而使用过的材料。我承诺，论文中的所有内容均真实、可信。

毕业论文作者签名：占俊飞　　签名日期： 2024 年 4 月 20 日

Scheduling in Serverless Computing for Solar Powered IoT Networks: A Two-Phase Approach

Abstract: This article explores the integration of serverless and edge computing within solar-powered Internet of Things (IoT) networks. The challenge of optimizing computational task scheduling and resource allocation across IoT devices to enhance efficiency and reduce network strain is addressed. Central to this investigation is the development of a two-phase optimization solution that strategically manages the deployment of computational tasks, taking into account the devices' limited energy, computational power, and storage capacity.

The core contributions of this work include: (i) a novel serverless computing model tailored for energy-constrained IoT environments, ensuring operations within the thresholds of device capabilities while maximizing task efficiency; (ii) a mixed integer linear programming (MILP) formulation that accurately models the dynamics of sub-tasks allocation and resource scheduling, providing a benchmark for optimization; and (iii) a pragmatic two-phase method that approximates the optimal MILP solution, achieving a performance that reaches 81.47% of the MILP's benchmark in terms of application completion rates.

Experimental results highlight the efficacy of the proposed approach in minimizing energy consumption and optimizing task execution, showcasing its potential to significantly improve the operational dynamics of solar-powered IoT networks. The findings not only validate the practical viability of the approach but also illustrate its significant potential to enhance the sustainability, efficiency, and resilience of IoT ecosystems.

Key Words: edge computing, serverless computing, mixed integer linear programming, Two-Phase Approach

太阳能物联网网络无服务器计算中的调度：两阶段方法

摘要：本文探讨了无服务器和边缘计算在太阳能物联网(IoT)网络中的集成。解决了优化物联网设备的计算任务调度和资源分配以提高效率和减少网络压力的挑战。这项研究的核心是开发一种两阶段优化解决方案，该解决方案可以战略性地管理计算任务的部署，同时考虑到设备有限的能量、计算能力和存储容量。

这项工作的核心贡献包括 (i) 为能源受限的物联网环境量身定制的新型无服务器计算模型，可确保在设备能力阈值范围内运行，同时最大限度地提高任务效率；(ii) 混合整数线性规划（MILP）公式，可准确模拟子任务分配和资源调度的动态，为优化提供基准；以及 (iii) 一种实用的两阶段方法，该方法近似于最优 MILP 解决方案，在应用完成率方面达到 MILP 基准的 81.47%。

实验结果强调了该方法在最小化能耗和优化任务执行方面的有效性，展示了其显著改善太阳能物联网网络运行动态的潜力。研究结果不仅验证了该方法的实际可行性，还说明了其在提高物联网生态系统的可持续性、效率和弹性方面的巨大潜力。

关键字：边缘计算，无服务器计算，混合整数线性规划，两阶段

Contents

1. Introduction

The emergence of smart cities, smart transportation, and precision agriculture has enabled more Internet of Things (IoT) devices to join our lives and are increasingly powered by solar energy to minimize environmental impact. This growth in IoT deployments has accelerated the shift to edge computing, decentralizing data processing to increase efficiency and reduce network stress.

The advent of edge computing has revolutionized data processing by decentralizing the execution of services, accelerating response times, and reducing energy consumption and network congestion. This decentralization plays a crucial role in IoT ecosystems, facilitating rapid data analytics and processing close to user devices, thereby minimizing latency and easing network loads [1],[2]. The integration of edge computing into IoT is further supported by significant industrial investment and the development of standardized Multi-access Edge Computing (MEC) interfaces, highlighting its role in enhancing network services [3].

Alongside the growth of edge computing, serverless computing has emerged as a transformative paradigm, introducing a consumption-based billing model and simplifying server management for developers. This model primarily employed Function as a Service (FaaS) to break down applications into independent, transient, stateless sub-tasks across different nodes [4],[5]. Incorporating serverless architecture into edge ecosystems optimizes resource use and ensures scalability, leading to more dynamic and cost-effective service delivery to end-users [6],[7],[8].

However, merging edge and serverless computing introduces complexity, especially in managing task scheduling and resource allocation within a serverless edge environment [9]. Existing studies provide initial insights but rarely address the detailed scheduling challenges under third-party resource coordination and the limited capacity of edge servers [10],[11]. Our research investigates these scheduling challenges for stateless functions in an IoT edge computing setting, focusing on the detailed decision-making required for energy management, computational limitations, and the availability of functions.

To illustrate this, imagine a smart city scenario where solar-powered IoT devices are deployed on both sides of the road to monitor traffic conditions. When a traffic accident occurs, IoT devices around the accident immediately recognize a new application that needs to be executed - road warning. This road alert application can be broken down into multiple sub-tasks, including real-time traffic data analysis, signal timing of traffic lights and redirect traffic flow,

aiming to quickly respond to accidents and minimize their impact on traffic. Each sub-task is designed to be lightweight and efficient, ensuring rapid execution even on devices with limited computing and energy resources. This process demonstrates how serverless computing can break down a complex application into manageable sub-tasks, each of which needs to be executed by a specific function. This approach not only improves processing efficiency, but also ensures that each sub-task can be processed in a timely and efficient manner, even when resources are limited.

In particular, this example highlights the following key decision points: (i) select which functions to deploy to the IoT devices to perform corresponding sub-tasks based on limited device resources, (ii) schedule which sub-tasks to be processed locally at a specific time, and (iii) schedule which sub-tasks to be uploaded to the cloud server for execution at a specific time. In addition to these considerations, a major optimization goal has emerged: to maximize the number of applications completed by IoT devices in a given time period, ensuring efficient utilization of on-premises and cloud resources. With the flexibility of serverless computing, even resource-constrained IoT devices can effectively participate in the execution of complex applications. This approach allows the device to dynamically choose the best way to perform sub-tasks based on its current capabilities and the urgency of the task, either locally or with computing power in the cloud.

Nevertheless,this system is confronted with several challenges. First, there exists a trade-off between the quantity of functions deployed locally and the storage capacity of IoT devices. More functions deployed locally consume additional storage space, but enable quicker execution of application sub-tasks. Second, the decision-making process is both combinatorial and conditional. Specifically, a local device may either execute sub-tasks locally with corresponding functions deployed or choose to upload tasks to the cloud server. Furthermore, it is necessary to decide the specific time slots for handling and uploading sub-tasks. Assuming N devices, T time slots, and an average of M sub-tasks per application, the solution space is approximately calculated as 2^{TNM^2}, considering each sub-task can be either executed locally or uploaded. Third, devices operate under the uncertainty of energy availability and network conditions, lacking predictive information regarding future energy intake and channel gains to cloud servers[12].

This article explores the scheduling issues within serverless edge computing for IoT, outlining our key contributions as follows:

1. We modeled a new serverless computing system designed for solar-powered IoT net-

works, based on an innovative solar energy harvesting model. This system not only supports charging IoT devices with solar energy but also enables the execution of applications on these devices. Applications are divided into multiple sub-tasks, with each sub-task executed by corresponding stateless functions. Importantly, the execution of these sub-tasks is a collaborative effort between the IoT devices and the cloud server, ensuring adherence to the device's energy, computation, and storage limits.

2. We proposed an optimization problem for serverless computing in IoT environments. This problem aims to maximize the total number of applications successfully executed within a certain period. It considers a set of predetermined function configurations for IoT devices, the execution of sub-tasks locally at specific times, and the uploading of sub-tasks for execution on cloud servers as decision variables, forming a Mixed Integer Linear Programming (MILP) problem.

3. To solve this optimization problem, we designed a two-phase solution. In the first phase, a simplified MILP problem is solved to determine a scheme for the function configuration set of IoT devices. In the second phase, based on the results from the first phase, we use Receding Horizon Control (RHC) and Gaussian Mixture Model (GMM) strategies to decide on the allocation of sub-tasks and the scheme for local or cloud execution at specific times.

4. Our experiments show that this two-phase method significantly improves resource allocation efficiency and application demand satisfaction compared to traditional MILP and random allocation methods. Especially, it dynamically adjusts based on real-time solar energy and channel gain estimates, demonstrating significant performance benefits over random allocation methods as the IoT network scales.

The rest of this article is organized as follows: Section 2 presents a review of the literature, establishing the context and background for our work. Section 3 is dedicated to building the system model and formulating the scheduling problem as a Mixed Integer Linear Programming (MILP) problem. In Section 4, we introduce our two-phase scheduling algorithm, detailing its design and operational principles. Section 5 evaluates the performance of our proposed scheme through extensive simulations, demonstrating its efficacy and advantages. Finally, Section 6 concludes the article, summarizing our findings and suggesting directions for future research.

2. Related Works

Table 1: A COMPARISON OF RELATED WORKS.

Prior works	**Energy harvesting**	**Multiple time slots**	**Multiple devices**	**Function configuration**
[13],[14]	✓	×	×	✓
[9],[15]	×	✓	✓	✓
[16],[17]	×	×	✓	✓
[18],[19]	×	×	×	✓
[20],[21]	×	✓	×	✓
This work	✓	✓	✓	✓

Many works have investigated the application of serverless computing within IoT networks, focusing on various aspects such as energy harvesting, time and device management, and function configuration. For instance, Ko *et al.* [13] and Aslanpour *et al.* [14] have delved into energy harvesting and function configuration, revealing potential in optimizing latency and energy consumption in IoT networks. While their contributions are pivotal, they do not extend their frameworks to encompass the complexities of solar energy sources or the dynamics involving multiple devices and time slots.

In addressing the challenges of managing multiple time slots and devices, Xie *et al.* [9] and Deng *et al.* [15] propose multi-objective optimization solutions aimed at minimizing energy consumption, time, and cost. Their research, however, does not integrate energy harvesting techniques, which is a crucial element in promoting sustainability in IoT networks.

The studies by Das *et al.* [16] and Bensalem *et al.* [17] concentrate on optimizing the performance and function allocation for multiple devices without incorporating multiple time slots or energy harvesting, thus limiting the scalability and adaptability of their solutions in dynamic environments.

Furthermore, Cicconetti *et al.* [18] and Bac *et al.* [19] have explored function configuration strategies for serverless computing in edge and IoT networks. Their research, primarily focused on single-device scenarios, underscores the need for advanced function configuration but does not address the challenges posed by energy constraints or the management of multiple devices and time slots.

Vahidinia *et al.* [20] and Agarwal *et al.* [21] have made strides in minimizing latency issues and function cold start frequencies, with an emphasis on time management and efficient function configuration. However, their solutions overlook the integration of energy harvesting and the complexity of handling multiple devices.

These prior works, while foundational, have not fully addressed the unique challenges and opportunities presented by integrating advanced energy harvesting techniques and optimizing function configuration for scalability and efficiency in serverless edge computing frameworks. This work proposes a comprehensive framework that incorporates solar energy harvesting to power multiple devices and optimizes application execution over various time slots with advanced function configuration strategies.

3. System Model

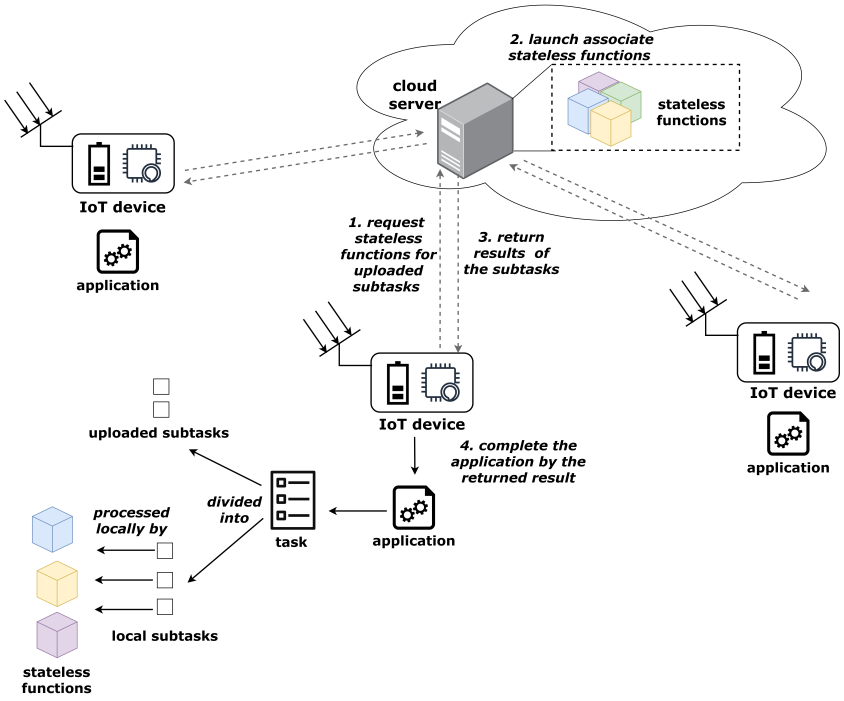

Figure 1: System Model

In this section, we introduce our serverless computing model designed for IoT networks. Our IoT system comprises a cloud server c and a set of solar-powered IoT devices, denoted by $\mathcal{V}$. Each IoT device $i \in \mathcal{V}$ is equipped with a solar panel and a rechargeable battery. Additionally, every device i hosts an application, which essentially is a task that needs to be computed and is composed of multiple sub-tasks. These sub-tasks are executed by corresponding stateless

functions. The cloud stores a library of functions along with the environment for all functions, represented by $\mathcal{F}$ as the set of functions. Given the limited energy, computation, and storage capacities of the devices, each device can only configure an environment for a subset of functions, denoted by $\mathcal{F}_i$. To complete an application, device i must request the cloud server s to execute the remaining sub-tasks of the application, since the device can only execute sub-tasks associated with functions within $\mathcal{F}_i$. We define $\mathcal{T}$ as the set of discrete time slots or the planning horizon. Each time slot is indexed by $t \in \mathcal{T}$ and has a duration of τ.

3.1 Energy Harvesting Model

IoT devices rely solely on solar energy, necessitating efficient management of energy arrivals and storage. Let E_i^t represent the solar energy arriving at device i in time slot t. The device's battery capacity, denoted as B_i, might not accommodate the entire E_i^t, leading to a situation where only a portion, ω_i^t, can be stored during time slot t. The energy available in device i at the beginning of time slot t is given by b_i^t. These variables are subject to the constraints:

$$\omega_i^t \leq E_i^t, \quad \forall i \in \mathcal{V}, \forall t \in \mathcal{T}, \tag{1}$$

$$b_i^t + \omega_i^t \leq B_i, \quad \forall i \in \mathcal{V}, \forall t \in \mathcal{T}. \tag{2}$$

To accurately model the variability of solar energy arrival, we apply a stochastic model. The model simulates E_i^t through a hidden Markov model with states indicating weather conditions: "Excellent", "Good", "Fair", and "Poor", denoted by $w \in \{w_E, w_G, w_F, w_P\}$. Each state has a Gaussian distribution of energy arrivals with mean μ_w and variance σ_w^2. State transitions are described by a probability matrix $\boldsymbol{P}$, where $\boldsymbol{P_{i,j}}$ shows the transition probability from state i to state j. Parameters for these distributions and transitions are based on historical solar irradiance data, as detailed in [22].

3.2 Application Model

In our IoT system, each device $i \in \mathcal{V}$ is dedicated to a single application. The set of sub-tasks for the application on device i is denoted as $\mathcal{S}_i$. For each sub-task $s \in \mathcal{S}_i$, its computational requirement is determined by its data size D_s (in bits) and its workload W_s (in CPU cycles per bit), leading to a product of $D_s W_s$ which denotes the required CPU cycles for execution.

Each device i has a pre-configured environment for a subset of functions $\mathcal{F}_i \subseteq \mathcal{F}$, where $\mathcal{F}$ is the set of all functions available in the cloud. A binary decision variable $f_{i,k}$ indicates whether

function k is pre-configured on device i, with $f_{i,k} = 1$ if yes, and 0 otherwise. sub-tasks require specific functions for execution. We express this through a binary parameter $\rho_{s,k} = 1$ if sub-task s requires function k, and 0 otherwise. A binary variable $x_{i,s}^t$ denotes whether sub-task s is executed on device i at time slot t. This leads to the following constraint:

$$x_{i,s}^t \leq \sum_{k \in \mathcal{F}_i} \rho_{s,k} f_{i,k}, \quad \forall s \in \mathcal{S}_i, \forall i \in \mathcal{V}, \forall t \in \mathcal{T}, \tag{3}$$

For sub-tasks that cannot be executed locally due to the absence of necessary functions, we introduce a binary variable $y_{i,s}^t$, where $y_{i,s}^t = 1$ if sub-task s is offloaded to the cloud at time t. We establish the relationship between $x_{i,s}^t$ and $y_{i,s}^t$ as:

$$x_{i,s}^t + y_{i,s}^t \leq 1, \quad \forall s \in \mathcal{S}_i, \forall i \in \mathcal{V}, \forall t \in \mathcal{T}, \tag{4}$$

ensuring that each sub-task is either executed locally or offloaded to the cloud, but not both.

The computational capability of device i is denoted by C_i, which limits the number of sub-tasks that can be executed locally at each time slot:

$$\sum_{s \in \mathcal{S}_i} D_s W_s x_{i,s}^t \leq C_i, \quad \forall i \in \mathcal{V}, \forall t \in \mathcal{T}. \tag{5}$$

Additionally, due to limited storage resources, each device can only configure a subset of the functions $\mathcal{F}_i$. The storage constraint for the stateless functions is given by:

$$\sum_{k \in \mathcal{F}} \sigma_k \cdot f_{i,k} \leq S_i, \quad \forall i \in \mathcal{V}. \tag{6}$$

where S_i denotes the storage capacity of device i, and σ_k represents the storage requirement of function k.

3.3 Communication Model

IoT devices utilize a wireless communication framework for data transmission to the cloud server, where efficiency is governed by channel power gain and transmission energy requirements. The channel power gain between an IoT device i and the cloud server c during a time slot t is given by:

$$g_{i,c}^t = h C_0 \left(\frac{D_{i,c}}{D_0} \right)^{-\alpha}, \quad \forall i \in \mathcal{V}, \forall t \in \mathcal{T} \tag{7}$$

where h represents the channel fading effect, modeled as an exponentially distributed random variable with an average value of one, C_0 is the path loss at a reference distance D_0, α is the path loss exponent, and $D_{i,c}$ is the Euclidean distance between device i and the cloud server c.

The energy required for an IoT device i to transmit its data to the cloud server in time slot t, denoted as $\lambda_{t,i}^{U}$, is calculated as:

$$\lambda_{t,i}^{U} = \sum_{s \in \mathcal{S}_a} \frac{D_s y_{i,s}^{t} \eta}{g_{i,c}^{t}}, \quad \forall i \in \mathcal{V}, \forall t \in \mathcal{T} \tag{8}$$

where D_s is the data size of sub-task s in bits, $y_{i,s}^{t}$ is a binary variable indicating whether the sub-task s is transmitted from device i to the cloud at time t, and η is a constant that represents the energy efficiency of the transmitter. Further, a device can only upload its sub-tasks in slot t when it has sufficient energy for that slot. The energy constraint for a device in slot t is given by:

$$\lambda_{t,i}^{U} \leq b_t^i, \quad \forall i \in \mathcal{V}, \forall t \in \mathcal{T}, \tag{9}$$

where $\lambda_{t,i}^{U}$ is the energy required for transmitting sub-tasks, and b_t^i is the available battery energy at the beginning of slot t.

In this section, the wireless communication framework required for IoT devices to transmit data to the cloud server is elaborated upon, taking into account channel power gain and the energy requirements for transmission. Notably, the model does not account for the communication process from the cloud server back to the IoT devices. This decision is based on the consideration that the data size of the results returned to the IoT devices, following the execution of sub-tasks, is typically minimal. Consequently, the impact of this returning data on the overall energy consumption and communication efficiency is negligible.

3.4 Device Energy Consumption Model

Device energy consumption involves two main operations: (i) sub-task upload and (ii) sub-task execution. The energy consumption for sub-task upload has been discussed in Subsection 3.3. For sub-task execution, we model the energy consumption using the data size D_s and workload W_s of the sub-task, along with the energy cost per CPU cycle denoted by ν.

The energy consumed for executing sub-tasks locally on device i is expressed as:

$$\lambda_{t,i}^{E} = \sum_{s \in \mathcal{S}_a} x_{i,s}^{t} D_s W_s \nu, \quad \forall i \in \mathcal{V}, \forall t \in \mathcal{T} \tag{10}$$

where $\lambda_{t,i}^{E}$ represents the energy required to execute local sub-task s on device i. This energy consumption is forced to zero when $x_{i,s}^{t} = 0, \forall s \in \mathcal{S}_a$. After defining the energy consumption of the two operations on device i, the total energy consumption for device i at time slot t is given

by:

$$\lambda_{t,i}^{T} = \lambda_{t,i}^{E} + \lambda_{t,i}^{U} \tag{11}$$

Furthermore, considering the total energy consumption for both local sub-task execution and sub-task uploading, the cumulative energy consumed for these operations must not exceed the energy harvested by device i over time planning horizon. Therefore, we impose the constraint:

$$\lambda_{t,i}^{T} \leq \sum_{t \in \mathcal{T}} \omega_{t}^{i}, \quad \forall i \in \mathcal{V}, \forall t \in \mathcal{T} \tag{12}$$

where ω_t^i is the energy stored during time slot t.

As a result, the energy level of device i evolves according to the equation:

$$b_{t}^{i} = b_{t-1}^{i} + \omega_{t-1}^{i} - \lambda_{t,i}^{T}, \quad \forall t \in \mathcal{T} \tag{13}$$

3.5 Problem Formulation

The primary performance metric of interest is the total number of applications completed within this system over $|\mathcal{T}|$ time slots. The decision-making process involves three main variables for each device i: (i) the function configuration, (ii) the decision to execute sub-tasks locally, and (iii) the decision to upload sub-tasks for cloud offloading. This metric, denoted as Z, quantifies the overall number of applications that are successfully completed throughout the planning horizon $\mathcal{T}$. Calculation of Z involves aggregating the completion status of all applications, factoring in the execution of all relevant sub-tasks $s \in \mathcal{S}_i$ through either local processing or cloud-based methods.

Within this system, z_i serves as a binary indicator variable for each device i, signifying whether the application on said device is fully executed ($z_i = 1$) or not ($z_i = 0$) over the period $\mathcal{T}$.To ensure comprehensive application completion, incorporating all its sub-tasks across every time slot, whether executed locally or offloaded to the cloud, the model introduces the following constraint:

$$\sum_{t \in \mathcal{T}} \sum_{s \in \mathcal{S}_i} (x_{i,s}^{t} + y_{i,s}^{t}) \geq |\mathcal{T}||\mathcal{S}_i| z_i, \quad \forall i \in \mathcal{V} \tag{14}$$

This constraint is pivotal for indirectly facilitating the maximization of Z, by setting a prerequisite for the completion status of applications based on the successful execution or offloading of all associated sub-tasks.

The problem can be modeled as the following MILP :

$$\begin{aligned} \underset{\mathbf{f},\,\mathbf{x},\,\mathbf{y}}{\text{maximize}} \quad & Z = \sum_{i \in \mathcal{V}} \sum_{t \in \mathcal{T}} \left(\prod_{s \in \mathcal{S}_i} (x_{i,s}^t + y_{i,s}^t) \right) = \sum_{i \in \mathcal{V}} z_i \\ \text{subject to} \quad & (1) - (6), (8) - (14) \end{aligned} \tag{15}$$

4. A Resource-Aware Two-Phase Optimization Solution

This section presents a resource-aware two-phase optimization solution addressing problem (15) in this IoT networks. At its core, the solution employs a straightforward strategy that classifies decision variables according to their time-dependence. These variables fall into two primary categories: static variables, which focus on the configuration of functions on devices, and dynamic variables, concerned with the execution of sub-tasks and their distribution between local processing and cloud offloading. Algorithm 1 shows the overall architecture of this two-phase optimization method.

In the first phase, the solution configures the functional capabilities of the IoT devices, ensuring that each is tailored to effectively handle the expected workload within its specific resource constraints. This configuration is static, meaning it remains unchanged over the planning horizon, enabling devices to operate within their energy, storage, and computational limits.

The second phase of the solution capitalizes on the established configurations, intelligently directing the execution of sub-tasks based on real-time evaluations of solar energy availability and channel gain fluctuations. This phase is dynamic, adeptly adjusting to the current state of network resources to optimize sub-task distribution. The objective is to enhance the overall system's productivity by maximizing the number of applications completed, all while adapting to the ever-changing energy and network conditions.

4.1 Phase One: Function Configuration Decision

In the first phase of the solution, a simplified MILP model is deployed to determine the optimal configuration of functions across Internet of Things (IoT) devices. The objective of this phase is to select a subset of functions, $\mathcal{F}_i$, for each device i from the universal set of functions $\mathcal{F}$, aiming to strike a balance between the expected workload of each device and its resource constraints, including storage, computational, and energy capacities. The objective of the MILP model is to maximize the coverage of sub-tasks by the configured functions on devices, aiming for the most efficient use of the devices' storage, computational and energy resources.

Algorithm 1 Two-Phase Optimization for IoT Serverless Computing

procedure Two_Phase_Optimization($\mathcal{V}, \mathcal{F}, \mathcal{T}$)
    Phase One: Function Configuration
    Estimate the average energy arrival $\overline{E_i}$ for each device i
    Determine optimal function configuration $\mathcal{F}_i$ for each device i
    Phase Two: sub-task Allocation
    for each time slot t in $\mathcal{T}$ **do**
        Estimate energy arrival $\hat{E}_i^t$ and channel gain $\hat{g}_i^t$ for each device i in time slot t
        Maximize number of completed applications based on $\mathcal{F}_i$
        Update estimation model and energy level b_i^t for each device i
    end for
end procedure

The simplified MILP formulation is as follows:

Objective:

$$\max_{f_{i,k}} \sum_{i \in \mathcal{V}} \sum_{k \in \mathcal{F}} \sum_{s \in \mathcal{S}} \rho_{s,k} f_{i,k} \tag{16}$$

Subject to:

Storage-Aware Constraint:

$$\sum_{k \in \mathcal{F}} \sigma_k f_{i,k} \leq S_i, \quad \forall i \in \mathcal{V} \tag{17}$$

Computation-Aware Constraint:

$$\sum_{k \in \mathcal{F}} \sum_{s \in \mathcal{S}_i} D_s W_s \rho_{s,k} f_{i,k} \leq C_i, \quad \forall i \in \mathcal{V} \tag{18}$$

Energy-Aware Constraint:

$$\sum_{k \in \mathcal{F}} \sum_{s \in \mathcal{S}_i} D_s W_s \nu \rho_{s,k} f_{i,k} \leq \overline{E_i}, \quad \forall i \in \mathcal{V} \tag{19}$$

where:

- $f_{i,k}$ is a binary variable indicating whether function k is configured on device i.
- $\rho_{s,k}$ indicates whether sub-task s requires function k.
- ν denotes the energy cost per CPU cycle

- σ_k represents the storage requirement for function k.
- S_i, C_i, and $\overline{E_i}$ represent the storage, computational, and average energy capacities of device i, respectively.

This formulation ensures that the selected function configurations for each device are within their resource limits, optimizing the network's overall capability to handle its tasks efficiently.

A distinction in this model from the constraints on $f_{i,k}$ in the MILP model, noted in problem (15), is the inclusion of an energy-aware constraint. This constraint accounts for the energy requirements of locally executing sub-tasks with the designated function on each device. It specifically ensures that the energy needed for the function—computed from the energy consumption rate ν per CPU cycle, the data size of the sub-task D_s, and the workload W_s—does not exceed the estimated average energy $\overline{E_i}$ of device i. This method emphasizes the significance of energy efficiency in the sub-task allocation process.

Furthermore, regarding the estimated average energy $\overline{E_i}$ of device i, we utilize a Gaussian Mixture Model (GMM) to estimate the solar energy arrival across all $|\mathcal{T}|$ slots. Then, we calculate the average value of these slots for each device. This approach allows for a more accurate and refined estimation of energy availability, enhancing the model's capability to allocate sub-tasks efficiently.

4.2 Phase Two: sub-task Allocation Decision

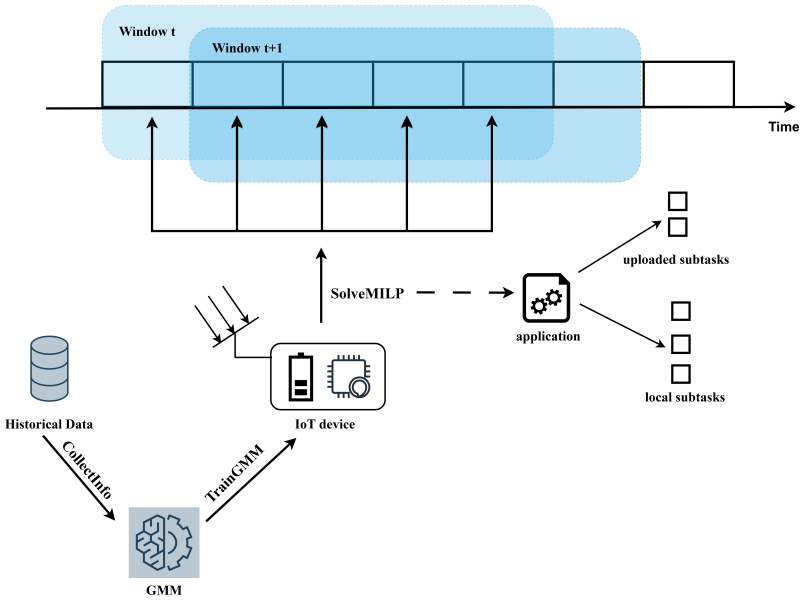

Figure 2: An Overview of Proposed Solution

With the configurations of functions established in Phase One, Phase Two advances the decision-making process for the execution of sub-tasks by leveraging the Receding Horizon Control (RHC) strategy. This method utilizes up-to-date forecasts of solar energy availability and channel gain, enhancing the effectiveness of decisions. Illustrated in Figure 2, a device applies the RHC method to identify the optimal course of action across a decision window of $K = 5$ time slots. Initially, the device executes the action determined for the current time slot t, and then shifts the decision window forward by one slot. Following this adjustment, the system revisits problem (15) to ascertain the optimal action for the next time slot $t + 1$, now incorporating refreshed estimates of channel conditions and solar energy arrivals. This process of continuous updating and reassessment ensures that the decision on whether to process sub-tasks locally or offload them to the cloud is made with the aim of optimizing resource allocation and task execution efficiency.

Our solution involves three stages: (i) Training, (ii) Prediction, and (iii) Decision.Let T be a large positive number. In the Training stage, each device collects historical data on solar energy arrival and channel gain over T slots. These two types of collected data are recorded in sets G_i^T for channel gain and E_i^T for energy harvesting, respectively. This data is then used to train Gaussian Mixture Models (GMMs) for predicting future solar energy arrivals and channel gains. For each IoT device, the historical data collected over T slots are used as input to the GMMs. The training of GMMs is performed via the Expectation-Maximization (EM) algorithm, which iteratively maximizes the likelihood function of the data. During the Expectation step, the algorithm estimates the probabilities that the data points were generated by each component of the mixture model [23]. In the Maximization step, the model parameters are updated to maximize the expected likelihood, refining the model's fit to the data. This iterative process continues until convergence, resulting in a model that can accurately predict future states based on past observations. The Prediction stage involves estimating the future state of these parameters using the trained GMMs, while the Decision stage involves solving the MILP problem of problem (15) without decisions of f_i^t to determine the optimal allocation of sub-tasks.

The solar energy arrival and channel gain are estimated as follows:

$$\hat{E}_i^t = \text{Predict}(\mathcal{E}_i^T), \quad \forall i \in \mathcal{V} \tag{20}$$

$$\hat{g}_i^t = \text{Predict}(\mathcal{G}_i^T), \quad \forall i \in \mathcal{V} \tag{21}$$

In the Algorithm 2,the Training Stage (Lines 4-7) begins with each IoT device i using `CollectInfo` to collect historical solar energy data G_i^T and channel gain data E_i^T. This data is then used to train two GMMs, `GMM_solar` and `GMM_channel`, with the `TrainGMM` function. The Prediction Stage (Lines 9-12) involves estimating the solar energy arrival $\hat{E}_i^t$ and channel gain $\hat{g}_i^t$ for the next time slot t for every device i using the trained GMMs. The Decision Stage (Lines 13-18) sets up a decision window $\mathcal{K}$ and employs `SolveMILP` to determine the efficient allocation of sub-tasks within that window. Once the sub-task decisions from Φ for the current time slot t are executed, the energy levels of the devices are updated for the next time slot.

Algorithm 2 sub-task Allocation Using RHC and GMM

1: **procedure** sub-taskAllocation($\mathcal{V}, \mathcal{T}, K$)
2: Initialize: Configure functions $\mathcal{F}_i$ for each device i.
3: **— Training Stage —**
4: **for** $i \in \mathcal{V}$ **do**
5: $\mathcal{G}_i^T, \mathcal{E}_i^T \leftarrow$ CollectInfo(i, T)
6: $GMM_{solar}, GMM_{channel} \leftarrow$ TrainGMM$(\mathcal{G}_i^T, \mathcal{E}_i^T)$
7: **end for**
8: **— Prediction Stage —**
9: **for** $t \in \mathcal{T}$ **do**
10: **for** $i \in \mathcal{V}$ **do**
11: $\hat{E}_i^t, \hat{g}_i^t \leftarrow$ Predict$(\mathcal{G}_i^T, \mathcal{E}_i^T, t, K)$
12: **end for**
13: **— Decision Stage —**
14: $\mathcal{K} \leftarrow \{t, t+1, \ldots, t+K-1\}$
15: $\Phi \leftarrow$ SolveMILP$(\mathcal{K}, \{\hat{g}_i^t\}, \{\hat{E}_i^t\})$
16: Execute decisions from Φ for time slot t
17: Update the device energy levels $b_i^{t+1} \leftarrow b_i^t -$ EnergyUsed(Φ)
18: **end for**
19: **end procedure**

5. Evaluation

We conducted our evaluation in a Python 3.9 environment, employing Gurobi 9.1.1 to solve the MILP problem delineated in (15). The simulation encompasses a network of IoT devices within a controlled experimental framework. The parameters defining our simulation are:

- Number of time slots (`Num_T`): 10
- Number of IoT devices (`Num_V`): 10
- Number of available functions (`Num_F`): 5
- Number of sub-tasks (`Num_Sub`): 5
- Battery capacity for each device: 100 Joules (`V_Energy`) [24]
- Solar panels per device: 30×30 cm^2 at 20% efficiency (`V_Solar_Panel`) [12]
- Computational capacity per device: 1.5e7 CPU cycles per time slot (`C_i`) [25]
- Storage capacity per device: 700 bits (`S_i`)
- Data size of sub-tasks: Randomly between 800 to 1500 bits (`D_s`)
- Workload of sub-tasks: Randomly between 1500 to 3000 CPU cycles per bit (`W_s`) [9]
- Storage needs for functions: Randomly between 100 to 200 bits (`sigma_k`)
- sub-task function requirements: Binary, randomly determined (`rho_s_k`)
- Transmitter energy efficiency: 0.3 (`eta`)
- Energy per CPU cycle: 2e-6 Joules (`nu`)
- Storage and battery capacities are constants for simplicity.

Channel gain and solar energy parameters are estimated according to the setup, with devices calibrated to optimize energy and computational efficiency within their operational constraints.

This setup evaluates the performance of our ”Two-phase” optimization method relative to a RANDOM approach, which makes sub-task allocations and function configurations decisions arbitrarily. Additionally, the MILP method serves as the benchmark, representing the optimal solution computed by Gurobi 9.1.1. Results presented are the mean of 10 iterations.

5.1 Impact of Number of Devices $|\mathcal{V}|$

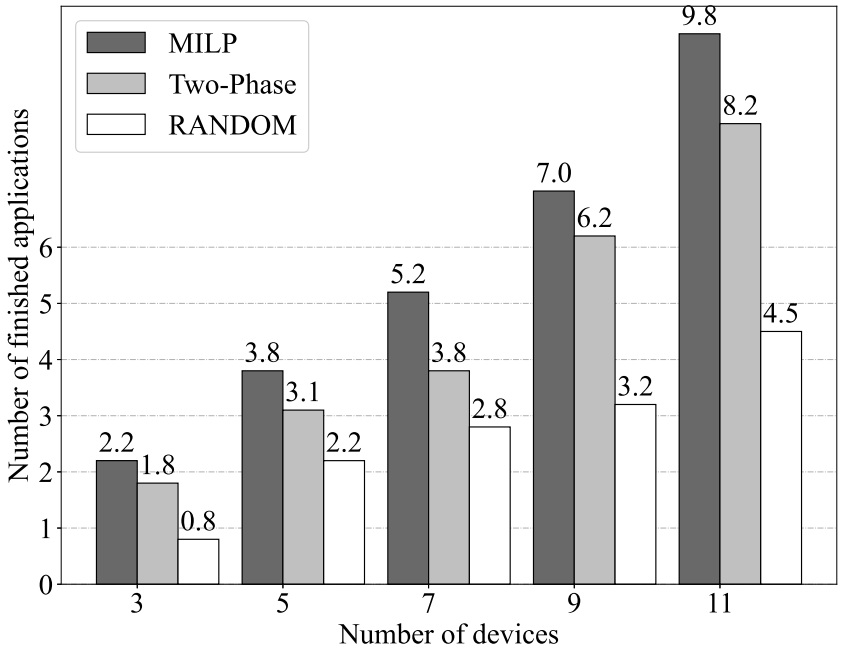

Figure 3: Impact of Number of Devices $|\mathcal{V}|$ on performance

Figure 3 delineates the performance impact based on the number of devices in the system. It is observed that the MILP method consistently exhibits superior performance across all device quantities. The Two-Phase approach follows closely, demonstrating robustness, particularly as the device count increases. In contrast, the RANDOM method shows a noticeable decline in performance with more devices. Notably, with a growing number of devices, the gap in performance between the RANDOM method and the other two approaches becomes increasingly pronounced, underscoring the effectiveness of systematic optimization over random allocation.

The MILP method showcases a consistent increase, with the number of finished applications rising from 2.2 to 9.8 as the number of devices escalates from 3 to 11. This pattern suggests an almost linear scalability, reflective of the method's ability to utilize additional resources effectively. Conversely, the Two-Phase method displays a moderate upswing, with finished applications ascending from 1.8 to 8.2, mirroring a strong but suboptimal scaling compared to MILP. Notably, the gap between MILP and Two-Phase narrows as the device count increases, indicating that the Two-Phase method's performance enhancement has a proportional relationship with system size. On the other hand, the RANDOM method exhibits the most limited progression, with the finished applications count incrementing only marginally from 0.8 to 4.5. This suggests

that while the RANDOM method benefits from additional devices, its lack of strategic planning leads to sublinear and inefficient scaling.

In comparison, the Two-Phase method commences with 81.8% of MILP's completed applications at three devices, but it experiences a relative improvement as the network size grows. Notably, with 11 devices, the Two-Phase method achieves 83.7% of the MILP's performance, hinting at its potential to scale robustly within larger networks, despite a lower starting point. This behavior underscores the Two-Phase method's ability to adapt and improve its relative efficiency as the system expands.

On the contrary, the RANDOM method displays a markedly different trend. Starting at only 36.4% of the MILP's performance with three devices, it shows a marginal increase in absolute terms but a decreased performance proportionally, with 45.9% at 11 devices. The RANDOM method's trajectory, marked by a less pronounced upward curve, suggests that it fails to capitalize on the increased network resources efficiently. This inefficiency is evident as its relative gain in performance, compared to MILP, does not correspond proportionally to the increase in the number of devices.

5.2 Impact of Number of subtaks $|\mathcal{S}_i|$

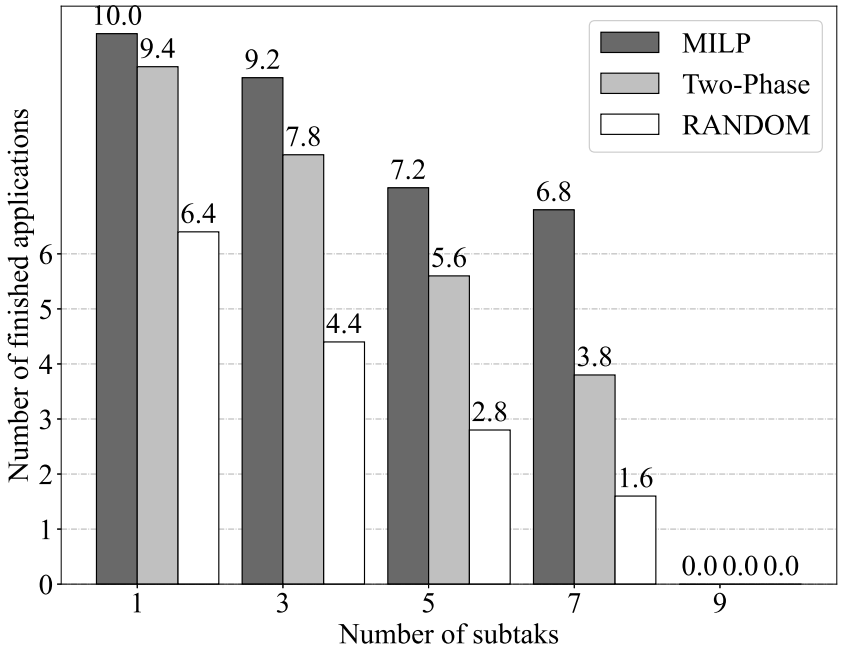

Figure 4: Impact of Number of sub-taks $|\mathcal{S}_i|$ on performance

Figure 4 presents a performance evaluation with varying numbers of sub-tasks $|\mathcal{S}_i|$. It reveals distinctive trends for each of the optimization methods as the number of sub-tasks increases. The MILP method commences with a high performance at 9.4 for a single sub-task, slightly diminishing to 9.2 with three sub-tasks, and then experiencing a more pronounced decrease as the sub-task number rises, reaching a low of 0 at nine sub-tasks. This suggests that while MILP handles smaller sub-task loads with near-perfect efficiency, its performance is inversely proportional to the number of sub-tasks.

The Two-Phase method starts with an excellent performance of 9.4 for a single sub-task, indicating its strength in scenarios with fewer sub-tasks. However, as the sub-task count grows, a decrement in performance is observed, dropping to 7.8 for three sub-tasks and progressively lowering to 0 for nine sub-tasks. This performance trajectory illustrates a resilience to increasing workloads, albeit with a decline in efficacy as the sub-task number becomes substantial.The RANDOM method demonstrates the least effective performance management across the board. It begins with a score of 6.4 and diminishes consistently with each increase in sub-tasks, concluding at 0 for nine sub-tasks. The absence of a strategic approach is evident as the RANDOM method shows the steepest decline in performance, indicating its vulnerability to increased complexity.

The comparison of completion rates presents the Two-Phase method at 94% of MILP's completion rate for a single sub-task, suggesting close competitiveness. Nevertheless, this percentage decreases as the number of sub-tasks grows, with the Two-Phase achieving 84.5%, 77.8%, and 55.9% of MILP's completions for three, five, and seven sub-tasks, respectively. This pattern indicates the Two-Phase method's performance is proportionately affected by the rising sub-task number, reflecting a decrement in efficiency at higher loads compared to MILP's more gradual decline. On the other hand, RANDOM's performance proportions significantly fall from 64% at one sub-task to 25% at seven sub-tasks, which is indicative of its comparative inefficiency in resource allocation and task management under increasing demands.

5.3 Impact of Storage Capacity of Device S_i

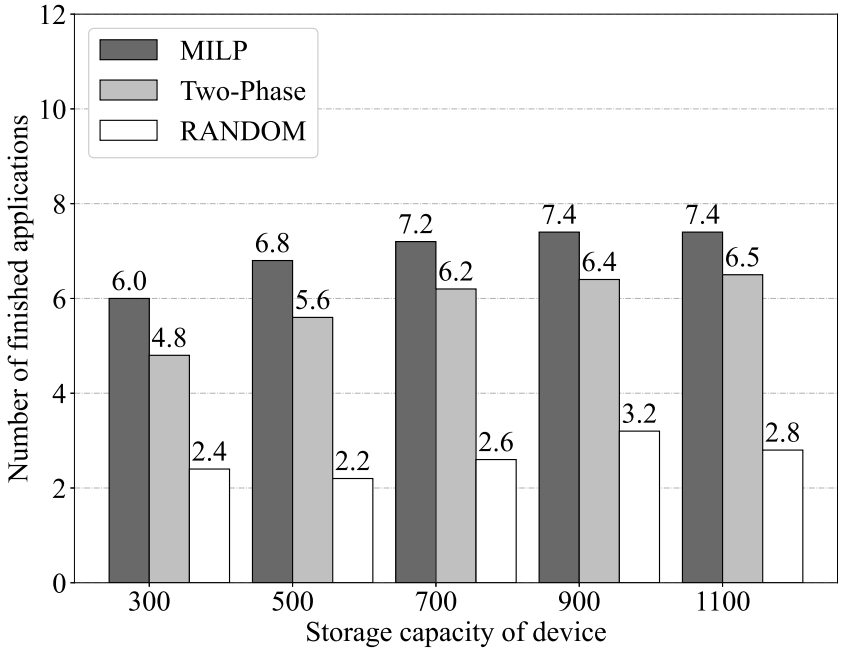

Figure 5: Impact of Storage Capacity of Device S_i on performance

Figure 5 illustrates the correlation between the storage capacity of IoT devices and their performance efficiency. It is evident from the graph that as the storage capacity increases, the number of completed applications by the MILP and Two-Phase methods shows a general upward trend, indicating that larger storage facilitates enhanced performance. The MILP method, in particular, maintains a consistently high number of completed applications across varying storage capacities, suggesting its robustness and efficient utilization of storage resources. In comparison, the Two-Phase method exhibits a gradual increase in performance as storage capacity grows, which implies that while the Two-Phase method benefits from increased storage, it does so with diminishing returns as capacity reaches higher thresholds.

Upon delving into the specifics, the MILP method begins at a completion count of 6 applications for a storage capacity of 300 bits, eventually plateauing at 7.4 applications for capacities of 900 bits and above. This indicates an initial sensitivity to storage which stabilizes at higher capacities. The Two-Phase method also starts at a lower completion count of 4.8 but sees a steadier climb, reaching 6.5 applications at the highest capacity. This suggests a consistent but moderate improvement aligned with storage enhancements. Conversely, the RANDOM method's trajectory is less predictable, with the number of completed applications not following a consistent

pattern as storage capacity increases. Its performance peaks at a capacity of 900 bits with 3.2 completed applications, only to dip slightly at a capacity of 1100 bits.

From the data presented in Figure 5, we observe that the Two-Phase method achieves a commendable proportion of the MILP's application completion count across various storage capacities. Specifically, at a storage capacity of 300 bits, the Two-Phase method accomplishes 80% of the applications completed by the MILP method. This ratio sees a slight decrease as the storage capacity increases; with the Two-Phase method reaching 88.5% of the MILP's performance at 500 bits, and gradually narrowing down to 87.8% at a storage capacity of 700 bits. Notably, at the higher capacities of 900 and 1100 bits, the Two-Phase method maintains a consistent completion ratio of 86.5% and 87.8%, respectively, compared to the MILP method.

These ratios reflect the Two-Phase method's relative efficiency compared to the optimized MILP approach, underscoring its potential as a near-optimal solution. Conversely, when considering the RANDOM method, the completion count reflects more significant fluctuations. At the lowest capacity of 300 bits, RANDOM achieves 40% of the MILP's completions, while at 900 bits, this performance peaks at 43.2%, demonstrating an inconsistent scaling with storage capacity increases. The RANDOM method, thus, does not exhibit a proportional increase in application completions with enhanced storage, which may be indicative of its inherent inefficiency in resource utilization, particularly at higher storage capacities.

6. Conclusion

This paper has presented a serverless computing model for solar-powered IoT networks that incorporates a novel approach to managing tasks within the constraints of energy, computation, and storage capacities. We introduced a Two-Phase optimization solution, set against a conventional MILP approach and a RANDOM method, to demonstrate the benefits of strategic resource management. Our model, underpinned by a set of IoT devices each equipped with a solar panel and battery, hosts applications that consist of multiple sub-tasks, necessitating efficient execution within the device's resource constraints.

The MILP approach, representing an optimal solution, and the Two-Phase method, an approximation of the MILP, were both shown to significantly outperform the RANDOM method. As the number of devices increased, the Two-Phase method displayed a proportional relationship in performance relative to MILP, reinforcing its scalability and robustness in larger IoT networks. Averagely, the Two-Phase method attained up to 87.8% of MILP's performance, a

testament to its near-optimal efficiency and adaptability.

However, the study revealed that the performance of both MILP and Two-Phase approaches deteriorated with an increased number of sub-tasks, eventually declining to zero for nine sub-tasks. This highlights a critical scalability issue as the complexity of task management rises, suggesting a future research direction towards enhancing the Two-Phase method to better manage larger sets of sub-tasks. Moreover, the results indicated a clear dependency on storage capacity, where larger capacities yielded improved performance for MILP and Two-Phase methods, whereas the RANDOM method showed no such correlation, peaking and then diminishing slightly, suggesting inefficient resource utilization.

For future endeavors, our objective is to leverage Directed Acyclic Graphs (DAGs) to model the interdependencies of sub-tasks within the serverless computing framework for IoT networks. By mapping the intricate precedence relations of sub-tasks via DAGs, we anticipate the creation of more sophisticated and scalable optimization strategies. This approach aims to enhance the orchestration of serverless functions, paving the way for more efficient task execution workflows. Embracing DAGs for task modeling is expected to not only improve the granularity of control over the execution order but also to reduce execution latency, thereby advancing the overall efficacy of IoT systems in distributed computing environments.

Acknowledgements

Undergraduate four years along the way, special thanks to all the teachers and classmates encountered in this journey, it is you who shaped my journey unforgettable and glittering, I hope that in the future, I can uphold the kindness and bravery, and continue to go on. In addition, I would like to thank my parents and sister, who have provided me with the most solid protection and support for my life and studies. My family is always the softest and toughest place in my heart. I would also like to express my special thanks to my undergraduate advisor, Prof. Tengjiao He, who not only opened the door of research for me, but also patiently guided me on how to write academic papers, implement algorithms, and how to effectively read the literature. All in all, there is no such thing as an unbroken banquet under the sky, so I hope you all will get better and better, have fun, eat, drink and sleep well in the coming days.

Appendix

This appendix includes the source code developed for the MILP and Two-Phase solutions as part of the above experiments.

MILP Solution Code

The first code snippet represents the Mixed-Integer Linear Programming (MILP) solution approach used for optimizing the given problem. This approach utilizes the Pyomo library for defining and solving the optimization model.

```
from pyomo.environ import *
import numpy as np

def call_MILP(V,F,Sub,T,B_i,C_i,S_i,D_s,W_s,sigma_k,rho_s_k,E_i,Dstore_Cap,
    Ebatt_Cap,eta,g_t,nu):

    # Initialize the model
    model = ConcreteModel(name="serverless")
    model.w_i = Var(T, V, bounds=(0, Dstore_Cap))  # $w_{i}^{t}$
    model.b_i = Var(T, V, bounds=(0, Ebatt_Cap))  # $b_{i}^{t}$
    # Decision variables
    model.f = Var(V, F, domain=Binary)  # Function configuration
    model.x = Var(T, V, Sub, domain=Binary)  # Subtask execution
    model.y = Var(T, V, Sub, domain=Binary)  # Subtask offloading

    # Constraint (1)
    def Energy_arrival_device_rule(model, t, i):
        return model.w_i[t, i] - E_i[t, i] <= 0

    model.Energy_arrival_device = Constraint(T, V, rule=
    Energy_arrival_device_rule)

    # Constraint (2)
    def Battery_device_rule(model, t, i):
        return model.b_i[t, i] + model.w_i[t, i] - B_i[i] <= 0

    model.Battery_device = Constraint(T, V, rule=Battery_device_rule)

```

```
    # Constraint (3)
    def application_constraint_rule(model, t, i, s):
        return model.x[t, i, s] <= sum(rho_s_k[s, k] * model.f[i, k] for k
    in F)

    model.ApplicationConstraint = Constraint(T, V, Sub, rule=
    application_constraint_rule)

    # Constraint (4)
    def subtask_offloading_rule(model, t, i, s):
        # Ensures a subtask is either executed locally or offloaded, but
    not both
        return model.x[t, i, s] + model.y[t, i, s] <= 1

    model.SubtaskOffloading = Constraint(T, V, Sub, rule=
    subtask_offloading_rule)

    # Constraint (5)
    def computational_capacity_rule(model, t, i):
        # Sum of CPU cycles required by all subtasks must not exceed the
    device's computational capacity
        return sum(D_s[s] * W_s[s] * model.x[t, i, s] for s in Sub) <= C_i[
    i]

    model.ComputationalCapacity = Constraint(T, V, rule=
    computational_capacity_rule)

    # Constraint (6)
    def storage_capacity_rule(model, i):
        # Sum of storage requirements of all functions configured on device
     must not exceed its storage capacity
        return sum(sigma_k[k] * model.f[i, k] for k in F) <= S_i[i]

    model.StorageCapacity = Constraint(V, rule=storage_capacity_rule)

    # Constraint (9)
    def energy_consumption_execution_rule(model, t, i):
        # Energy consumed for executing subtasks locally on device i at
    time t
```

```
57      return sum(model.y[t, i, s] * D_s[s] * W_s[s] * eta/g_t[t,i] for s
    in Sub) <= model.b_i[t, i]
58
59  model.EnergyConsumptionExecution = Constraint(T, V, rule=
    energy_consumption_execution_rule)
60
61  # Constraint (12)
62  def energy_consumption_total_rule(model,t, i):
63      # Energy consumed for executing subtasks locally on device i at
    time t
64      return sum(model.y[t, i, s] * D_s[s] * W_s[s] * eta/g_t[t,i] for s
    in Sub) + sum(model.x[t, i, s] * D_s[s] * W_s[s] * nu for s in Sub) <=
    sum(model.w_i[t,i] for t in T )
65
66  model.EnergyConsumptiontotal = Constraint(T, V, rule=
    energy_consumption_total_rule)
67
68  # Constraint (13)
69  def battery_state_update_rule(model, t, i):
70      # This constraint updates the battery state based on previous state
    , energy harvested, and energy consumed
71      # Assuming energy consumed for transmission (\lambda^{U}_{t,i}) is
    calculated or defined elsewhere
72      if t == 0:  # Initial condition
73          return Constraint.Skip  # Or define an initial state for model.
    b_i[0, i]
74      else:
75          return model.b_i[t, i] == model.b_i[t - 1, i] + model.w_i[t -
    1, i] - (sum(model.y[t, i, s] * D_s[s] * W_s[s] * eta/g_t[t,i] for s in
    Sub) + sum(model.x[t, i, s] * D_s[s] * W_s[s] * nu for s in Sub))
76
77  model.BatteryStateUpdate = Constraint(T, V, rule=
    battery_state_update_rule)
78
79  #Object
80  # Decision variables for application completion
81  model.z = Var(V, domain=Binary)  # Application completion indicator for
     each device
82
```

```
    # Constraint (14) to ensure subtask completion leads to application completion
    def application_completion_rule(model, i):
        # For each device i, ensure that if all subtasks are either executed or offloaded, then z[i] can be 1
        return sum(model.x[t, i, s] + model.y[t, i, s] for t in T for s in Sub) >= len(T) * len(Sub) * model.z[i]

    model.ApplicationCompletion = Constraint(V, rule=application_completion_rule)

    # Objective: Maximize the number of completed applications
    def objective_rule(model):
        return sum(model.z[i] for i in V)

    model.objective = Objective(rule=objective_rule, sense=maximize)

    #  Solve the MILP
    opt = SolverFactory('gurobi', solver_io='python')
    results = opt.solve(model)  # solves and updates instance
    Max_min_Opt = value(model.objective)
    model.f.pprint()
    return Max_min_Opt
```

Two-Phase Solution Code

The second code snippet outlines the Two-Phase solution method. The first phase deals with the function configuration, while the second phase focuses on decision-making based on the function configurations derived from the first phase.

```
#Phase 1
def call_function_config_MILP(V, F, S_i, C_i, sigma_k, rho_s_k, D_s, W_s, Sub):
    # Initialize the model
    model = ConcreteModel(name="FunctionConfiguration")

    # Decision variables
    model.f = Var(V, F, domain=Binary)  # Function configuration

```

```
    # Constraints
    # Constraint (1): Storage capacity constraint for each device
    def storage_capacity_rule(model, i):
        return sum(sigma_k[k] * model.f[i, k] for k in F) <= S_i[i]
    model.StorageCapacity = Constraint(V, rule=storage_capacity_rule)

    # Constraint (2) - Consider computational capacity if necessary
    # the computational demand (C_demand) of having a function on a device.
    def computational_capacity_rule(model, i):
        C_demand = {k: W_s[k] * D_s[k] for k in F}  # Example way to calculate demand, adjust as needed
        return sum(C_demand[k] * model.f[i, k] for k in F) <= C_i[i]
    model.ComputationalCapacity = Constraint(V, rule=computational_capacity_rule)

    # Objective: Maximize the coverage of subtasks by the configured functions on devices
    def objective_rule(model):
        return sum(sum(rho_s_k.get((s, k), 0) * model.f[i, k] for k in F) for s in Sub for i in V)
    model.objective = Objective(rule=objective_rule, sense=maximize)

    # Solve the MILP
    solver = SolverFactory('gurobi')
    results = solver.solve(model, tee=True)

    # Extract and return the function configuration solution
    f_sol = np.zeros((len(V),len(F)))
    for i in range(len(V)):
        for j in range(len(F)):
            f_sol[i,j] = model.f[i,j].value
    # f_solution = {(i, k): value(model.f[i, k]) for i in V for k in F}
    return f_sol


#Phase 2
def MPC_call_MILP(ban_i,energy_level,energy_store,x_dec,y_dec,current_slot,f_known,V,F,Sub,T,B_i,C_i,S_i,D_s,W_s,sigma_k,rho_s_k,E_i,Dstore_Cap,Ebatt_Cap,eta,g_t,nu):
```

```
42
43      # Initialize the model
44      model = ConcreteModel(name="serverless")
45      model.w_i = Var(T, V, bounds=(0, Dstore_Cap))  # $w_{i}^{t}$
46      model.b_i = Var(T, V, bounds=(0, Ebatt_Cap))  # $b_{i}^{t}$
47      # Decision variables
48      model.x = Var(T, V, Sub, domain=Binary)  # Subtask execution
49      model.y = Var(T, V, Sub, domain=Binary)  # Subtask offloading
50      # Fixed function configuration from Phase 1
51      f_init = {(v, f): f_known[v, f] for v in range(len(V)) for f in range(
    len(F))}
52      model.f = Param(V, F, initialize=f_init, within=Binary)
53      if current_slot>0:
54          # update the record information
55          for v in range(len(V)):
56              model.b_i[0, v].fix(energy_level[v])  # Fixed battery status
57              model.w_i[0, v].fix(energy_store[v])  # Fixed energy store
58              for s in range(len(Sub)):
59                  model.x[0, v, s].fix(x_dec[v, s])  # Fixed subtask
    execution
60                  model.y[0, v, s].fix(y_dec[v, s])  # Fixed subtask
    offloading
61      # Ban the device that finish its device
62      for i in range(len(V)):
63          if ban_i[i] == 1:
64          # Fix all decision variables related to the banned device to zero
65              for t in range(len(T)):
66                  for s in range(len(Sub)):
67                      model.x[t, i, s].fix(0)  # Do not execute any tasks
68                      model.y[t, i, s].fix(0)  # Do not offload any tasks
69
70      def ban_x_rule(model, t, i, s):
71          if ban_i[i] == 1:
72              return model.x[t, i, s] == 0
73          else:
74              return Constraint.Skip
75
76      model.ban_x_con = Constraint(T, V, Sub, rule=ban_x_rule)
77
```

```
    def ban_y_rule(model, t, i, s):
        if ban_i[i] == 1:
            return model.y[t, i, s] == 0
        else:
            return Constraint.Skip

    model.ban_y_con = Constraint(T, V, Sub, rule=ban_y_rule)

    # Constraint (1)
    def Energy_arrival_device_rule(model, t, i):
        return model.w_i[t, i] - E_i[t, i] <= 0

    model.Energy_arrival_device = Constraint(T, V, rule=
Energy_arrival_device_rule)

    # Constraint (2)
    def Battery_device_rule(model, t, i):
        return model.b_i[t, i] + model.w_i[t, i] - B_i[i] <= 0

    model.Battery_device = Constraint(T, V, rule=Battery_device_rule)

    # Constraint (3)
    def application_constraint_rule(model, t, i, s):
        return model.x[t, i, s] <= sum(rho_s_k[s, k] * model.f[i, k] for k
in F)

    model.ApplicationConstraint = Constraint(T, V, Sub, rule=
application_constraint_rule)


    # Constraint (4)
    def subtask_offloading_rule(model, t, i, s):
        # Ensures a subtask is either executed locally or offloaded, but
not both
        return model.x[t, i, s] + model.y[t, i, s] <= 1

    model.SubtaskOffloading = Constraint(T, V, Sub, rule=
subtask_offloading_rule)

```

```
112     # Constraint (5)
113     def computational_capacity_rule(model, t, i):
114         # Sum of CPU cycles required by all subtasks must not exceed the
    device's computational capacity
115         return sum(D_s[s] * W_s[s] * model.x[t, i, s] for s in Sub) <= C_i[
    i]
116
117     model.ComputationalCapacity = Constraint(T, V, rule=
    computational_capacity_rule)
118
119     # Constraint (6)
120     def storage_capacity_rule(model, i):
121         # Sum of storage requirements of all functions configured on device
     must not exceed its storage capacity
122         return sum(sigma_k[k] * model.f[i, k] for k in F) <= S_i[i]
123
124     model.StorageCapacity = Constraint(V, rule=storage_capacity_rule)
125
126     # Constraint (9)
127     def energy_consumption_execution_rule(model, t, i):
128         # Energy consumed for executing subtasks locally on device i at
    time t
129         return sum(model.y[t, i, s] * D_s[s] * W_s[s] * eta/g_t[t,i] for s
    in Sub) <= model.b_i[t, i]
130
131     model.EnergyConsumptionExecution = Constraint(T, V, rule=
    energy_consumption_execution_rule)
132
133     # Constraint (12)
134     def energy_consumption_total_rule(model,t, i):
135         # Energy consumed for executing subtasks locally on device i at
    time t
136         return sum(model.y[t, i, s] * D_s[s] * W_s[s] * eta/g_t[t,i] for s
    in Sub) + sum(model.x[t, i, s] * D_s[s] * W_s[s] * nu for s in Sub) <=
    sum(model.w_i[t,i] for t in T )
137
138     model.EnergyConsumptiontotal = Constraint(T, V, rule=
    energy_consumption_total_rule)
139
```

```
    # Constraint (13)
    def battery_state_update_rule(model, t, i):
        # This constraint updates the battery state based on previous state, energy harvested, and energy consumed
        # Assuming energy consumed for transmission (\lambda^{U}_{t,i}) is calculated or defined elsewhere
        if t == 0:  # Initial condition
            return Constraint.Skip  # Or define an initial state for model.b_i[0, i]
        else:
            return model.b_i[t, i] == model.b_i[t - 1, i] + model.w_i[t - 1, i] - (sum(model.y[t, i, s] * D_s[s] * W_s[s] * eta/g_t[t,i] for s in Sub) + sum(model.x[t, i, s] * D_s[s] * W_s[s] * nu for s in Sub))

    model.BatteryStateUpdate = Constraint(T, V, rule=battery_state_update_rule)

    #Object
    # Decision variables for application completion
    model.z = Var(V, domain=Binary)  # Application completion indicator for each device

    # Constraint (14) to ensure subtask completion leads to application completion
    def application_completion_rule(model, i):
        # For each device i, ensure that if all subtasks are either executed or offloaded, then z[i] can be 1
        return sum(model.x[t, i, s] + model.y[t, i, s] for t in T for s in Sub) >= len(T) * len(Sub) * model.z[i]

    model.ApplicationCompletion = Constraint(V, rule=application_completion_rule)

    # Objective: Maximize the number of completed applications
    def objective_rule(model):
        return sum(model.z[i] for i in V)

    model.objective = Objective(rule=objective_rule, sense=maximize)
```

```
    #  Solve the MILP
    opt = SolverFactory('gurobi', solver_io='python')
    results = opt.solve(model)  # solves and updates instance
    Max_min_Opt = value(model.objective)

    # return necessary information
    dec_slot = 1
    energy_levelc = np.zeros(len(V))
    energy_storec = np.zeros(len(V))
    x_decc = np.zeros((len(V),len(Sub)))
    y_decc = np.zeros((len(V), len(Sub)))
    com_app = np.zeros(len(V))
    for i in range(len(V)):
        energy_levelc[i] = model.b_i[dec_slot,i].value
        energy_storec[i] = model.w_i[dec_slot,i].value
        for j in range(len(Sub)):
            x_decc[i,j] = model.x[dec_slot, i, j].value
            y_decc[i,j] = model.y[dec_slot, i, j].value

    x_judge = np.zeros((2,len(V), len(Sub)))
    y_judge = np.zeros((2,len(V), len(Sub)))
    for ii in range(2):
        for jj in range(len(V)):
            for kk in range(len(Sub)):
                x_judge[ii,jj,kk] = model.x[ii,jj,kk].value
                y_judge[ii,jj,kk] = model.y[ii,jj,kk].value
    for ie in range(len(V)):
        # Calculate the product of the concatenation of (x_i^t + y_i^t) on
    subtask s for all time steps for device i
        product_along_subtasks = np.prod(x_judge[:, ie, :] + y_judge[:, ie,
     :], axis=1)  # Consecutive multiplication over subtask dimensions
        # The results are then summed over all time steps
        product_sum = np.sum(product_along_subtasks)
        com_app[ie] = product_sum >= 1


    return com_app, energy_levelc, energy_storec, x_decc, y_decc
```

Reference

[1] M. Campbell. “Smart edge: The center of data gravity out of the cloud”. In: *Computer* 52 (Dec. 2019), pp. 99–102.

[2] W. Duan, J. Gu, M. Wen, G. Zhang, Y. Ji, and S. Mumtaz. “Emerging technologies for 5G-IoV networks: Applications, trends and opportunities”. In: *IEEE Netw.* 34.5 (Sept. 2020), pp. 283–289.

[3] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella. “On multi-access edge computing: A survey of the emerging 5G network edge cloud architecture and orchestration”. In: *IEEE Commun. Surveys Tuts.* 19.3 (2017), pp. 1657–1681.

[4] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski. “The rise of serverless computing”. In: *Commun. ACM* 62.12 (Nov. 2019), pp. 44–54.

[5] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. “Serverless computation with openlambda”. In: *Proc. 8th USENIX Conf. Hot Topics Cloud Comput.* 2016, pp. 33–39.

[6] C. Cicconetti, M. Conti, A. Passarella, and D. Sabella. “Toward distributed computing environments with serverless solutions in edge systems”. In: *IEEE Commun. Mag.* 58.3 (Mar. 2020), pp. 40–46.

[7] R. Xie, Q. Tang, S. Qiao, H. Zhu, F. R. Yu, and T. Huang. “When serverless computing meets edge computing: Architecture, challenges, and open issues”. In: *IEEE Wireless Commun.* 28.5 (Oct. 2021), pp. 126–133.

[8] T. Rausch, A. Rashed, and S. Dustdar. “Optimized container scheduling for data-intensive serverless edge computing”. In: *Future Gener. Comput. Syst.* 114 (2021), pp. 259–271.

[9] R. Xie, D. Gu, Q. Tang, T. Huang, and F. R. Yu. “Workflow Scheduling in Serverless Edge Computing for the Industrial Internet of Things: A Learning Approach”. In: *IEEE Transactions on Industrial Informatics* 19.7 (July 2023), pp. 8242–8252. doi: 10.1109/TII.2022.3217477.

[10] C. Cicconetti, M. Conti, and A. Passarella. “A decentralized framework for serverless edge computing in the Internet of Things”. In: *IEEE Trans. Netw. Service Manag.* 18.2 (June 2021), pp. 2166–2180.

[11] A. Glikson, S. Nastic, and S. Dustdar. “Deviceless edge computing: Extending serverless computing to the edge of the network”. In: *Proc. 10th ACM Int. Syst. Storage Conf. (SYSTOR)*. 2017, p. 28.

[12] T. He, K.-W. Chin, H. Ren, and X. Liu. “Maximizing Virtual Network Embedding Requests in RF-Charging IoT Networks”. In: *IEEE Commun. Lett.* 26.4 (Apr. 2022), pp. 863–867.

[13] H. Ko, S. Pack, and V. C. M. Leung. “Performance Optimization of Serverless Computing for Latency-Guaranteed and Energy-Efficient Task Offloading in Energy-Harvesting Industrial IoT”. In: *IEEE Internet Things J.* 10.3 (June 2023), pp. 1897–1907.

[14] M. S. Aslanpour, A. N. Toosi, M. A. Cheema, and R. Gaire. “Energy-Aware Resource Scheduling for Serverless Edge Computing”. In: *Proc. 22nd IEEE Int. Symp. Cluster Cloud Internet Comput. (CCGrid)*. June 2022, pp. 190–199.

[15] S. Deng, H. Zhao, Z. Xiang, C. Zhang, R. Jiang, Y. Li, J. Yin, S. Dustdar, and A. Y. Zomaya. “Dependent Function Embedding for Distributed Serverless Edge Computing”. In: *IEEE Trans. Parallel Distrib. Syst.* 33.10 (Oct. 2022), pp. 2346–2357.

[16] A. Das, S. Imai, S. Patterson, and M. P. Wittie. “Performance Optimization for Edge-Cloud Serverless Platforms via Dynamic Task Placement”. In: *Proc. 20th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGRID)*. May 2020, pp. 41–50.

[17] M. Bensalem, F. Carpio, and A. Jukan. “Towards Optimal Serverless Function Scaling in Edge Computing Network”. In: *Proc. IEEE Int. Conf. Commun. (ICC)*. May 2023, pp. 828–833.

[18] C. Cicconetti, M. Conti, and A. Passarella. “A Decentralized Framework for Serverless Edge Computing in the Internet of Things”. In: *IEEE Trans. Netw. Serv. Manag.* 18.2 (June 2021), pp. 2166–2180.

[19] T. P. Bac, M. N. Tran, and Y. Kim. “Serverless Computing Approach for Deploying Machine Learning Applications in Edge Layer”. In: *Proc. Int. Conf. Inf. Netw. (ICOIN)*. Jan. 2022, pp. 396–401.

[20] P. Vahidinia, B. Farahani, and F. Shams Aliee. “Mitigating Cold Start Problem in Serverless Computing: A Reinforcement Learning Approach”. In: *IEEE Internet Things J.* 10.5 (May 2023), pp. 3917–3927.

[21] S. Agarwal, M. A. Rodriguez, and R. Buyya. “A Reinforcement Learning Approach to Reduce Serverless Function Cold Start Frequency”. In: *Proc. IEEE/ACM 21st Int. Symp. Cluster Cloud Internet Comput. (CCGrid)*. May 2021, pp. 797–803.

[22] M.-L. Ku, Y. Chen, and K. J. R. Liu. “Data-Driven Stochastic Models and Policies for Energy Harvesting Sensor Communications”. In: *IEEE J. Sel. Areas Commun.* 33.8 (Aug. 2015), pp. 1505–1520.

[23] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[24] J. Zhan, J. Wu, T. He, and K.-W. Chin. “Task Offloading and Approximate Computing in Solar Powered IoT Networks”. In: *IEEE Netw. Lett.* 6.1 (Mar. 2024), pp. 26–30.

[25] J. Tang, W. P. Tay, T. Q. S. Quek, and B. Liang. “System Cost Minimization in Cloud RAN With Limited Fronthaul Capacity”. In: *IEEE Trans. Wireless Commun.* 16.5 (May 2017), pp. 3371–3384.

本科生毕业设计（论文）评定表

指导教师意见	**论文评语：** In the thesis, the author considers a serverless computing-based IoT system. He proposes a novel problem of maximize the total number of executed applications, and models the problem as an MILP. After that, the authors designs a Receding Horizon Control-based solution. The experimental results demonstrate the effectiveness of the proposed method. This thesis is written well, and the organization structure is proper. Overall, the thesis meets the requirements for undergraduate dissertation. **评定分数：** ____95____ **签章：** **2024 年 3 月 28 日**
评阅人意见	**论文评语：** 占俊飞同学的毕业论文提出了一种两阶段方法用于太阳能物联网网络无服务器计算中的调度，通过数值实验验证了所提出方法的有效性。论文结构清晰，观点明确，论据充分，对实际应用具有一定的参考价值。该论文能达到学士学位论文的要求，同意进行答辩。 **评定分数：** ____91____ **签章：** **2024 年 4 月 14 日**
学院意见	该学生答辩评分成绩为 92，按照规定的记分权重计算出的最终成绩为 92，评分等级为优秀。 分数：92/100 **签章：** 2024 年 6 月 15 日