CoxNet: A Computation Reuse Architecture at the Edge

Zouhir Bellal, Boubakr Nour, *Member, IEEE*, and Spyridon Mastorakis, *Member, IEEE*

***Abstract*—In recent years, edge computing has emerged as an effective solution to extend cloud computing and satisfy the demand of applications for low latency. However, with today's explosion of innovative applications (e.g., augmented reality, natural language processing, virtual reality), processing services for mobile and smart devices have become computation-intensive, consisting of multiple interconnected computations. This coupled with the need for delay-sensitivity and high quality of service put massive pressure on edge servers. Meanwhile, tasks invoking these services may involve similar inputs that could lead to the same output. In this paper, we present *CoxNet*, an efficient computation reuse architecture for edge computing. *CoxNet* enables edge servers to reuse previous computations while scheduling dependent incoming computations. We provide an analytical model for computation reuse joined with dependent task offloading and design a novel computing offloading scheduling scheme. We also evaluate the efficiency and effectiveness of *CoxNet* via synthetic and real-world datasets. Our results show that *CoxNet* is able to reduce the task execution time up to 66% based on a synthetic dataset and up to 50% based on a real-world dataset.**

***Index Terms*—Edge computing, service offloading, computation reuse, serverless computing.**

I. Introduction

TODAY'S networks are witnessing a massive growth in the number of connected devices, such as smartphones, wearable devices, Internet of Things (IoT) [1]. This wave of connected devices led to a major development of information technology, improving human life quality, and calls for paradigms for green networking and computing. Juniper Research has found that the number of IoT devices will reach 50 billion by 2022 [2]. In recent years, Internet service providers have undergone rapid growth in both variety and complexity. The Cisco Annual Internet Report forecasts that connected home applications will dominate Machine-to-Machine communication by 2023 and smart car applications will grow more than 30% over the next two years [3]. Furthermore, both users' and applications' requirements have changed: services are computation-intensive and feature new requirements, such as location-awareness and delay-sensitivity. The traditional cloud computing paradigm [4] along with the limited Internet bandwidth and the long propagation delays cannot meet the desired Quality of Service (QoS) [5].

To address these challenges, edge computing [6] has emerged as an extension of cloud computing to push processing services to the proximity of end-devices deployed at the edge of the Internet (e.g., wireless access points, IoT gateways, and routers) [7]. The deployment of small-scale data-centers at the edge, consisting of edge servers, will offer low latency due to the servers' proximity to the source of data [8]. Although data-centers at the edge are expected to serve a wide range of services, their computation scale may be considerably smaller compared to cloud computing. The growth witnessed in Artificial Intelligent (AI)-based applications led to the birth of highly complex and computation-intensive services, such as facial recognition, natural language processing, and computer vision [9]. These services commonly consist of multiple interdependent sub (atomic) services (Functions-as-a-Service) [10] that form a workflow that can be represented as a Directed Acyclic Graph (DAG). For example, an online social network similar to Facebook may consist of approximately 170 functions on Amazon's Lambda [11].

In addition, today's applications and user demands witness a natural many-to-one relationship between the computation's input-output imposed with location-aware computing. This phenomenon leads to a temporal, spatial, or even semantic correlation among input data and therefore the computation may be mapped to the same output [12]. Let us consider an example where a group of tourists visits Nero's Golden Palace (Domus Aurea), one of the most popular landmarks in ancient Rome. An archaeological Virtual Reality (VR) service powered by nearby edge servers can be offered to visitors, "transporting" them back in time and allowing them to see Nero's Golden Palace as it used to be centuries ago. The visitors can live this experience by wearing virtual reality headsets while touring the site. In this use-case, the massive computation cost required for VR rendering makes it challenging to meet the desired Qualify of Experience (QoE) [13]. However, driven from the fact that the tourists' angles of view could be overlapping, the input data of the computation may be similar, leading to the same output. Eliminating such redundant/duplicate computation by reusing the output results of previous (similar) tasks instead of re-computing them from scratch reduces the

Manuscript received December 31, 2020; revised February 26, 2021 and March 19, 2021; accepted April 2, 2021. Date of publication April 7, 2021; date of current version May 20, 2021. This work was supported in part by the National Institutes of Health under Grant NIGMS/P20GM109090; in part by the National Science Foundation under Award CNS-2016714; and in part by the Nebraska University Collaboration Initiative. *(Corresponding author: Boubakr Nour.)*

Zouhir Bellal is with the LabRI-SBA Lab, Ecole Superieure en Informatique, Sidi Bel Abbes 2045, Algeria (e-mail: z.bellal@esi-sba.dz).

Boubakr Nour is with the School of Computer Science, Beijing Institute of Technology, Beijing 100081, China (e-mail: n.boubakr@bit.edu.cn).

Spyridon Mastorakis is with the Department of Computer Science, University of Nebraska Omaha, Omaha, NE 68182 USA (e-mail: smastorakis@unomaha.edu).

Digital Object Identifier 10.1109/TGCN.2021.3071497

2473-2400 © 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: University of Illinois. Downloaded on September 28,2021 at 23:28:22 UTC from IEEE Xplore. Restrictions apply.

execution cost and enables edge servers with low computation capacity to sustain the expected QoE [14].

To address these issues, we complement edge computing with the reuse of computation, where previously executed tasks are stored at edge servers, so that the outputs of these tasks are reused to execute newly incoming similar tasks. We present *CoxNet*, a novel computation reuse architecture that schedules dependent tasks with deadline constraints. Our evaluation, based on a synthetic and a real-world dataset, shows that *CoxNet* achieves the best performance compared to several baseline strategies, reducing the total execution time by up to 66%.

In summary, our contribution is three-fold:

- First, we extend the edge computing paradigm with the reuse of computation to effectively meet applications' QoS requirements and utilize edge computing resources.
- Second, we present an analytical model for computation reuse combined with computation offloading.
- Third, we design and implement a novel scheduling scheme taking into consideration services represented as DAGs in the context of computation reuse and we validate the performance of the designed architecture based on both synthetic and real-world datasets.

The rest of the paper is organized as follows. Section II presents a brief background and reviews related works. Section III presents different use-cases and motivates *CoxNet*. We introduce our system model and our problem formulation in Section IV. Section V presents in detail the proposed architecture and designed mechanisms. We evaluate the performance and effectiveness of *CoxNet* in Section VI. Finally, we conclude the paper in Section VII.

II. Background & Related Work

Edge Computing: Cloud computing has shown great resilience in providing massive amounts of computing resources [15]. Yet, the requirements of today's applications demand low latency, which may not be satisfied when computing resources are offered by distant clouds. Edge computing, as an extension of cloud computing, has been proposed to overcome this challenge by bringing computing resources physically close to end-users [6]. It is envisioned that edge computing resources will be organized in small-scale data centers at the edge, called *cloudlets* [16]. Given the small scales of cloudlets (in comparison to cloud deployments), the allocation and utilization of their resources become vital issues.

In-Network Computing: Researchers have explored different approaches to enhance QoS by providing computation directly in the network [17], [18]. Instead of providing only network functions, a router is able to examine the received packets and then perform computation locally instead of forwarding the task to remote servers [19]. This concept pushes application developers to re-design the logic of their services and decompose them into multiple sub-services, and then offload them to the edge in order to enable in-network computing [20]. This capability empowers edge serves to support various optimization techniques, such as computation reuse.

Computation Reuse: This concept refers to reusing the execution results of a previously executed function/service for the execution of forthcoming computations [21], [22], [23]. To do so, an edge server stores previous computations (e.g., service name, input data, and output results), and thereafter uses efficient indexing and lookup mechanisms to find similar computation [24]. As a result, the usage of computation resources and the task execution times can be significantly reduced [25]. This computation paradigm is suitable for applications (e.g., image recognition, real-time video processing), where "similar" task input data could lead to the same output (computation results).

Approaches have been proposed to exploit this feature in several domains. For instance, Cachier [26] is one of the first approaches that reduces the number of executed image recognition tasks by extracting features of the requested recognition tasks and seeking to find a match with a similar computed task from a cache. Yet, the same concept can be adopted in other use-cases and applications. Guo *et al.* [12] proposed FoggyCache, a framework that reduces the computation cost of image and audio recognition applications by reusing approximate computation results across devices. The reuse of computation is achieved through an Adaptive Locality Sensitive Hashing (A-LSH) and a Homogenized K-Nearest Neighbors (H-kNN) algorithm. The former indexes input data, while the latter represents the similarity between task input data. Similarly, work in [27] reuses image recognition tasks within a single device and leverages a set of algorithms to assess the input similarity and maximize de-duplication opportunities.

Due to the complexity involved in today's applications, the execution of tasks cannot be seen a monolithic workflow [28]. Instead, the execution consists of sub-tasks following an execution workflow commonly represented by a DAG, different approaches have been proposed to explore computation reuse in dependent tasks computation [29]. For instance, DryadInc [30] and Nectar [31] reduce incremental large-scale computation graphs by storing and reuses the results of performed computations. However, these approaches require that the DAG remains static. Recently, Barreiros, Jr. *et al.* [32] presented an approach that reuses computations to optimize parameter sensitivity analysis. In a sensitivity analysis, a task is performed multiple times over the same dataset with partially varied inputs that open opportunities to reuse computation. A similar approach is proposed in [33], where the authors reduce hyperparameter tuning by reusing the results of overlapping hyperparameter configurations. Yet, due to the sequential nature of the targeted problems, the authors did not seek to reuse the computation of an intermediate node in the DAG if its predecessor was not reused.

Serverless Computing: Serverless computing has received significant attention from both the research community and the industry. Serverless provides various features, such ease-of-maintenance, auto-scaling, and a pay-as-you-go model, that contributes to the fast deployment and maintenance of services/applications [34]. Many cloud service providers have released their own serverless platforms, such as Google Cloud functions, Amazon Lambda, and Microsoft Azure functions. A serverless-based service consists of a collection of loosely coupled stateless functions, where each one

Authorized licensed use limited to: University of Illinois. Downloaded on September 28,2021 at 23:28:22 UTC from IEEE Xplore. Restrictions apply.

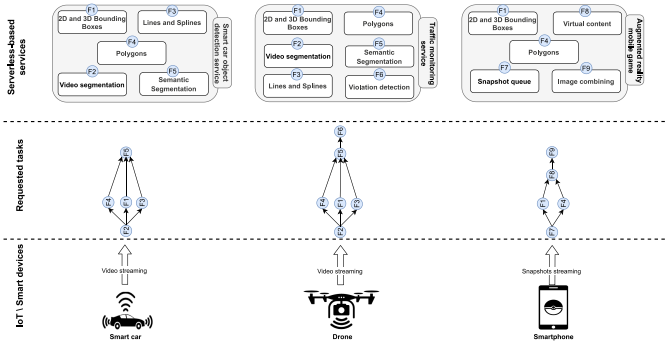


Fig. 1. The serverless paradigm and computation reuse use-cases.

implements specific capabilities. These functions collaborate with each other to accomplish the desired tasks. There can be many interdependent functions, which process and transfer data to each other and return the result to the service consumer [34]. The collection of functions could be deployed at the edge servers using independent containers (e.g., Docker). The reliance of today's services on such a paradigm delivers a complex form of computation, consisting of multiple interconnected functions in the form of DAGs (e.g., Facebook's video processing application [35]). In order to accelerate the development and the deployment of such complex services, many cloud platforms (e.g., AWS) allow developers to share their functions via private application repositories or even public ones, enabling different services to share and use the same public functions. This function sharing among multiple services can help with accelerating computation and application deployment (e.g., machine learning, artificial intelligence algorithms). Moreover, by sharing functions among different services, there is a high probability of re-computing the same function multiple times. This opens opportunities to reduce the required computation by reusing similar computation results instead of recomputing them from scratch. Fig. 1 illustrates an example of three services that use shared functions.

III. Motivation

We envision edge computing environments, where each service consists of an execution graph (workflow) represented as a DAG. Subsequently, a DAG is composed by a set of functions that are executed, so that the overall functionality of a service is realized. In *CoxNet*, a function may be common (shared) among several services, while the execution results of each function are stored by edge servers, so that they can be reused during the execution of the same function with the same or similar input(s) in the future. This reuse of computation enables edge servers to avoid the execution of duplicate computation, thus utilizing their computing resources effectively. To signify the premise of *CoxNet*, let us consider the following scenarios (Fig. 1).

Object Detection Service: Let us consider a scenario, where self-driving vehicles use cameras to avoid obstacles around them through an object detection service at the edge. The workflow of such a service may involve different functions, as shown in Fig. 1. A video segmentation function (F2) is the starting point for this service, since it is responsible for the segmentation of the input video to individual images, which will be passed to subsequent functions for the detection of physical objects. The functions 2D and 3D Bounding Boxes (F1), Lines and Splines (F3), and Polygons (F4) will identify physical objects around the vehicle (e.g., other vehicles, pedestrians, bicycles) [36] and define landline and boundary recognition on roads. Finally, the function semantic segmentation (F5) will enable the annotation of every pixel within an image.

Traffic Monitoring Service: Let us consider a scenario, where drones equipped with cameras and wireless communication capabilities are deployed in a smart city for traffic monitoring purposes [37]. These drones capture and offload video recordings of traffic to a traffic monitoring service, which detects violations, provides the identity of cars, and reports these violations to authorities. Similarly to the object detection service, this service is organized as a DAG that includes two phases. In the first phase, cars are identified in the video streams (F2 $\longrightarrow$ F1, F3, F4 $\longrightarrow$ F5), while, in the second phase, the identified cars are passed to the violation detection function (F6), which identifies the violation and notifies the authorities.

Augmented Reality Mobile Game: Mobile augmented reality games allow virtual content like 3D models, animations, and annotations to be placed on top of a real-world environment [38]. For example, in Pokémon GO, players seek to capture and develop Pokémons, and challenge other players. GPS is used to match the player's location with the virtual world, then augmented reality is used to put a Pokémon creature on top of the real-world. Players can interact with this virtual world using their smartphone cameras. As augmented reality games become more sophisticated, the computing capacity they require exceeds the limited capacity of mobile devices. To this end, the deployment of services at the edge for such games may be necessary. In the service illustrated in Fig. 1, the received snapshots are preserved in a waiting queue (F7) and are analyzed by functions F1 and F4 to identify physical objects in the scene. Subsequently, the virtual content function (F8) associates Pokémon creatures with the identified objects and the player's surroundings. Finally, the image combining function (F9) combines the snapshots of the real and virtual content, so that the combined content can be displayed on the player's device.

These scenarios demonstrate that: 1) edge services can be composed by common functions (each function can be used by several services) and 2) the input to common functions of the same service or among services may be similar enough, thus yielding the same results when these functions are executed. In other words, due to the nature of the service usage, requested (intermediate or final) computations may be redundant. To this end, the results of previous executions of such functions can be stored by edge servers, so that they are reused in the future. For example, if multiple users play Pokémon GO as they navigate through the same city, they may capture videos or images of the same sights from different angles or with different lighting, especially, if these sights are popular (e.g., playing Pokemon GO around the Eiffel Tower). The

Authorized licensed use limited to: University of Illinois. Downloaded on September 28,2021 at 23:28:22 UTC from IEEE Xplore. Restrictions apply.

TABLE I
SUMMARY OF THE MOST COMMONLY USED NOTATIONS

Model Parameters	
$\psi(e)$	The current computation capacity of edge server e
S^e	The set of offloaded services at edge server e
A^e	The set of the accepted tasks at edge server e
D_t	The input data of task t
O_t	The output data of task t
Δ_t	The deadline of task t
W_t	The total workload of task t
G_t	The Directed Acyclic Graph (DAG) of task t
ν_t	The set of sub-tasks of task t
P_t	The remaining time to finish the execution of task t
$\phi(t)$	The complexity of task t
$PDL(\tau)$	The proportional deadline of sub-task τ
$\omega(\tau)$	The workload of sub-task τ
g_τ	The gain of sub-task τ
$Pred_\tau$	The set of direct predecessors of sub-task τ
Suc_τ	The set of direct successors of sub-task τ
σ_τ	The potential reusability of sub-task τ
R_τ	The set of performed computations of sub-task τ
$r_\tau \in R_\tau$	A performed computation of sub-task τ
$rd(r_\tau)$	The rate of computation r_τ
D^r_τ	The input data of computation r_τ
O^r_τ	The result of computation r_τ
$S(r)$	The score of computation r

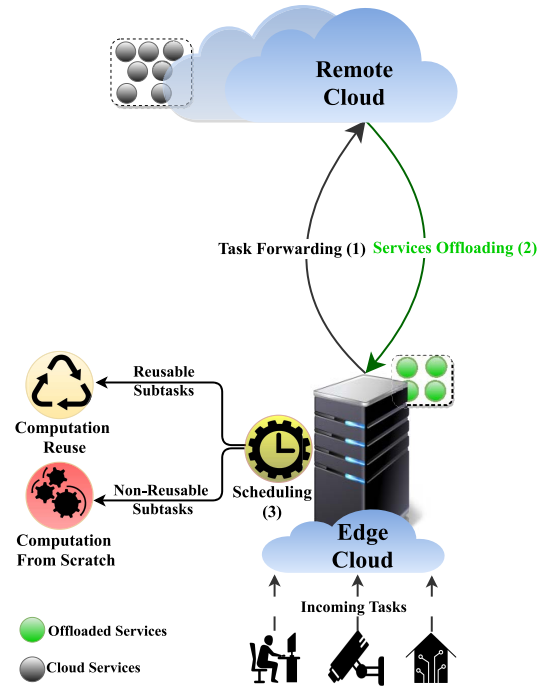


Fig. 2. Reference model.

same applies to vehicles driving close to each other (e.g., adjacent lanes of a highway). Finally, vehicles in crowded streets and users playing augmented reality games in these streets may capture overlapping video frames, thus the same objects may be detected in these frames.

How does CoxNet differ from prior work: CoxNet introduces a deadline-based scheduling architecture for emerging serverless-based edge computing services. Rather than focusing on domain-specific services, *CoxNet* realizes an architecture that capitalizes on offering computation deduplication in a universal manner to all services deployed at the edge. In this context, *CoxNet* aims to achieve the right tradeoff among: 1) identifying which computation to be stored for potential reuse in the future; 2) storing as much computation that may be useful in the future as possible; and 3) alleviating the overhead of searching for computation that can be reused once an incoming task is received by an edge server.

IV. System Model & Problem Formulation

In this section, we present our system model and problem formulation. Table I summarizes the most commonly-used notations in our work.

A. System Model

Network Stakeholders: We consider a system, where a cloud provider v manages a set of services $S = \{s_1, s_2, \ldots, s_n\}$. We assume each service is decomposed into a set of atomic services $s = \{a_1, a_2, \ldots, a_k\}$, where $s \in S$. This decomposition aims to facilitate the execution and deployment of services by composing new services in short periods of time. This composition can be obtained using the dependency information attached to the received task (also called the task's DAG). For each atomic service a, we denote $I(a)$ as the range of possible inputs that can be used to call the atomic service a.

Fig. 2 illustrates a reference model where the cloud provider offloads services based on different factors (e.g., utilization, popularity) to edge servers to improve QoS (e.g., low latency, bandwidth optimization) and QoE. We assume that the service provider has already offloaded important services at the edge. Let $S^e = \{s^e_1, \ldots, s^e_n\}$ denote the set of offloaded services at an edge server e.

Task Modeling: We assume that services are consumed by users with mobile devices. Let $T_s = \{t^s_1, t^s_2, \ldots, t^s_l\}$ denote the set of tasks that invoke a service s over time.

Each task $t \in T_s$ is characterized by input data D_t, output data O_t, a workload W_t, and a deadline constraint Δ_t, which denotes the available time for an edge server to execute the task. $\Delta_t = \Delta'_t - \Gamma(D_t) - \Gamma(O_t)$, where Δ'_t is the deadline constraint specified by the user to receive the task execution results, $\Gamma(D_t)$ and $\Gamma(O_t)$ represent the required time to transmit the input and output data respectively from the user to the edge server and vice versa. We denote by $A^e = \{t_1, \ldots, t_j\}$ a set of currently accepted tasks at an edge server e. For each accepted task $t \in A^e$, P_t denotes the expected time to complete the execution of task t. A task is composed of a set of sub-tasks $t = \{\tau_1, \ldots, \tau_k\}$ that have certain dependencies to each other. These dependencies are represented by a DAG $G_t(\nu_t, \xi)$, where ν_t represents the set of sub-tasks and $\xi = \{(i,j) | i,j \in t, i \rightarrow j\}$ the set of edges that describe the dependencies among sub-tasks. For example, $(i,j) \in \xi$ implies that sub-task τ_j can be executed only and only after sub-task τ_i is completed. In other words, the output of τ_i is required as an input for τ_j. Each sub-task $\tau \in t$ requires the execution of an atomic service a from the set of atomic services that form the service requested by task t. The execution of a sub-task refers to the invocation of the required atomic service with the sub-task inputs. For each sub-task τ,

Authorized licensed use limited to: University of Illinois. Downloaded on September 28,2021 at 23:28:22 UTC from IEEE Xplore. Restrictions apply.

$pred(\tau)$ and $suc(\tau)$ denote the set of direct predecessors and successors of sub-task τ respectively.

The input data $D(\tau)$ of a sub-task τ is given as $D(\tau) = \bigcup_{\tau\prime \in pred(\tau)} O(\tau\prime)$, where $O(\tau\prime)$ is the output of sub-task $\tau\prime$. The sub-task that all of its predecessors have been executed is called a *ready sub-task*.

Each sub-task has a workload $\omega(\tau)$ expressed as an amount of Floating Point Operations Per Second (FLOPs), indicating the required computation cost (i.e., the number of floating-point operations that the system needs to perform to execute sub-task τ). Therefore, the overall computation cost of a task is obtained by $W_t = \sum_{i=1}^{k} \omega(\tau_i)$. Each sub-task has a *proportional deadline PDL*(τ) that represents the execution deadline for sub-task τ. To meet the entire task's deadline Δ_t, the system has to satisfy all of its sub-tasks' *proportional deadlines*.

Computation Modeling: We identify three computation cases: 1) remote execution on the cloud: the task is fully executed at a cloud server; 2) full execution at the edge: the service has been offloaded to the edge where the task is fully executed; and 3) reuse of computation: instead of executing each sub-task from scratch, we reuse the results of a previously executed sub-task that the edge server has stored. Specifically, we reuse the results of sub-tasks with "similar enough" input data, since applying the same computation to similar input will likely generate the same output due to high semantic correlation. In such cases, we may perform a lookup to find the results of similar previously executed sub-tasks, however, if such results cannot be found, we will execute a sub-task from scratch.

The computation cost for each case is shown in Eqs. (1a), (1b), and (1c) respectively:

$$\chi(t) = \frac{\sum_{i=1}^{k} \omega(\tau_i)}{\Delta_t \times \Psi_c} \tag{1a}$$

$$\chi(t) = \frac{\sum_{i=1}^{k} \omega(\tau_i)}{\Delta_t \times \Psi_e} \tag{1b}$$

$$\chi(t) = \frac{W_{lp} \sum_{i=1}^{k}(l_i) + \sum_{i=1}^{k}((1 - x_i) \times \omega(i))}{\Delta_t \times \Psi_e} \tag{1c}$$

where Ψ_c and Ψ_e denote the computation capacity at the cloud and edge server, respectively. The task complexity $\phi(t) = \frac{W_t}{\Delta_t}$ therefore refers to the minimum needed computation capacity to execute task t. x_τ indicates whether the result of sub-task τ is achieved through computation reuse (1) or not (0) and l_τ indicates whether a lookup operation was conducted before the execution of the sub-task τ (1) or not (0). Finally, W_{lp} designates the cost of a lookup operation.

Communication Modeling: A task can be executed either at the edge or the cloud. The communication cost is based on the size of the input data (D_t), the output data (O_t), and the available bandwidth b_q. Eqs. (2a) and (2b) indicate the required communication cost when a task is executed at the cloud and edge respectively.

$$\Gamma(t) = \frac{D_t + O_t}{b_q^v} \tag{2a}$$

$$\Gamma(t) = \frac{D_t + O_t}{b_q^e}. \tag{2b}$$

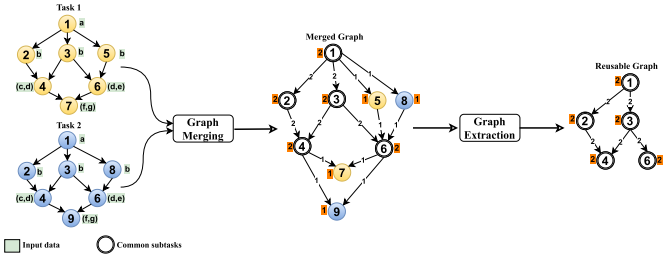


Fig. 3. Example of a merged graph/reusable graph generated from the DAGs of two tasks.

Task's Overall Cost: The overall cost to execute a task is defined as the sum of computation cost for all sub-tasks and communication cost: $\overline{W}_t = \chi(t) + \Gamma(t)$.

B. Problem Formulation

Our problem is defined as an objective to minimize the task's execution time. The same formulation can refer to minimizing the resource utilization. The edge server reuses computation to reduce the overall volume of performed computations (CPU cycles) by reusing previously executed stored sub-tasks, as shown in Eq. (3a).

$$\min_{x,t} \sum_{t \in T} x_t \overline{W}_t \tag{3a}$$

$$\text{subject to} \quad x_t \in \{0, 1\}. \tag{3b}$$

V. *CoxNet*: A Computation Reuse Architecture at the Edge

In this section, we present the design of *CoxNet*, a practical architecture built on top of edge computing. *CoxNet* features a computation reuse-aware design that aims to reduce the utilization of computing resources and the execution time of tasks at the edge. *CoxNet* consists of two main steps: 1) *merged/reusable graph generation:* generate a global view about sub-tasks' invocation and their usage and 2) *task admission policy:* allow edge servers to accept/execute only the tasks that they are capable of meeting their deadlines.

A. Merged/Reusable Graph

Merged Graph ($\mathcal{MG}$): The merged graph is a directed graph that stores DAGs of tasks that have been executed at the edge. Each edge server maintains a merged graph for all offloaded services, which is created gradually as tasks are received and executed. Specifically, after each successful execution of a task, the task's DAG is appended to the merged graph. The merged graph is not only used for aggregating the performed sub-tasks at the edge and the dependencies among them, but also for revealing the estimation of reusing the results of these sub-tasks in future computations. Fig. 3 illustrates an example of merging the DAGs of two tasks to create a merged graph.

Each sub-task τ (vertex in the graph) has a weight value σ_τ that represents the *potential reusability* of the sub-task's computation results. Two sub-tasks τ and $\tau\prime$ are connected with a weighted edge $\rho_{\tau,\tau\prime}$ that represents the *predominance* of a dependency between the two sub-tasks. The predominance

Authorized licensed use limited to: University of Illinois. Downloaded on September 28,2021 at 23:28:22 UTC from IEEE Xplore. Restrictions apply.

refers to how many times an output of τ has been used as an input for $\tau\prime$. Its value is obtained as the number of times that edge $(\tau, \tau\prime)$ is present in DAGs of tasks that have been executed by an edge server.

Each sub-task τ in the merged graph has a set of *records*, which represent previous executions of this sub-task. These records are denoted as $R_\tau = \{r_\tau^1, \ldots, r_\tau^j\}$. A record $r_\tau = (D_\tau^r, O_\tau^r)$ represents a previous execution of sub-task τ with D_τ^r as input data and O_τ^r as the resulting output. Each record r_τ collected during a previous execution of sub-task τ has a rate $rd(r_\tau)$ that indicates the number of times that sub-task τ has been invoked with input D_τ^r. Moreover, each record in the merged graph admits a score *S*(*r*) that indicates the overall impact among other records. This score is calculated using: 1) the rate of the record's input $rd(r_\tau)$ and 2) the workload of the sub-task $\omega(\tau)$. These two metrics are normalized using the normalization technique "the larger the better" to end up with the normalized rate $\overset{*}{r}d(r_\tau)$ and the normalized workload $\overset{*}{\omega}(\tau)$. Thus, the score is given as $S(r) = \sqrt{\overset{*}{\omega}(\tau)\overset{*}{r}d(r_\tau)}$.

The potential reusability metric, $\sigma_\tau = \frac{\sum_{r\in R_\tau} rd(r_\tau)}{|R_\tau|}$, is calculated by the division of how many times a sub-task τ has been computed over how many times its input data was different. $\sum_{r\in R_\tau} rd(r_\tau)$ is the presence of sub-task τ in all the received tasks' DAGs and $|R_\tau|$ is the input variation of the sub-task τ (i.e., the different inputs used by sub-task τ). It is important to indicate that two sub-tasks invoking the same atomic service are considered as identical.

Reusable Graph ($\mathcal{RG}$): The reusable graph is a sub-graph of the merged graph $\mathcal{RG} \subseteq \mathcal{MG}$. It identifies sub-tasks that bring high gain g_τ if they are reused during the execution of incoming sub-tasks in the future. This gain is expressed as $g_\tau = \sqrt{\omega(\tau)\sigma_\tau}$, ensuring a proper balance between the potential reusability of a subtask and its workload. The extraction of $\mathcal{RG}$ from $\mathcal{MG}$ depends on g_{thres} and ρ_{thres}, which indicate the minimum gain values of a sub-task (vertex) and an edge, respectively, in order to be a part of the reusable graph. These threshold values may be dynamically adapted by each edge server based on its available storage resources.

The mechanism to extract a reusable graph from a merged graph consists of two main steps: 1) *vertex-omitted sub-graph*: it starts by removing sub-tasks with a weight under the threshold weight g_{thres} and their incoming and outgoing edges from the merged graph and 2) *edge-omitted sub-graph*: it passes through the remaining graph and deletes each edge with a lower weight than the edge weight threshold ρ_{thres}. Fig. 3 shows an extracted reusable graph from the merged graph for $g_{thres} = \rho_{thres} \geq 2$. For simplicity reason, we assume that the workload of all sub-tasks are the same and they equal to 2.

Dynamic Evolving Graph Processing: The merged graph operations involve adding new vertices or edges, or updating their weights. However, performing these operations based on limited edge computing resources poses challenges since the execution of incoming tasks should be timely, since graph-structured updates (e.g., a set of newly added vertices) are continuously streaming in. As shown in Fig. 4, after the execution of a task, *CoxNet* assigns a *worker* to conduct the required updates on the merged graph for the DAG of each completed

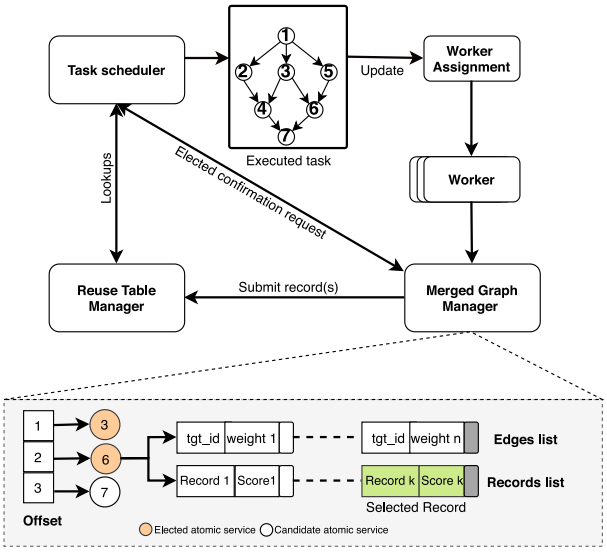


Fig. 4. Dynamic reusable graph generation and usage.

task. Multiple workers can be launched in parallel enabling fast convergence of updates of the merged graph.

As described in Algorithm 1, a worker explores all vertices starting from the root node of a task's DAG and updates the merged graph. For each vertex in the DAG, the worker adds it to the merged graph if itself has not been added already. To achieve that, the *Merged Graph Manager* (MGM) attaches an offset to each inserted vertex to reduce the complexity of searching for a vertex by offering direct access to any vertex for the workers (Line 6). Note that two sub-tasks requesting the same atomic service are represented with the same offset.

After finding the offset of a vertex (Line 8), the worker updates its records (performed computations) either by adding a new record *r* if the requested atomic service has never been executed with the current sub-task input data D_τ^r (Line 21), or by increasing the rate of the record $rd(r_\tau)$ if it has already been performed once before (Line 10). These updates allow workers to also update the record's score of sub-tasks as they are being performed (Line 11). Subsequently, the worker is able to calculate the potential reusability of a sub-task and its gain (Lines 23-24) to decide whether to add it to the reusable graph. To do so, the worker first determines the threshold value (Line 25) indicating minimum gain that a sub-task should have, so that it can be part of the reusable graph. This threshold is changing dynamically over time, which can be obtained as the average between the received task's gain and the former threshold (Line 25).

Next, the worker verifies whether the sub-task's gain surpasses the newly obtained threshold. If so, it submits the sub-task's records to the Reuse Table Manager (RTM) to include them in the reusable graph (Line 28). Based on the records' scores, RTM selects the records with greater scores than the ones it already holds. To do so, RTM always keeps a trace of the record with the minimum score *min_score* after each update. Once one record or more are selected, *CoxNet* marks the requested atomic service for that sub-task as elected (Lines 33-35). From that point and onwards, these selected

Authorized licensed use limited to: University of Illinois. Downloaded on September 28,2021 at 23:28:22 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1: *CoxNet*'s Dynamic Graph Generation

Global Var.: $\mathcal{MG}$: merged graph, M_{thres}: dynamic threshold, min_score: minimum score
Input: G: incoming task's DAG.

1 **Function** *DGGen(G):*
2 offset ← G[root].offset;
3 **if** *(G[root].visited = False)* **then**
4 G[root].visited ← True;
5 **if** *(offset = ∅)* **then**
6 offset ← Add($\mathcal{MG}$, G[root]);
7 **end**
8 v ← GetVertex($\mathcal{MG}$, offset); record ← FindRecord(v, $D_{\mathrm{G[root]}}$);
9 **if** *(record ≠ ∅)* **then**
10 rd(record)++;
11 S(record) ← UpdateScore(r);
12 **if** *(v.elected=True)* **then**
13 **if** *(S(record) > min_score & r.selected = False)* **then**
14 SubmitRecordToRTM(record);
15 **if** *(record.selected = True)* **then**
16 v.SelectedRecords++;
17 **end**
18 **end**
19 **end**
20 **else**
21 AddRecord(v, record);
22 **end**
23 $\sigma_\nu \leftarrow \frac{\sum_{r \in R_\nu} rd(r_\tau)}{|R_\nu|}$;
24 $g_\nu \leftarrow \sqrt{\omega(\nu)\sigma_\nu}$;
25 $g_{thres} \leftarrow \frac{g_{thres}+g_\nu}{2}$;
26 **if** $(g_\nu \geq g_{thres} + \alpha$ & $\nu.elected = False)$ **then**
27 **for** *(record ∈ R_ν)* **do**
28 SubmitRecordToRTM(record);
29 **if** *(record.selected = True)* **then**
30 v.SelectedRecords++;
31 **end**
32 **end**
33 **if** *(v.SelectedRecords > 0)* **then**
34 v.elected ← True;
35 **end**
36 **end**
37 **for** *(c ∈ suc(n))* **do**
38 AddEdges(v, **DGGen(c)**);
39 **end**
40 **end**
41 **return** v;
42 **end**

records are available to be reused in response to incoming sub-tasks. Finally, the worker adds the edges that linked the visited sub-task with its successors after visiting them (Lines 37-39).

Reuse Table Management: The reuse table stores records of sub-tasks that are likely to be reused in the future. The table's size has a strong impact on lookup operation costs. A large table could maximize the potential of computation reuse, but at the same time make lookup operations expensive and lengthy. Regardless of the storage capacity at the edge, the size of the reuse table should be limited and make the best use of this limited size. Therefore, when a worker submits a set of records for a sub-task to be included in the reuse table, the decision to keep or evict an existing record *r* is based on the record's score *S*(*r*) (records with the lowest scores will be evicted). Therefore, whenever all the records of a specific atomic service are evicted, the RTM requests the MGM to wipe out this atomic service from the list of elected ones. Moreover, RTM shares the minimum record score in the reuse table *min_score* with MGM to let workers submit updates for records of an elected vertex only if the scores of these records are higher than the minimum score (Algorithm 1, Lines 13-18).

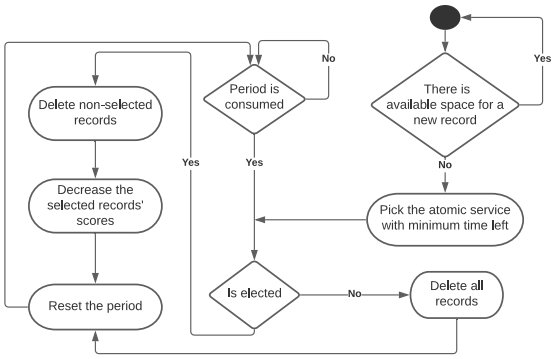


Fig. 5. Merged graph recycling mechanism.

Merged Graph Recycling: The merged graph is stored and managed as an adjacency list based on the architecture proposed in [39] to reduce the complexity of random access operations. Due to the incremental process of the merged graph generation, an efficient mechanism to control its gradual growth and maintain its size is fundamental, especially at an edge server where storage resources are limited. Different mechanisms can be adopted to deal with this issue:

- *Periodic Recycling:* It consists of removing all merged graph's records after a fixed period of time regardless of the merged graph's actual size. However, the available storage space for the merged graph could be fully occupied before the period expires, making the system incapable of adding new records.
- *Threshold-based Recycling:* It consists of deleting all records once the merged graph size reaches a fixed threshold. Although this mechanism ensures space availability for incoming records, it may be unfair among atomic services. For example, let us consider two atomic services a_1 and a_2, where a_1 is inserted in the merged graph at time t_1, while a_2 is inserted after a while at time t_2 ($t_1 < t_2$). Since a_1 is part of the merged graph for a longer period of time than a_2, it is more likely that a_1 becomes a part of the reusable graph.

To address the aforementioned issues, we define a custom mechanism to control the size of the merged graph and, at the same time, guarantee a fair and equal opportunity for all atomic services to join the reusable graph. Essentially, *CoxNet* combines both mechanisms that we mentioned above. As presented in Fig. 5, instead of using a generic period, *CoxNet* uses an independent local period for each atomic service, so that each service is managed individually. Consequently, when an atomic service consumes its local period, MGM deletes its records to release storage space. However, when no space is available to preserve new records, MGM will not wait until one of the atomic services consumes its local period, but it will rather remove the records of the atomic service with the minimum time left.

Although the presented policy guarantees that each atomic service has an equal period to be in the merged graph and ensures the availability of space for forthcoming records,

Authorized licensed use limited to: University of Illinois. Downloaded on September 28,2021 at 23:28:22 UTC from IEEE Xplore. Restrictions apply.

removing the selected records of an already elected atomic service after consuming its local period leads to decreasing the opportunity of computation reuse, since these records are expected to have high reusability. Therefore, *CoxNet* conserves valuable records for as long as they are beneficial. When an elected atomic service consumes its local period, *CoxNet* resets its period and keeps only its selected records while the rest (non-selected records) are deleted. Extending the local period allows these selected records to accumulate high scores compared to the rest of the records in the reuse table. This can lead to essentially making these records untouchable by the eviction policy, since the policy evicts records based on their scores. Therefore, to avoid this side-effect, *CoxNet* decreases the score of selected records to the current minimum score of the reuse table.

B. Task Admission Policy

The limited computing capacity of edge servers poses a challenge, especially when the deadline constraint is the primary factor to consider for received tasks. To prevent edge servers from accepting tasks beyond their computation capabilities, we propose an admission policy that enables edge servers to accept only the tasks, whose deadline constraints they can satisfy. Otherwise, servers forward received tasks that exceed their computing capacity to the cloud.

Algorithm 2 summarizes the *CoxNet* admission policy. Once a task is received for execution by an edge server, *CoxNet* first checks if the requested service is available at the server. If this is not the case, the task is forwarded to the cloud for execution (Lines 2-4). If the requested service is available at the edge, *CoxNet* accepts the task if the server possesses sufficient resources to accommodate its execution. Otherwise, the received task is forwarded to the cloud. To confirm the availability of computing resources, one of the following cases should be verified in *CoxNet*:

- *Case (1):* The computation resources currently available at the edge server *e* are enough to execute and deliver the task within its deadline (Lines 7-10).
- *Case (2):* When the currently available resources are insufficient, *CoxNet* explores whether it is able to wait until the computation resources to be released by terminated tasks are sufficient to satisfy the deadline of the incoming task (Lines 13-22). To do so, *CoxNet* seeks a time-slot $P_{t'}$ after finishing task t', where the computation resources available at that time $\psi(P_{t'})$ will be enough to meet the deadline of the incoming task. If this is the case, *CoxNet* accepts the task and postpones its execution by $P_{t'}$ sec (Lines 18-21). Otherwise, it forwards *t* to the cloud (Lines 24-27). However, postponing the task increases its complexity, hence *CoxNet* recalculates the task's complexity (Line 15) to meet its requirements.

By relying on such an admission policy, *CoxNet* is able to avoid accepting tasks beyond the computational capacity of edge servers and avoid missing the tasks' deadline constraints. It is important to state that whenever a task is accepted, *CoxNet* adopts either a computation from scratch or a computation reuse approach for each sub-task to minimize

Algorithm 2: *CoxNet*'s Admission Policy

```
   Input: s: requested service, t: Task, Δ_t deadline
1  Accept ← False;
2  if (s ∉ S^e) then
3  |  Forward(s, t, Δ_t);  return
4  else
5  |  P_t ← Δ_t;
6  |  ψ_e ← Ψ_e − Σ_{t∈A^e} φ_t;
7  |  if (ψ_e ≥ φ_t) then
8  |  |  Accept ← True;
9  |  |  return 1;
10 |  else
11 |  |  L ← clone(A^e);
   |  |  // Sort A^e based tasks' remaining time P_t
12 |  |  Sort(L, P_t);
13 |  |  while (∃ t' ∈ L & not(Accept) & P_t' < P_t) do
14 |  |  |  ψ_e(P_t') ← ψ_e + φ_t';
15 |  |  |  φ_t ← W_t / (Δ_t − P_t');
16 |  |  |  P_t ← Δ_t − P_t';
17 |  |  |  Remove(L, t');
18 |  |  |  if (ψ_e(P_t') ≥ φ_t) then
19 |  |  |  |  Accept ← True;
20 |  |  |  |  return 1;
21 |  |  |  end
22 |  |  |  ψ_e ← ψ_e(P_t');
23 |  |  end
24 |  |  if (not(Accept)) then
25 |  |  |  Forward(s, t, Δ_t);
26 |  |  |  return 0;
27 |  |  end
28 |  end
29 end
```

the task's execution time. *CoxNet* first calculates the proportional deadlines for all sub-tasks based on the task deadline Δ_t. The proportional deadline of a sub-task τ_i can be obtained as follows:

$$PDL(\tau_i) = \begin{cases} \Delta_t - CT_{\tau_{end}}, & \text{if } \tau_i = \tau_{end} \\ \min\left(\bigcup_{\tau_j \in suc(\tau_i)} PDL(\tau_j)\right) - CT_{\tau_i}, & \tau_i \neq \tau_{end} \end{cases}$$

where $CT_\tau = \frac{\omega(\tau)}{\phi_t}$ denotes the execution time of sub-task τ. Then, *CoxNet* appends an initialization sub-task τ_0 to the task's DAG G_t, which represents the predecessor of all starting sub-tasks. The sub-task τ_0 is considered as a sub-task without a workload ($\omega(\tau_0) = 0$). The initialization sub-task is used as a bridge for the task input data, where $D(\tau_0) = D_t$, and $O(\tau_0) = D(\tau_0)$.

By using the Reuse Table, *CoxNet* is able to search for stored tasks that are similar to the received task and reuse stored results instead of executing the incoming task from scratch. To schedule a task, *CoxNet* starts by scheduling the ready successors of the initialization sub-task $Rsuc(\tau_0)$. This set of sub-tasks are stored based on their proportional deadline. Then, for each sub-task $\tau_j \in Rsuc(\tau_0)$, *CoxNet* checks whether the requested atomic service has been elected. If it has been elected, *CoxNet* searches whether the Reuse Table contains previous executions of this atomic service with input data that is identical/similar to the incoming sub-task's input data $D(\tau_j)$. *CoxNet* directly reuses the output of such a previous execution and considers the sub-task as terminated/executed.

Authorized licensed use limited to: University of Illinois. Downloaded on September 28,2021 at 23:28:22 UTC from IEEE Xplore. Restrictions apply.

TABLE II
EVALUATION PARAMETERS

Items	Parameters
Simulation parameters	
Atomic service workload range	5-15
Atomic service input range	5-25
Atomic service estimated execution time	$100\omega(\tau)$ ns
Lookup workload	5
Threshold	3
Reuse table size (RG size)	400 Records
Number of atomic services	100
Number of services	20
Number of atomic services in the service	30
Number of sub-task in the task	5
Parameters for the Alibaba dataset evaluation	
Atomic service input range	4-16
Lookup workload	5
Reuse table size (RG zise)	300 records
Threshold	Dynamic

If there is no match in the Reuse Table, the sub-task is redirected to the ready queue, so that it is executed from scratch. *CoxNet* maintains a priority-based queue in which sub-tasks with the earliest deadline are prioritized. This lookup procedure could be bypassed, allowing *CoxNet* to submit a sub-task directly to the ready queue when the requested atomic service is not elected. In this fashion, when a sub-task is executed, *CoxNet* iteratively schedules its successors until the whole task is executed.

VI. EVALUATION

In this section, we present the evaluation of *CoxNet* in two phases. In the first phase, we evaluate the performance of *CoxNet* via an extensive simulation study using our analytical model. In the second phase, we conduct our evaluation using the Alibaba cluster dataset.[1] We implemented *CoxNet* in Java and we conducted our experiments on a computer equipped with an Intel Core i7-7500U processor and 8GB of RAM running Windows 10. Our evaluation parameters are shown in Table II. We have evaluated *CoxNet* in comparison with the following strategies:

1) *Always Reuse:* This strategy aims to increase the reuse opportunities by conducting a lookup operation on the reuse table before starting scheduling any subtask. The updates of the reuse table is based on the Last Recently Used (LRU) policy.
2) *Random:* This strategy randomly decides whether a lookup operation on the reuse table should be done. The LRU policy is used to update the reuse table.
3) *Greedy:* Similarly to [40], we implement a greedy strategy. Its goal is to reduce the overall required workload by looking only for sub-tasks that require a high workload within a task.

To gauge the performance of *CoxNet*, we select the following evaluation metrics, which are commonly used for the evaluation of scheduling algorithms and caching policies:

1) *Workload:* The estimated amount of computation that must be performed to complete all tasks received by a server.
2) *Execution time:* The required time to successfully execute all received tasks.
3) *Lookup rate:* The total number of successful lookups (i.e., that found a match in the reuse table) over the total number of performed lookups.

A. Evaluation of Analytical Model

We have implemented *CoxNet* in Java using the *JgraphT*[2] library that covers mathematical graph theory along with a collection of sophisticated graph algorithms. The implementation consists of the admission policy, the task scheduler, and the reuse table manager. We evaluated the impact of input range (number of possible inputs), dynamic threshold, and task's distribution on the metrics we mentioned above.

1) Impact of Input Range: To investigate the impact of the number of possible similar inputs (i.e., the atomic service's input range) on the performance, we generate randomly 10000 tasks and vary the input range within the domain of [1-100]. Fig. 6(a) shows the performance of different strategies expressed by the required workload to complete all tasks. For all strategies, the overall required workload increases after an input range of 4, since with smaller input ranges, the size of the reuse table can still hold all the performed computations. *CoxNet* always achieves the lowest required workload compared to the other three strategies. This stems from two reasons: 1) the efficient management of the reuse table's size: *CoxNet* caches only a valuable subset of the inputs of atomic services and their resulting outputs instead of preserving all possible inputs as the greedy strategy or including worthless ones as the random and the always reuse strategies and 2) minimizing false lookups: *CoxNet* conducts a lookup only for an atomic service with high reusability, maximizing the chances of finding a reusable result in the reuse table. Another promising finding is that even with large input ranges for each atomic service, no extra workload is generated under *CoxNet*. Even in the worst case, its performance converges to the without-reuse performance. However, this is not guaranteed with the rest of the strategies. They all surpass the needed workload without any reuse for input ranges greater than 20. Furthermore, their potential generated extra workloads are extremely high. Notably, with an input range of 100, the always reuse strategy performs 38% more workload to complete all tasks, followed by greedy with 28%, and finally the random strategy with 19%.

To uncover the source of this extra workload, we examined the lookup rate of different strategies. Fig. 6(b) shows the impact of the input range on the strategies' lookup rate. The lookup rate of all baseline strategies records an extreme decline when the reuse table is unable to hold all required computation. The greedy strategy lookup rate decreases from 99% to 6%. The always reuse and random strategies show the same decline. On the other hand, even with a large input range, *CoxNet* conserves a remarkable lookup rate (more than 65%).

[1] Alibaba trace: https://github.com/alibaba/clusterdata/.

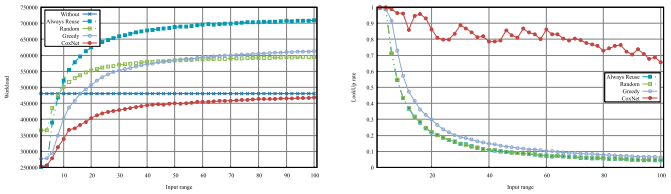
[2] JgraphT: https://jgrapht.org.

Authorized licensed use limited to: University of Illinois. Downloaded on September 28,2021 at 23:28:22 UTC from IEEE Xplore. Restrictions apply.

TABLE III
IMPACT OF TASK DISTRIBUTION ON *CoxNet*'S PERFORMANCE

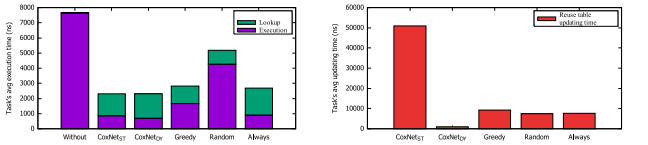
Metrics	Distribution	Strategies									
		1000 Tasks					5000 Tasks				
		Without	*Always*	*Random*	*Greedy*	*CoxNet*	*Without*	*Always*	*Random*	*Greedy*	*CoxNet*
Workload	*Uniform*	375804	270139	323607	210168	213986	1882353	1339425	1612084	1019527	953750
	ZipF	335774	189651	262549	226081	160785	1528848	652975	955202	906150	578758
Execution Time (ns)	*Uniform*	37580400	25115932	31628094	20971754	23025794	188235300	122989506	155750686	95516704	88354142
	ZipF	33577400	16764813	25196363	21372415	14340223	152884800	61888245	107429753	107006598	61634997
Lookup Rate	*Uniform*	/	0.48046	0.4784	0.72882	0.64070	/	0.4866	0.48518	0.74708	0.77698
	ZipF	/	0.6692	0.67053	0.61897	0.74708	/	0.7924	0.79145	0.53099	0.83920
Gain (%)	*Uniform*	/	33	16	44	39	/	35	17	49	53
	ZipF	/	50	25	36	57	/	60	30	30	60

Metrics	Distribution	Strategies				
		10000 Tasks				
		Without	*Always*	*Random*	*Greedy*	*CoxNet*
Workload	*Uniform*	3763134	2678177	3222928	2037633	1855774
	ZipF	2894658	1103075	2000968	2269161	1173575
Execution Time (ns)	*Uniform*	376313400	246187927	311272460	190853737	171838771
	ZipF	289465800	88466726	189338945	214937568	99469134
Lookup Rate	*Uniform*	/	0.48669	0.48589	0.74697	0.82171
	ZipF	/	0.8962	0.89547	0.48649	0.90210
Gain (%)	*Uniform*	/	35	17	49	54
	ZipF	/	69	35	26	66



(a) Workload (b) Lookup

Fig. 6. Impact of input range.



(a) Execution time. (b) Required time to update reuse table.

Fig. 7. The impact of the dynamic threshold on task's execution.

2) Impact of Dynamic Threshold: The reuse table is updated through a background process. Even though the side effect of this process may not impact the task execution time, it could be costly. For that reason, *CoxNet* relies on a dynamic threshold that limits the number of records (i.e., performed computation) to be checked, and accesses only the records that are likely to be part of the reuse table. To investigate the efficiency of this design, we evaluated *CoxNet* with two different configurations. The first is *CoxNet* $_{\mathrm{ST}}$ which implies a static threshold value (i.e., fixed to 3), while a dynamic value is used in the second configuration (i.e., *CoxNet* $_{\mathrm{DY}}$). For this evaluation, we randomly generate 10000 tasks and we compare both configurations with the baseline strategies based on the average time required to update the reuse table. Fig. 7(a) shows the task's average execution time split between time spent on a lookup and the time consumed for execution from scratch. As we observe, there is no significant difference between the two configurations. Both of them reduce the required execution time by 70%. However, there is an indisputable difference in the average time spent on a reuse table update between the configurations. Fig. 7(b) shows the average time required to update the reuse table. *CoxNet* with a dynamic threshold configuration is 53x faster than *CoxNet* with a static threshold. Moreover, it outperforms other baseline strategies. It is 10x faster compared to greedy, and 8x faster than both random and always reuse.

3) Impact of Task Distribution: The distribution of the requests for atomic services could impact the performance of a computation reuse strategy. For example, a workload-centered computation reuse strategy may not be effective for low request frequencies for atomic services with a high workload. Likewise, a distribution that asserts similar request frequencies among the offloaded atomic services could put a frequency-centered computation reuse strategy in a disadvantaged situation. To explore the impact of the atomic service request distribution on the *CoxNet* performance, we evaluated the performance of different strategies under two distributions:

- *Continuous uniform distribution:* For this distribution, we use a linear congruential generator to generate similar request frequency rates for each atomic service. Then we rely on this distribution to randomly generate 10000 tasks.
- *Zipf distribution:* The Internet traffic follows the Zipf distribution, which is a continuous probability distribution. We generate 10000 tasks following the Zipf distribution.

Table III presents the performance for all strategies under both distributions based on the workload, the lookup rate, the

Authorized licensed use limited to: University of Illinois. Downloaded on September 28,2021 at 23:28:22 UTC from IEEE Xplore. Restrictions apply.

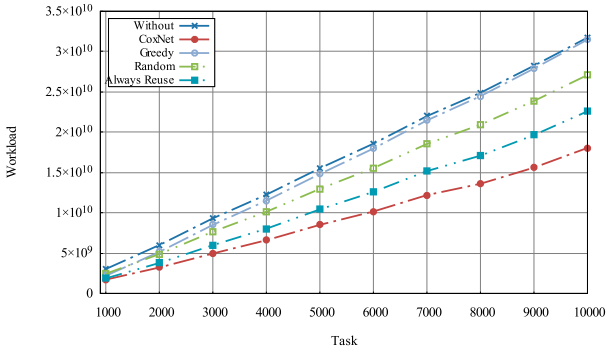


Fig. 8. The total workload of tasks.

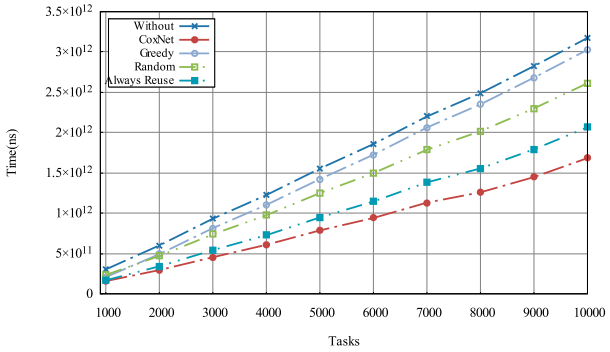


Fig. 9. The total task execution time.

execution time, and the delivered gain (compared to cases where reuse is not applied). Under the continuous uniform distribution, the greedy strategy achieves the best results. It achieves a lower workload than *CoxNet* and outperforms the rest of the strategies by reducing the execution time by 44%. On the other hand, *CoxNet* delivers an acceptable performance. It reduces the execution time by 39% compared to no reuse. However, its outcome improves while increasing the number of tasks. After 5000 tasks, *CoxNet* achieves the best performance in terms of execution time by achieving a reduction of 53%. This improvement is due to the effect of the number of tasks on the lookup rate, since the accuracy of identifying reusable computations increases and the lookup decision-making becomes more precise.

However, under the Zipf distribution, both always reuse and the random strategies show improved performance. This distribution also has a positive impact on *CoxNet*'s performance. For 1000 tasks, *CoxNet* outperforms all other strategies and achieves the same amount of reduction of the execution time (60%) as the always reuse strategy after completing 5000 tasks. On the other hand, greedy is the only strategy that receives an adverse impact under the Zipf distribution due to focusing only on time-consuming atomic services. Overall, the results confirm that *CoxNet* achieves high performance even with extreme task distributions.

B. Real Dataset Experiments

In this subsection, we evaluate *CoxNet* with a real-world dataset. We use Alibaba's dataset, which has almost 3 million jobs (which we call a task in this work) with their associated DAG dependency information as well as the duration of each sub-task and its resource utilization in the cluster (e.g., CPU, memory). We pre-process the dataset to keep only the terminated dependent tasks. Moreover, we normalize sub-tasks in terms of CPU usage, so that all sub-tasks occupy 100% of the CPU. We also reduce the duration of sub-tasks to be coherent with the full usage of the CPU. For instance, if a sub-task occupies 75% of the CPU, its duration is reduced by 25%. We extract a total of 10000 tasks, where the average number of sub-tasks within a task is 4. Furthermore, the sub-tasks' durations are scaled to a millisecond for every minute to make them compatible with the characteristics of edge computing workloads.

Workload Performance: To determine the workload performance, we first assign a workload for each atomic service. This workload is estimated based on the duration of the atomic service. Each 100ns represents one unit of workload (e.g., FLOP). The workload of a lookup operation is set as static regardless of the applied computation reuse strategy.

We compare *CoxNet* with the different strategies based on the total workload needed to complete all received tasks. The results are shown in Fig. 8. *CoxNet* results in the lowest workload among all strategies, reducing the overall required workload by 43-47% compared to cases of no reuse. On the other hand, the *greedy*, *random*, and *always reuse* strategies reduce the total needed workload by 1% (up to 28%), 15% (up to 19%), and 29% (up to 38%), respectively compared to cases of no reuse. This difference among the strategies relates directly to the extra workload generated from false lookups.

Execution Time Performance: We further evaluated the different strategies based on execution time with Fig. 9 showing the evaluation results. The results tie well with former experience (i.e., the workload performance experiment), where all the strategies reflect the same behavior compared to the one observed in the previous experiment. The execution time required by *CoxNet* is lower by $147 \times 10^9(ns)$ up to $1491 \times 10^9(ns)$ than the execution time needed without computation reuse. At the same time, $147 \times 10^9(ns)$, $562 \times 10^9(ns)$, and $1109 \times 10^9(ns)$ are the maximum corresponding execution time reduction under the greedy, random, and always reuse strategies, respectively. A further novel finding is that the real performance is higher than the one revealed on the workload experiment. The performance of *CoxNet*, greedy, and the random strategy is greater by 4% compared to their performance in terms of workload. Only the always reuse strategy achieves a performance greater than 6% compared to its workload performance. This equal variation of the first three strategies proves that the time spent on conducting a lookup is similar among these strategies, while the difference for the always reuse strategy can be explained by the absence of lookup-decision-making operations (i.e., no time is spent on deciding whether to conduct a lookup or compute the sub-task from scratch).

Lookup Rate Performance: Fig. 10 shows the lookup rate during the execution of different strategies. Robust results are achieved by *CoxNet*, since its lookup rate starts from 75% and increases until stabilizing to 84% after executing 3000 tasks. On the contrary, the lookup rates of other strategies

Authorized licensed use limited to: University of Illinois. Downloaded on September 28,2021 at 23:28:22 UTC from IEEE Xplore. Restrictions apply.

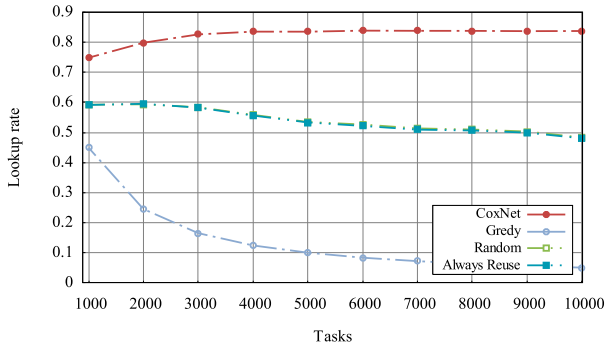


Fig. 10. The total lookup rate.

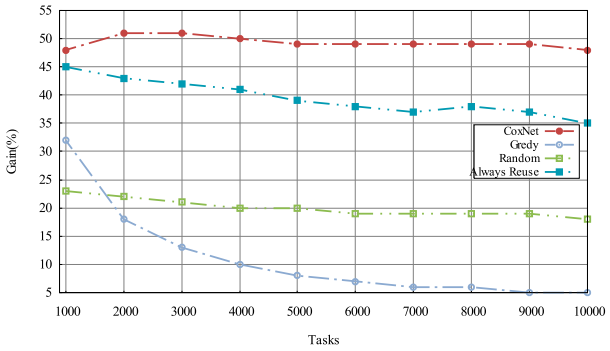


Fig. 11. The overall reduction.

are not stable during the experiment, where their performance is inversely related to the number of tasks. The performance of the greedy strategy drops from 45% with 1000 tasks to 5% after finishing 10000 tasks. The greedy strategy applies a cache management policy that focuses only on high workload computations regardless of their popularity. However, under a near Zipf distribution of the atomic services as the one present in the Alibaba dataset, this shortcoming leads to a saturation of the reuse table with computation that is not beneficial for future reuse. The other strategies show an equivalent behavior. On the other hand, *CoxNet* stores the most valuable computations for future use by essentially providing a cache with two layers (i.e., merged graph, reusable graph) that takes into account the potential reusability of an atomic service.

Moreover, the lookup decision policy applied in *CoxNet*, in contrast to other strategies, guarantees an awareness of the cache's preserved computations. In other words, a lookup is conducted only if an atomic service requested by a sub-task is elected, which implies that the reuse table holds at least one reusable computation for this atomic service. As a result, *CoxNet* achieves the lowest probability of false lookups.

Gain Performance: We evaluated the gain (i.e., the time reduction achieved by each of the strategies) throughout the experiment. Fig. 11 shows the gain for the all strategies. The results show that the minimum gain achieved by *CoxNet* is initially 48% (after finishing 1000 tasks). This is due to the limited number of tasks that lead to similar potential reusability for the invoked atomic services and their election. This explains the highest amount of false lookups at this point (27%) compared to the rest of the experiment (16-20%). After this point, *CoxNet*'s gain increases and stabilizes around 50% during the rest of the experiment. This confirms the positive impact of the number of tasks on *CoxNet*'s performance. However, this is not the case with other strategies, since they all get adversely impacted as the number of tasks increases. Random and always reuse strategies achieve their maximum gains with small number of tasks. Their gains start to decrease gradually while the number of tasks increases. Although the always reuse strategy explores all the reuse opportunities, it does not deliver the highest performance. This confirms the efficiency of *CoxNet*, which reuses computation only when it is likely to perform a positive lookup (i.e., when an atomic service is elected). On the other hand, the greedy strategy follows an exponentially decreasing performance, since the reuse table gets rapidly filled with worthless computations for future reuse. This phenomenon is clearly shown in Fig. 11, where with a limited number of tasks, the reuse table can meet 45% of lookups. However, as the number of tasks increases and the computation results stay in the reuse table without being reused, the performance of the greedy strategy declines dramatically.

VII. Conclusion

In this paper, we proposed *CoxNet*, a computation reuse architecture at the edge that enables edge servers to reuse previous computations while scheduling inter-dependent tasks. *CoxNet* has been validated via an analytical system model and evaluated using synthetic and real-world datasets. *CoxNet* achieved the best performance among various task scheduling and execution strategies due to its efficient computation caching and lookup mechanisms. Finally, *CoxNet* consistently outperformed all compared strategies in terms of crucial factors, such as the range of possible inputs that sub-task can be invoked with, the arrival task distributions, and the post-execution burden.

References

[1] F. Khan, M. A. Jan, A. U. Rehman, S. Mastorakis, M. Alazab, and P. Watters, "A secured and intelligent communication scheme for IIoT-enabled pervasive edge computing," *IEEE Trans. Ind. Informat.*, vol. 17, no. 7, pp. 5128–5137, Jul. 2021.

[2] *The Internet of Things: Consumer, Industrial & Public Services 2020–2024*. Accessed: Nov. 2020. [Online]. Available: https://www.juniperresearch.com/researchstore/devices-technology/internet-of-things-iot-data-research-report

[3] *Cisco Annual Internet Report (2018–2023)*. Accessed: Nov. 2020. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11–741490.html

[4] J. Pan and J. McElhannon, "Future edge cloud and edge computing for Internet of Things applications," *IEEE Internet Things J.*, vol. 5, no. 1, pp. 439–449, Feb. 2018.

[5] P. Hofmann and D. Woods, "Cloud computing: The limits of public clouds for business applications," *IEEE Internet Comput.*, vol. 14, no. 6, pp. 90–93, Nov./Dec. 2010.

[6] A. Filali, A. Abouaomar, S. Cherkaoui, A. Kobbane, and M. Guizani, "Multi-access edge computing: A survey," *IEEE Access*, vol. 8, pp. 197017–197046, 2020.

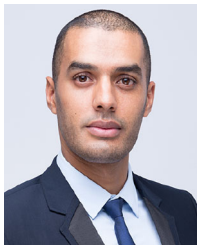
[7] S. Mastorakis, X. Zhong, P.-C. Huang, and R. Tourani, "DLWIoT: Deep learning-based watermarking for authorized IoT onboarding," 2020. [Online]. Available: arXiv:2010.10334.

[8] O. Serhane, K. Yahyaoui, B. Nour, and H. Moungla, "A survey of ICN content naming and in-network caching in 5G and beyond networks," *IEEE Internet Things J.*, vol. 8, no. 6, pp. 4081–4104, Mar. 2021.

Authorized licensed use limited to: University of Illinois. Downloaded on September 28,2021 at 23:28:22 UTC from IEEE Xplore. Restrictions apply.

[9] X. Li *et al.*, "Hardware impaired ambient backscatter NOMA systems: Reliability and security," *IEEE Trans. Commun.*, early access, Jan. 11, 2021, doi: 10.1109/TCOMM.2021.3050503.
[10] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proc. Symp. Cloud Comput.*, 2017, pp. 445–451.
[11] G. Adzic and R. Chatley, "Serverless computing: Economic and architectural impact," in *Proc. Joint Meeting Found. Softw. Eng.*, 2017, pp. 884–889.
[12] P. Guo, B. Hu, R. Li, and W. Hu, "FoggyCache: Cross-device approximate computation reuse," in *Proc. Annu. Int. Conf. Mobile Comput. Netw. (MobiCom)*, 2018, pp. 19–34.
[13] B. Nour, S. Mastorakis, and A. Mtibaa, "Compute-Less Networking: Perspectives, Challenges, and Opportunities," *IEEE Netw.*, vol. 34, no. 6, pp. 259–265, Nov./Dec. 2020.
[14] B. Nour and S. Cherkaoui, "How far can we go in compute-less networking: Computation correctness and accuracy," 2021. [Online]. Available: arXiv 2103.15924.
[15] M. Armbrust *et al.*, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, 2010.
[16] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for VM-based cloudlets in mobile computing," *IEEE Pervasive Comput.*, vol. 8, no. 4, pp. 14–23, Oct./Dec. 2009.
[17] M. Tang, L. Gao, and J. Huang, "Communication, computation, and caching resource sharing for the Internet of Things," *IEEE Commun. Mag.*, vol. 58, no. 4, pp. 75–80, Apr. 2020.
[18] M. Król, S. Mastorakis, D. Oran, and D. Kutscher, "Compute first networking: Distributed computing meets ICN," in *Proc. ACM Conf. Inf. Centric Netw.*, 2019, pp. 67–77.
[19] A. Yousefpour *et al.*, "All one needs to know about fog computing and related edge computing paradigms: A complete survey," *J. Syst. Archit.*, vol. 98, pp. 289–330, Sep. 2019.
[20] P. Mach *et al.*, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 3, pp. 1628–1656, 2017.
[21] S. Mastorakis, A. Mtibaa, J. Lee, and S. Misra, "ICedge: When edge computing meets information-centric networking," *IEEE Internet Things J.*, vol. 7, no. 5, pp. 4203–4217, May 2020.
[22] B. Nour *et al.*, "A network-based compute reuse architecture for IoT applications," 2021. [Online]. Available: arXiv 2104.03818
[23] J. Lee, A. Mtibaa, and S. Mastorakis, "A case for compute reuse in future edge systems: An empirical study," in *Proc. IEEE Global Commun. Conf. Workshop*, 2019, pp. 1–6.
[24] B. Nour, S. Mastorakis, and A. Mtibaa, "Whispering: Joint service offloading and computation reuse in cloud-edge networks," in *Proc. IEEE Int. Conf. Commun. (ICC)*, 2021, pp. 1–6.
[25] A.-C. Nicolaescu, S. Mastorakis, and I. Psara, "Store edge networked data (SEND): A data and performance driven edge storage framework," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, 2021, pp. 1–11.
[26] U. Drolia, K. Guo, J. Tan, R. Gandhi, and P. Narasimhan, "Cachier: Edge-caching for recognition applications," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst. (ICDCS)*, 2017, pp. 276–286.
[27] P. Guo and W. Hu, "Potluck: Cross-application approximate deduplication for computation-intensive mobile applications," in *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2018, pp. 271–284.
[28] S. Sundar and B. Liang, "Offloading dependent tasks with communication delay and deadline constraint," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, 2018, pp. 37–45.
[29] J. Xu, L. Chen, and P. Zhou, "Joint service caching and task offloading for mobile edge computing in dense networks," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, 2018, pp. 207–215.
[30] L. Popa, M. Budiu, Y. Yu, and M. Isard, "DryadInc: Reusing work in large-scale computations," in *Proc. HotCloud*, vol. 9, pp. 2–6, 2009.
[31] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang, "Nectar: Automatic management of data and computation in datacenters," in *Proc. OSDI*, vol. 10, 2010, pp. 75–888.
[32] W. Barreiros, Jr., *et al.*, "Optimizing parameter sensitivity analysis of large-scale microscopy image analysis workflows with multilevel computation reuse," *Concurr. Comput. Pract. Exp.*, vol. 32, no. 2, 2020, Art. no. e5403.
[33] L. Li, E. Sparks, K. Jamieson, and A. Talwalkar, "Exploiting reuse in pipeline-aware hyperparameter tuning," 2019. [Online]. Available: arXiv:1903.05176.
[34] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: A survey of opportunities, challenges and applications," 2020. [Online]. Available: arXiv:1911.01296.
[35] Q. Huang *et al.*, "SVE: Distributed video processing at Facebook scale," in *Proc. Symp. Oper. Syst. Principles*, 2017, pp. 87–103.
[36] M. Abbasi, A. Najafi, M. Rafiee, M. R. Khosravi, V. G. Menon, and G. Muhammad, "Efficient flow processing in 5G-envisioned SDN-based Internet of Vehicles using GPUs," *IEEE Trans. Intell. Transp. Syst.*, early access, Dec. 7, 2020, doi: 10.1109/TITS.2020.3038250.
[37] B. Ji *et al.*, "A survey of computational intelligence for 6G: Key technologies, applications and trends," *IEEE Trans. Ind. Informat.*, early access, Jan. 18, 2021, doi: 10.1109/TII.2021.3052531.
[38] S. L. Kim, H. J. Suk, J. H. Kang, J. Mo Jung, T. H. Laine, and J. Westlin, "Using unity 3D to facilitate mobile augmented reality game development," in *Proc. IEEE World Forum Internet Thin. (WF-IoT)*, 2014, pp. 21–26.
[39] W. Jaiyeoba and K. Skadron, "GraphTinker: A high performance data structure for dynamic graph processing," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2019, pp. 1030–1041.
[40] L. Liu, H. Tan, S. H.-C. Jiang, Z. Han, X.-Y. Li, and H. Huang, "Dependent task placement and scheduling with function configuration in edge computing," in *Proc. IEEE/ACM Int. Symp. Qual. Service (IWQoS)*, 2019, pp. 1–10.

Zouhir Bellal received the B.Sc. and M.Sc. degrees in computer science from Djillali Liabes University, Sidi Bel Abbes, Algeria, in 2016 and 2014, respectively. He is currently pursuing the Ph.D. degree with the Ecole Superieure en Informatique, Sidi Bel Abbes. His research interests include human-computer interaction, edge computing, and in-network computing.



Boubakr Nour (Member, IEEE) received the Ph.D. degree in computer science and technology from the Beijing Institute of Technology, Beijing, China. His research interests include next-generation networking. He is a recipient of the Best Paper Award at IEEE GLOBECOM in 2018, and the Excellent Student Award at Beijing Institute of Technology in 2016, 2017, and 2018.

Spyridon Mastorakis (Member, IEEE) received the five year Diploma degree (equivalent to M.Eng.) in electrical and computer engineering from the National Technical University of Athens in 2014, and the M.S. and Ph.D. degrees in computer science from the University of California at Los Angeles in 2017 and 2019, respectively. He is an Assistant Professor in Computer Science with the University of Nebraska Omaha. His research interests include network systems and protocols, edge computing and IoT, and security.

Authorized licensed use limited to: University of Illinois. Downloaded on September 28,2021 at 23:28:22 UTC from IEEE Xplore. Restrictions apply.