Energy-Aware Resource Scheduling for Serverless Edge Computing

Mohammad Sadegh Aslanpour* †, Adel N. Toosi*, Muhammad Aamir Cheema*, and Raj Gaire†
*Department of Software Systems and Cybersecurity, Faculty of Information Technology,
Monash University, Clayton, Australia
†CSIRO DATA61, Australia

2022 22nd International Symposium on Cluster, Cloud and Internet Computing (CCGrid) | 978-1-6654-9956-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/CCGRID54584.2022.00028

***Abstract*—In this paper, we present energy-aware scheduling for Serverless edge computing. Energy awareness is critical since edge nodes, in many Internet of Things (IoT) domains, are meant to be powered by renewable energy sources that are *variable*, making low-powered and/or overloaded (bottleneck) nodes unavailable and not operating their services. This awareness is also required since energy challenges have not been previously addressed by Serverless, largely due to its origin in cloud computing. To achieve this, we formally model an energy-aware resource scheduling problem in Serverless edge computing, given a cluster of battery-operated and renewable-energy powered nodes. Then, we devise zone-oriented and priority-based algorithms to improve the operational availability of bottleneck nodes. As assets, our algorithm coins terms "sticky offloading" and "warm scheduling" in the interest of the Quality of Service (QoS). We evaluate our proposal against well-known benchmarks using real-world implementations on a cluster of Raspberry Pis enabled with container orchestration, Kubernetes, and Serverless computing, OpenFaaS, where edge nodes are powered by real-world solar irradiation. Experimental results achieve significant improvements, up to 33%, in helping bottleneck node's operational availability while preserving the QoS. With energy awareness, now Serverless can unconditionally offer its resource efficiency and portability at the edge.**

***Index Terms*—edge computing, Serverless, function-as-a-service, energy-aware, scheduling**

I. Introduction

Edge computing is generally understood as a cluster of tiny computers, i.e., edge nodes, connected with a central, and presumably more powerful, computer as the gateway/controller at the edge of network [23]. Among all proposed architectures [23], edge computing platforms that are fully or partially independent of the cloud are becoming increasingly popular, which are sometimes called mist computing, extreme edge or things-edge in the literature [16], [23]. This architecture is unique due to the hybrid role of edge nodes—simultaneous task generation (sensing/actuation) and computation—and, more importantly, their power supply challenges. Single-board Computers (SBCs) such as Raspberry Pis (Pis) have constantly been used as edge nodes in extreme edge architecture, mainly due to their low-power ARM architecture [7], [8].

In remote applications where extreme edge architecture is the platform of choice, edge nodes are generally powered by electricity generated from local renewable energy sources and local energy storage devices. In practice, this architecture is home to the IoT applications such as Smart City, Smart Manufacturing, and, particularly, Smart Farming and Forestry.

However, renewable energy sources are known to be unstable [11], intermittent, unpredictable [14] or simply variable. The *variability* is also expected in load generation on IoT devices. If such variability is disregarded, it results in the unavailability of some edge nodes—we call them bottlenecks—due to power outage, overloading, and node failure while others waste their excess or unused energy [12]. A bottleneck can significantly affect the flow of a system. In edge clusters, ignoring bottleneck nodes severely increases the difference between the operational availability—the time duration a node is up and running—of bottleneck nodes than that of other nodes. This ignorance prohibits the seamless operation of the entire cluster where a bottleneck node will leave its servicing area uncovered during its unavailability [15].

Given this issue, edge systems are desperate for energy efficiency mechanisms while demanding the preservation of the Quality of Service (QoS) [16]. As an incentive of edge, this prospective mechanism can benefit from enabled resource sharing among edge nodes, by computation offloading, to tackle the issue [23]. Offloading allows migrating computation, e.g., services, to peers. Hence, a resource scheduler that dynamically considers the energy state of bottleneck nodes and migrates services is demanded. However, mounting all energy variability challenges to merely a scheduler appears naive since the deployment (monolithic or microservices), scalability (single unit or allowing multiple units), and portability of services play critical roles.

Serverless Computing has recently attracted special attention as the edge computing facilitator [5]. Serverless offers simplified deployment, computing efficiency through scalability, and most importantly, portability, under the banner of Function-as-a-Service (FaaS) [5]. Serverless decomposes the application into microservices and containerizes them [5]. The finished product is one or a set of Functions, often wrapped as containers. Serverless places a function on a node and dynamically auto-scales required resources according to the demand, even to zero, by adjusting the number of instances. Functions are often stateless, and function scaling results in new identical instances or so-called *Replicas*. The **key question** is *how to achieve energy-aware resource scheduling in Serverless edge computing to improve operational availability*

978-1-6654-9956-9/22/$31.00 ©2022 IEEE
DOI 10.1109/CCGrid54584.2022.00028

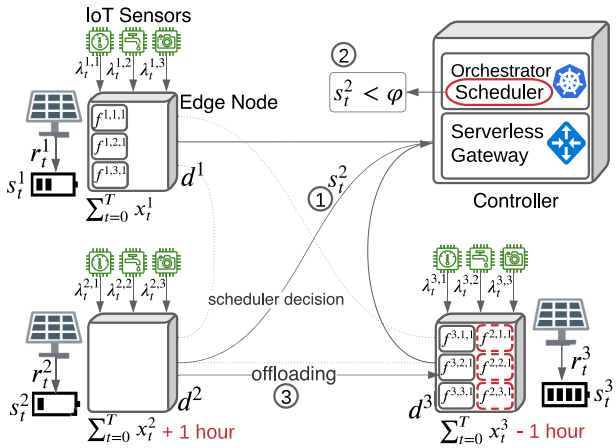

Fig. 1: A schematic view of the system

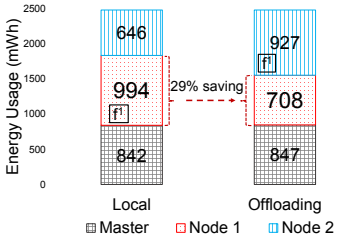

Fig. 2: Saving energy by offloading—2 experiments using 2 Pis for 15 minutes where Node 1 generates tasks and hosts the function locally versus offloading the function to Node 2.

while preserving QoS on renewable energy- and battery-powered devices? Addressing this is not trivial since Serverless platforms are natively designed for cloud computing [5], hence agnostic to energy conditions.

Energy-aware resource scheduling at Serverless edge computing has not been considered previously [18]. Big IT industries hesitate to recognize the energy necessity, despite initial steps in redesigning edge-friendly Serverless platforms such as Amazon IoT Greengrass and Microsoft Azure IoT Edge [18]. Technical limitations also include: nodes should actively collaborate while hosting their functions; nodes with a high level of energy availability may receive an excessive number of peers' functions, making them prone to failure; offloading functions to peers is not free of cost as it imposes communication costs, and nodes have a bounded capacity and cannot accept any given computation while the scheduler is desperate to place all functions.

To address the above challenges, we make the following key **contributions**:

- the problem of energy-aware resource scheduling for renewable energy- and battery-powered edge nodes running on a Serverless edge computing is formally formulated;
- an energy-aware resource scheduling algorithm is designed and implemented that works in a zone- and priority-based manner with features such as "warm scheduling" and "sticky offloading";
- a real-world Serverless edge prototype is implemented using Kubernetes and OpenFaaS on Raspberry Pis; and
- the evaluation of proposed scheduling is performed by developing a realistic containerized use case of IoT applications from the Smart Farming domain and by real traces of renewable energy.

II. System Model and Problem Formulation

A schematic view of the motivating system is shown in Fig. 1 where edge nodes are connected together and to a controller node. Edge nodes are powered by renewable energy sources and storage devices (e.g., batteries) with an unequal energy generation rate. They are equipped with sensing capability generating IoT task requests at variable and irregular rates. A Serverless computing platform orchestrates resources at the edge and assigns tasks to the corresponding function. A controller performs the role of the gateway and scheduler simultaneously. The controller gateway distributes requests to its corresponding function for execution. The scheduler monitors nodes' State of Charge (SoC) and exploits offloading to place functions on well-powered nodes. For example, in Fig. 1, functions of node d^2 are moved to d^3 as this node suffers from low energy density. Our early observations, as shown in Fig. 2, demonstrate that the offloading technique can considerably save energy (by up to 29%), without imposing additional overhead. For instance, a bottleneck node in production, enabled with Industry 4.0, can improve the production line by 29%.

A. System Model

We discretize the time into equal time slots defined as $\mathbb{T} = \{t \mid t \in [0, T], (t+1) - t = \Delta t \text{ seconds}\}$ and the scheduler revisits the placements at the beginning of each time slot.

Edge Nodes: A set of nodes defined as $D = \{d^i \mid i \in [1, n]\}$ is considered as a cluster of edge devices connected with surrounding IoT sensors in a star-like network. A central controller which is excluded from D and is not necessarily powered by renewable energy, performs management, scheduling, etc. SoC and resource (CPU) capacity are among major characteristics of d^i which are later used for scheduling decisions. SoC represents the actual battery charge on nodes at each time-slot and is defined by $S = \{s_t^i \mid i \in [1, n], t \in \mathbb{T}, \vartheta \geq s_t^i \geq 0\}$ where ϑ indicates the maximum battery charge. SoC depends on both energy input and energy consumption. Let $R = \{r_t^i \mid i \in [1, n], t \in \mathbb{T}, r_t^i \geq 0\}$ denote the energy input (e.g., renewable energy). Energy consumed on nodes during t is denoted as $E = \{e_t^i \mid i \in [1, n], t \in \mathbb{T}, e_t^i \geq 0\}$ which depends on executed workload.

Resource capacity of nodes in Million Instruction Per Second (MIPS) is defined as $C = \{c_t^i \mid i \in [1, n], t \in \mathbb{T}, c_t^i \in [0, \omega]\}$ where ω indicates the maximum capacity. Defining an exclusive set for resource capacity allows for heterogeneity considerations. Note that S, R, E, and C represent values per d^i at time slot $t \in \mathbb{T}$.

Application & Workload: An IoT application normally runs multiple microservices. Let $A = \{a^j \mid j \in [1,m]\}$ define a set of microservices. An edge node owns, hosts and runs a full set of microservices in A that are identical across all nodes. The rate at which workload (sensor data/tasks/requests) is generated for microservices at different nodes at different time slots is modelled as $\Lambda = \{\lambda_t^{i,j} \mid i \in [1,n], j \in [1,m], t \in \mathbb{T}, \lambda_t^{i,j} \geq 0\}$. Each task constitutes a certain amount of processing in Million Instructions (MI) to be executed, that is expressed as $M = \{\mu^j \mid j \in [1,m], 0 \leq \mu^j \in \mathbb{R}\}$. The workload modelling is used later on as the basis for the estimation of the number of required function replicas.

Serverless Functions: The unit of resource scheduling in our model is a function in Serverless. We define a set of functions as $F = \{f^{i,j} \mid i \in [1,n], j \in [1,m]\}$ where function $f^{i,j}$ executes tasks of microservice a^j owned by node d^i. Function $f^{i,j}$ requires a certain amount of resources to be deployed on a node. The required resource capacity is defined as $V = \{v^j \mid j \in [1,m], v^j \in [0,\omega]\}$, where v^j is capped at the node's maximum capacity ω.

Function Replicas (instances): Given the enabled auto-scaling in Serverless, $f^{i,j}$ may have a varied number of function replicas at t, capped at $\aleph$, depending on the incoming workload. Hence, set $\Gamma = \{\gamma_t^{i,j} \mid i \in [1,n], j \in [1,m], t \in \mathbb{T}, \gamma_t^{i,j} \in [0,\aleph]\}$ defines the required replica per function by evaluating the corresponding microservice workload over computation capacity. The number of replicas of a^j on d^i at t is measured as $\gamma_t^{i,j} = \min\left(\aleph, \lceil \frac{\lambda_t^{i,j} \times \mu^j}{v^j} \rceil\right)$ that can get a value of 0–$\aleph$ depending on the expected workload. In an optimal offline model with future knowledge, Γ is assumed to be known. If d^i is down at t, then $\gamma_t^{i,j} = 0$ replicas are deployed in the system. In Serverless, zero replica, or so-called scaled to zero functions, consume zero capacity of the node.

For replica-level modelling, we upgrade F to include replica indexes, denoted by $F = \{f_t^{i,j,k} \mid i \in [1,n], j \in [1,m], t \in \mathbb{T}, k \in [1, \gamma_t^{i,j}]\}$ where k indicates the k-th replica.

Function Placement: Functions' replicas can either be placed on their own node locally, or be offloaded to peer edge nodes, upon the scheduler decision. Given SoC s_t^i on d^i at time slot t, the scheduler sets local and foreign function placements per nodes. Denote local placements as $\Psi = \{\psi_t^{i,j} \mid i \in [1,n], j \in [1,m], t \in \mathbb{T}, \psi_t^{i,j} \geq 0\}$ and foreign placements as $\Theta = \{\theta_t^{i,j} \mid i \in [1,n], j \in [1,m], t \in \mathbb{T}, \theta_t^{i,j} \geq 0\}$. This differentiation is particularly essential, as we involve communication overheads in the case of offloading functions for both sender and receiver. That is, if $f_t^{i,j,k}$ is offloaded to d^q, a sending cost for d^i and a receiving cost for d^q is applied. A cost rate per microservice for either one is defined as $O = \{o^{j,h} \mid j \in [1,m], h \in \{send, recv\}, o^{j,h} \in [0,1]\}$, where $o^{j,h}$, $h = send$ or $h = recv$, specify the overhead energy cost imposed by offloading function $f_t^{i,j,k}$ and hosting peers function $f_t^{i,j,k}$, respectively. This cost is mutated into energy usage. Note that local placement $\psi_t^{i,j}$ of replicas of a function is bound to $\gamma_t^{i,j}$, so the constraint $\psi_t^{i,j} \leq \gamma_t^{i,j}$ must be maintained by scheduler.

Node Availability (Up or Down): Nodes generate tasks and run functions only if they are up, i.e., if they satisfy low energy threshold $s_t^i \geq \varphi$. Otherwise, they are technically excluded from hosting any local or foreign function. Node availability (status) is denoted by $X = \{x_t^i \mid i \in [1,n], t \in \mathbb{T}, x_t^i \in \{0,1\}\}$ where $x = 0$ means the node is unavailable, or down, during time slot t, and vice versa. Given that, nodes' binary status x_t^i at t are represented as follows:

$$\forall d^i \in D : x_t^i = \begin{cases} 1 & \text{if } s_t^i \geq \varphi \\ 0 & \text{else} \end{cases} \quad (1)$$

where a node's SoC s_t^i merely depends on the SoC at the previous time slot s_{t-1}^i, renewable energy input r_t^i, and consumed energy e_t^i at time slot t.

$$\forall d^i \in D : s_t^i = \min\left(\vartheta, \max(0, s_{t-1}^i + r_t^i - e_t^i)\right) \quad (2)$$

where SoC can vary between 0 and ϑ. Assuming that renewable inputs is known, the energy consumed e_t^i depends on the hosted functions and for all $d^i \in D$ can be computed as:

$$e_t^i = \left(\frac{c_t^i}{\omega} \times p + \left(\sum_{j=1}^{m} \left(o^{j,send} \times (\gamma_t^{i,j} - \psi_t^{i,j})\right) + \sum_{j=1}^{m} (o^{j,recv} \times \theta_t^{i,j})\right)\right) \times \Delta t \quad (3)$$

which takes into account: (a) *direct usage* and (b) *offloading overhead*. For the *direct usage*, occupied resources are measured by $\frac{c_t^i}{\omega}$ and multiplied by a power consumption rate p. The occupied resources of a node, as a key element in e_t^i measurements, is obtained by $c_t^i = \sum_{j=1}^{m} \left((\psi_t^{i,j} + \theta_t^{i,j}) \times v^i\right)$. For the *offloading overhead*, energy consumption is calculated the send ($o^{j,send}$) and receive ($o^{j,recv}$) overhead multiplied by the offloaded functions ($\gamma_t^{i,j} - \psi_t^{i,j}$) and by the received foreign functions ($\theta_t^{i,j}$), respectively. This overhead measurement is performed for each microservice j. The power consumed by (a) and (b) is multiplied by the length of time slot, Δt, to obtain energy.

B. Problem Formulation

The resource scheduling problem is to dynamically adjust the placement of functions across the cluster to increase nodes' operational availability and reduce availability variance among the nodes, mainly in favor of low-powered and overloaded nodes, we call them bottleneck. This is particularly important due to the fact that if an energy-agnostic scheduler is adopted, such nodes are prone to run out of energy, while well-powered and/or underutilized nodes may waste their energy. The imbalance in power and load distribution between nodes is inevitable in practice [13]. The problem falls in the category of Bottleneck Generalized Assignment Problems (BGAP) [19]. Hence, the objective function (4) maximizes the operational

availability of the edge node with the minimum availability, formulated as:

$$\underbrace{\max \min_{i\in[1,n]} \sum_{t=0}^{T} x_t^i}_{\mathbb{T},D,X,S,R,E,A,\Lambda,M,F,\Gamma,V,\Psi,\Theta,O,C} \tag{4}$$

subject to:

$$\Big(\sum_{i=1}^{n} \psi_t^{i,j} + \theta_t^{i,j}\Big) = \Big(\sum_{i=1}^{n} \gamma_t^{i,j}\Big) \qquad \forall t \in \mathbb{T}, \forall a^j \in A \tag{5}$$

$$\gamma_t^{i,j} = x_t^i \times \min\Big(\aleph, \Big\lceil \frac{\lambda_t^{i,j} \times \mu^j}{v^j} \Big\rceil\Big) \forall t \in \mathbb{T}, \forall d^i \in D, \forall a^j \in A \tag{6}$$

$$\Big(c^i = \sum_{j=1}^{m} \big((\psi_t^{i,j} + \theta_t^{i,j}) \times v^j\big)\Big) \leq (x_t^i \times \omega) \forall t \in \mathbb{T}, \forall d^i \in D \tag{7}$$

At any given t, constraint (5) specifies that the scheduler must not leave any function replica unattended; constraint (6) defines that if a node is down, scheduling of its microservices is not considered, i.e., scale to zero; constraint (7) defines that the sum of the capacity required for placed functions must not exceed the node maximum capacity ω, either local or foreign, and inclusion of x_t^i forces no placement on down nodes.

Solving this problem optimally appears difficult, as it is known NP-Hard [19]. Also, it assumes that renewable energy input and incoming workload at each $t \in \mathbb{T}$, are known to the scheduler in advance, which are difficult to achieve in practice. Hence, in the following section, we propose a greedy algorithm that does not consider such assumptions.

III. Energy-Aware Scheduling

In this section, we propose a greedy algorithm inspired by a bin packing idea [21] that encompasses the following improvements: *(a) Encouraged local placements*: local placement of functions is favorable if possible and if there exists sufficient SoC to reduce the offloading overhead. *(b) Prioritized placements*: we basically form zones of relatively similarly powered nodes and adopt a rigorous prioritizing scheme to handle efficient placements. *(c) Beneficial offloading:* Offloading functions only to a zone with a significantly better SoC. *(d) Warm scheduling*; in cloud scenarios, to avoid cold start of functions, a function is invoked by sending fake requests, known as warm functions [5]. Inspired by that user-driven technique, we introduce a scheduler-driven technique *warm scheduling* that pre-schedules a function of a down node on peers with extra/unused capacity to make functions serviceable immediately, in case its owner node becomes up; in cold scheduling, a function from a recharged node has to remain unscheduled until the next scheduling round. *(e) Sticky offloading:* we introduce *sticky offloading* to mitigate recurring movements of a function over the course of offloading decisions.

The *key added incentive by Serverless* in this approach is that the scheduler is enabled to decide on placements in a function replica level as they are made stateless in Serverless and hence are made independently portable. This facilitates fine-grained resource migrations, the key to precise efficiency. Contrary, conventional monolithic deployments typically fail to achieve this granularity, mainly due to coarse-grained portability. Even general service-oriented deployments require resolved dependency between microservices to be adaptable with such fine-grained scheduling [5], [13]. Another significant incentive is that function replicas are added/removed by embedded auto-scaling in Serverless, dynamically. The scheduler only decides on the placement thereof.

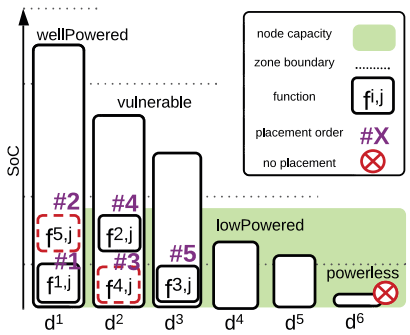

Fig. 3: A demonstration of placements

The details of the improved greedy scheduler is discussed in the following subsections. The scheduler is executed at regular intervals and updates function placements dynamically.

A. Function Scheduling

We first introduce the concept of zones, driven by nodes' SoC, denoted by $Z = \{z_t^i \mid i \in [1,n], t \in \mathbb{T}, z_t^i \in \{\texttt{wellPowered}, \texttt{vulnerable}, \texttt{lowPowered}, \texttt{powerless}\}\}$ where z_t^i stands for node i's zone at t. The entire range of SoC, i.e., $[0,\vartheta]$, is divided into four non-overlapping zones. A node belongs to a zone z_t^i if its SoC s_t^i is within the range belonging to the zone.

Algorithm 1 shows the process of function placements at any given time t. Initially, no function is allocated to any node, node capacities are reset to the maximum capacity ω, nodes' SoC are updated based on their current monitoring, and nodes' zones are assigned accordingly (Lines 1–4). Nodes are sorted in descending order of their SoC. Functions are sorted based on their owner node's zone priority. Zone priorities are ordered from the highest to lowest as follows: (1) `wellPowered`; (2) `lowPowered`; (3) `vulnerable`; and (4) `powerless`. This means functions of a node in the `wellPowered` zone are placed first, which guarantees their local placement. Then functions of nodes in desperate need of power, i.e., `lowPowered` are placed. The functions of `vulnerable` nodes are placed afterwards. Finally, the functions of powerless nodes are placed under warm scheduling. The functions within each zone are sorted by the owner's SoC ascendingly, but the opposite applies for functions in `powerless` zone (Lines 5–6). The reason behind this is that we speculate that nodes in a powerless zone with a higher SoC are more likely to become operational again in the next scheduling round.

Algorithm 1: Energy-Aware Function Scheduling

Data: D, C, S, F, Z, V
Result: F, C
1 $F \leftarrow \{F \mid f_t^{i,j,k} \in F, k = null\}$
2 $C \leftarrow \{C \mid c_t^i \in C, c_t^i = \omega\}$
3 $S \leftarrow \{S \mid s_t^i \in S, s_t^i =$ *current SoC*$\}$
4 $Z \leftarrow \{Z \mid z_t^i \in Z, z_t^i \propto s_t^i\}$
5 **sort** (D descendingly by S)
6 **sort** (F ascendingly by owner's zone priority Z, and within zones ascendingly by owner's S, but `powerless`)
7 **for** $f_t^{i,j,k}$ *in* F **do**
8 $\quad place \leftarrow null$
9 $\quad$ **if** $z_t^i \in \{$`wellPowered`$\}$ **then**
10 $\quad\quad place \leftarrow d_t^i$ ▷ `local placement`
11 $\quad$ **else**
12 $\quad\quad cands \leftarrow \{D \mid d_t^q \in D, (z_t^q \in \{$`wellPowered`$\}) \vee (z_t^q \in \{$`vulnerable`$\} \wedge z_t^i \notin \{$`vulnerable`$\})\}$
13 $\quad\quad place \leftarrow$ Offloading$(D, F, S, C, V, cands, f_t^{i,j,k})$
14 $\quad$ **if** $place = null$ **then**
15 $\quad\quad place \leftarrow d^i$ ▷ `local placement`
16 $\quad f_t^{i,j,k} \leftarrow place$
17 $\quad$ **if** $place = d^i$ **then**
18 $\quad\quad c_t^i \leftarrow c_t^i - v^j$
19 $\quad$ **else**
20 $\quad\quad c_t^q \leftarrow c_t^q - v^j$
21 **return** F, C

Algorithm 2: Function Offloading

Data: $D, F, V, S, Z, cands, f_t^{i,j,k}$
Result: $place$
1 $place \leftarrow null$
`/* if it was offloaded earlier, call sticky */`
2 **if** $f_t^{i,j,k} \neq d^i$ **then**
3 $\quad place \leftarrow$ StickyOffloading$(D, F, S, C, V, cands, f_t^{i,j,k})$
4 **if** $place = null$ **then**
5 $\quad$ **for** $cand^q \in cands$ **do**
$\quad\quad$ `/* verify available capacity on node */`
6 $\quad\quad$ **if** $c_t^q -$ Reservation$(cand^q, v^j) \leq v^j$ **then**
7 $\quad\quad\quad$ ignore $cand^q$
$\quad\quad$ `/* place on the candidate node */`
8 $\quad\quad$ **if** $z_t^i \in \{$`lowPowered, vulnerable`$\}$ **then**
9 $\quad\quad\quad place \leftarrow d^q$
$\quad\quad$ `/* verify warm scheduling */`
10 $\quad\quad$ **if** $z_t^i \in \{$`powerless`$\}$ **then**
11 $\quad\quad\quad$ **if** $z_t^q \in \{$`wellPowered`$\} \vee (\nexists z_t^i \in Z \mid z_t^i \in \{$`wellPowered`$\})$ **then**
12 $\quad\quad\quad\quad place \leftarrow d^q$
13 **return** $place$

The scheduler iterates over the functions replicas and sets their new placements by setting $place$ (Lines 7–20). For a given function $f_t^{i,j,k}$, if it belongs to a `wellPowered` node, it is placed locally (Line 9–10), as for d^1 in Fig. 3. Otherwise, a set of candidate nodes ($cands$) are selected and delivered to the `Offloading` algorithm (more details in Section III-B) to determine the placement (Lines 12–13). Candidates always belong to zones with higher SoCs than that of the function's owner. Also, only `wellPowered` and `vulnerable` nodes can be the placement destinations, as for d^1, d^2, d^3 in Fig. 3, to practise *beneficial offloading*. If the `Offloading` algorithm cannot find a suitable node to offload $f_t^{i,j,k}$, it is placed locally (Lines 14–15), e.g., $f^{3,j}$ in Fig. 3. After the new placement is set for $f_t^{i,j,k}$, its required capacity is deducted from the hosting node's capacity (Lines 17– 20).

B. Function Offloading

Given a list of candidate nodes $cands$, the `offloading` algorithm (Algorithm 2) attempts to determine the best placement for $f_t^{i,j,k}$ on peer nodes. The algorithm first allows for `StickyOffloading` (Lines 2–3). The sticky offloading is a mechanism (more details in Section III-C) that ensures that the placements are *sticky* in the sense that a function offloaded to a node is not offloaded again to a different node in future time slots unless a significantly better hosting node is found. If the `StickyOffloading` attempt is unsuccessful ($place = null$), the offloading algorithm iterates over candidate nodes $cands$ in descending order of their SoC, to choose a placement node (Lines 5–12). To do so, the algorithm first checks whether the available capacity on $cand_t^q$ is sufficient to host the function (Lines 6–7). A `Reservation` module is invoked which calculates the capacity the candidate node $cand_t^q$ needs to reserve for its local functions. This is to *encourage local placements* as it ensures that $cand_t^q$ has reserved sufficient capacity for its local functions. The candidate node is ignored if it does not have enough remaining capacity to host $f_t^{i,j,k}$. This is merely applied to $cand_t^q \in \{$`vulnerable`$\}$ due to its low priority.

If the node has sufficient capacity, a `lowPowered` or `vulnerable` function is placed on the $cand_t^q$ which is always a better node in terms of SoC (Lines 8–9). However, in case of a `powerless` function (Lines 10–12), the offloading algorithm places $f_t^{i,j,k}$ on $cand_t^q$ only if (1) $cand_t^q$ is a `wellPowered` node; or (2) none of the nodes in Z is `wellPowered`. As an example, in Fig. 3, the d^6's function is ignored for warm scheduling on d^3, as the only candidate d^3 with sufficient capacity to host is a `vulnerable` node while `wellPowered` nodes d^1 and d^2 are already filled. This is done because scheduling priority for `powerless` functions is the lowest; therefore, they are highly likely to be placed on the weakest (in terms of SoC) candidate nodes. However, in a future time slot when the owner node of such a function gets enough SoC and possibly becomes `lowPowered`, the

Algorithm 3: Sticky Function Offloading

Data: $D, F, S, C, V, cands, f_t^{i,j,k}, d^q$
Result: returns d^q if offloading should stick to d^q; returns $null$ otherwise

```
/* node's eligibility and available capacity */
```
1 **if** $d^q \notin cands \vee c_t^i - \mathsf{Reservation}(d^q, v^j) \leq v^j$ **then return** $null$

```
/* exception for warm scheduling */
```
2 **if** $(z_t^i \in \{\texttt{powerless}\} \wedge d^q \in \{\texttt{vulnerable}\} \wedge \exists z \in Z \mid z \in \{\texttt{wellPowered}\})$ **then return** $null$

```
/* compare SoC to determine stickiness */
```
3 $d^b \leftarrow \mathsf{BestPlace}(f_t^{i,j,k})$
4 **if** $(s_t^q / s_t^b \geq \texttt{stickiness})$ **then**
5 **return** d^q
6 **return** $null$

function has the highest priority for offloading and is likely to be rescheduled on a `wellPowered` candidate node if it exists. Therefore, the exclusive conditions for `warm` scheduling (Lines 10–12) try to avoid future rescheduling of such functions.

C. Sticky Offloading

As described earlier, sticky offloading ensures that a function $f_t^{i,j,k}$ when offloaded to a node d^q *sticks* to this node in future time slots unless a significantly better placement node is found. Enabled stateless deployments in Serverless and not having to couple to other functions, enable this stickiness. Algorithm 3 shows the details. If d^q is not in the list of candidates anymore or it does not have enough capacity to host the function, the algorithm returns $null$ indicating that the sticky placement of $f_t^{i,j,k}$ on d^q is impossible (Line 1). Also, as stated earlier, if the function belongs to a `powerless` node, it cannot be placed on d^q if d^q is `vulnerable` and there exists some `wellPowered` nodes in Z. Therefore, $null$ is returned similarly in this case (Line 2). Otherwise, the sticky offloading algorithm invokes `BestPlace` module to greedily choose the best possible placement node d^b and compares it against d^q. The `BestPlace` module performs a greedy search similar to the `Offloading` algorithm to find the best node d^b. The algorithm sticks to the current node d^q if SoC of d^q is not significantly smaller than that of d^b, i.e., ratio of s_t^q to s_t^b is at least equal to a stickiness parameter (Line 3–5). Otherwise, the algorithm returns $null$ indicating that a different node should be chosen for the placement.

IV. Implementation

A. Serverless Edge Platform

Main ingredients to implement the modeled system include: (A) an edge platform, (B) powered by variable renewable energy sources, (C) enabled with Serverless computing (D) that runs IoT applications, (E) where nodes are exposed to variable sensor data.

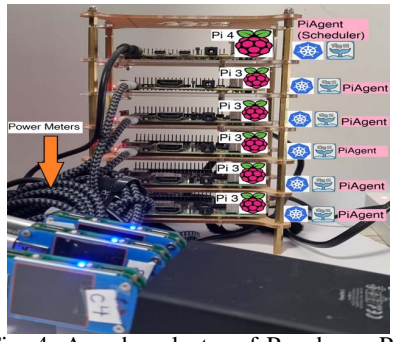

Fig. 4: An edge cluster of Raspberry Pis

Raspberry Pis are employed to build a cluster of edge nodes coexisting in a local network (see Fig. 4). A `PiAgent` module is written in $\sim$ 3700 lines of codes in Python 3.7 using Flask[1] micro-framework, and is open-sourced on GitHub[2]. `PiAgent` realizes the autonomy of nodes and also their collaborations with the scheduler in a parallel and distributed fashion.

Nodes are exposed to different rates of renewable energy sources, i.e., solar irradiation (the controller node is not necessarily battery-operated). Real traces of solar irradiation in a particular time-frame, presenting substantial variations, are injected to the `PiAgent` throughout the experiments. Energy usage on a Pi is measured by a UM25C USB power meter [4].

Serverless platforms are often deployed on top of a container orchestrator such as Kubernetes, Apache Mesos and Docker Swarm. A lightweight Kubernetes, K3s[3], tailored for edge and IoT is used. We use OpenFaaS as the Serverless platform, since it demonstrates higher adaptability to resource constrained edge devices and ARM-based architectures [20].

A Smart Agriculture application [17] is chosen as the use case, as they are supposed to run without an internet connection and be powered by solar panels and batteries in remote and vast farming areas. It is decomposed into three microservices: (1) a smart irrigation management system, (2) a farming monitoring system, and (3) soil moisture monitoring. They receive sensors' data such as humidity, temperature, luminosity, moisture, spatial location and water level. Functions either make a decision such as water connection/disconnection or record the data. In the case of recording, since functions are stateless and ephemeral, Serverless decouples storage systems, so a containerized stateful/serverful in-memory data storage Redis server is also deployed on the controller node.

Sensor data generation, as a HTTP request, is simulated by `PiAgent`. The generated Sensor data requests invoke the corresponding functions.

B. Scheduler Framework

The scheduler resides on the central controller node (see Fig. 1), and its implementation follows the IBM reference framework, MAPE-K [9]: (1) **Monitor**: The `PiAgent` on

[1]https://flask.palletsprojects.com/en/2.0.x/
[2]https://github.com/aslanpour/AdaptiveEdge
[3]https://k3s.io/

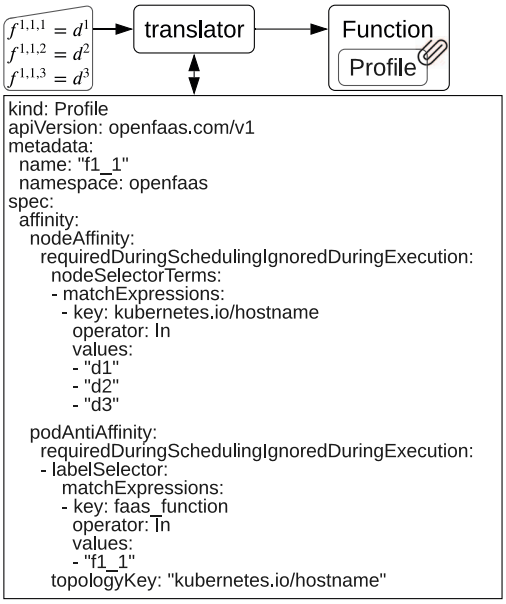

Fig. 5: Translating decisions into Kubernetes terminology

Pis responds to the scheduler upon a request with the latest SoC obtained on the node. (2) **Analyze**: Analysis of zone members according to updated SoCs is performed. (3) **Plan**: The proposed algorithms undertake this step and deliver a list of functions with placement decisions to the executor. (4) **Execute**: Placement decisions are made effective.

Technically, placement executions are carried out by the orchestrator, Kubernetes. Kubernetes allows specifying the placement of (microservice) objects by using terminologies such as *Pod/node affinity/anti-affinity, toleration*, etc., described in Yaml language [22]. Utilizing OpenFaaS, the intended objects are OpenFaaS Functions that must turn into Kubernetes *Deployments*. Hence, Functions cannot directly use the placement terminology. Instead, *Profile* objects are introduced in OpenFaaS to handle that. While placement decisions in our algorithms are per individual replicas of a function, Kubernetes defines replicas by a unified Deployment object. We address this by developing a translator. Fig. 5 demonstrates a sample of developed translation solution for a particular function $f^{1,1}$ that asks for 3 replicas to be placed on d^1, d^2 and d^3. To achieve this, nodes are filtered using *nodeAffinity* and replicas are forced apart by *podAntiAffinity*.

Finally, dynamic scheduling demands CI/CD (continuous integration and continuous deployment). Maintaining a vast number of Yaml files for $n \times m$ deployments is highly unconventional. We automate the entire execution process using Helm[4], an automation tool for Kubernetes deployments, which allows largely conventional deployment descriptions.

[4]https://helm.sh/

V. Performance Evaluation

A. Experimental Setup

We conduct a 24-hour experiment and repeat each experiment 5 times as per employed scheduler. Main settings are as follows:

∘ ***Edge Nodes***: A cluster of 5 wireless Raspberry Pi Model 3 B+ ($n = 5$) is used, and modeled to be powered by solar panels using real traces [1] to set R, shown in Fig. 6. This cluster is orchestrated by Kubernetes K3s (version v1.212) and OpenFaaS (version 8.0.4). A 4-core Pi gives $\omega = 4000$, but we keep a safety net of 10% to avoid overheating. The maximum battery capacity is modeled from the original-size battery of PiJuice for Pi 3 B+ [4] and is set to $\vartheta = 1250$ mWh, and the low energy threshold φ is set to $\frac{\vartheta}{10} = 125$ mWh.

∘ ***Application and Workload***: The Smart Agriculture application runs in three microservices ($m = 3$), executing randomly generated workloads using Poisson distribution with different λ per node and per function, as in [20]. A random seed is set for reproducibility.

∘ ***Functions***: Functions CPU requirements are set as $V = \{400, 300, 100\}$ and allowed maximum replicas of $\aleph = 3 \equiv k \leq 3$. Docker containers of functions are pre-cached on all nodes to eliminate cache interference.

∘ ***Scheduler***: The four non-overlapping zones of `powerless`, `lowPowered`, `vulnerable` and `wellPowered` are set to $(0, 0.1\vartheta]$, $(0.1\vartheta, 0.25\vartheta]$, $(0.25\vartheta, 0.75\vartheta]$, and $(0.75\vartheta, \vartheta]$, respectively. `Stickiness` parameter is set to 0.8, empirically. Scheduler performs every 30 minutes, i.e., $\Delta t = 1800$s.

B. Benchmark algorithms

The proposed energy-aware scheduler, shown as *proposed* in figures, is evaluated against the following benchmarks:

∘ ***Optimal***: This is an offline optimal algorithm which requires the future knowledge of renewable energy input and incoming workload for each time slot and solves the constrained optimisation problem described in Section II modeled in MiniZinc[5] (version 2.5.5) exploiting Gurobi[6] solver (version 9.1.12).

∘ ***Local***: This baseline algorithm always deploys functions locally, i.e., $f_t^{i,j,k} \leftarrow d^i$. This is worth evaluating to understand the impact of offloading.

∘ ***Default***: This is the default performance-aware scheduler in Kubernetes.

∘ ***Random***: This randomly places functions across the cluster.

C. Metrics

∘ ***Operational Availability***: The duration of time a node has been available (i.e., not in powerless state) to the total time. This is to represent the objective achievement, as in (4), by evaluating the degree at which the minimum operational availability of bottleneck nodes is maximized.

∘ ***Function Replacements***: The total number of times the host of functions of a node are changed (i.e., # of replacements).

∘ ***Throughput***: The total number of tasks executed per second.

[5]https://www.minizinc.org/

[6]https://www.gurobi.com/

○ ***Response Time***: The amount of time it takes from when the sensor data is generated to when the final output of the function execution is returned to the device.

D. Results

1) Operational Availability: Fig. 7 demonstrates that the *optimal* and *proposed* algorithm maximize the operational availability of the bottleneck node (node 5) by up to 33% over the *default* Kubernetes scheduler, and even further over *local* and *random* (see differences between green columns). Another achievement is the even energy consumption of cluster's nodes despite uneven distribution, which is measured by the standard deviation (SD) of the operational availability of the cluster nodes; the smaller, the better. In this regard, the line chart in Fig. 7 shows that the *optimal* (5.9) and *proposed* (7.16) algorithms improve the SD over *default* Kubernetes scheduler (11.95) by 101%, and once again, more over others. Another indication of the cluster-wide improvement is viewed in the smaller difference, i.e., gap, between the operational availability of the maximum and minimum available nodes (shown by numbers in rectangles in Fig. 7). Fig. 7 shows the *optimal* (16) and *proposed* (20) algorithms narrow the gap over the *default* (33) Kubernetes scheduler by up to 103%.

Why do other algorithms fail to satisfy the objective? The *local* and *default* perform relatively alike such that they do not utilize dynamic resource scheduling and any node exhausts its own energy in isolation. Both, with different rates, represent Node 2 as the longest available node. This is because, in such algorithms, Node 2 hosted its own functions locally and had a lower workload generation than Node 1; given that, while other nodes suffered from energy shortage, Node 2 refused to collaborate and wasted its excess energy. Besides, the *default* algorithm is allowed to utilize the resource sharing, but it is a performance-aware algorithm and no energy consideration is involved in its one-off placement. Thus, this algorithm performs the best only if the highest-performance node is receiving the highest energy input which is not always the case. Contrary, the *random* algorithm is allowed to offload but imposes many recurring replacements which disallows operational availability improvements, explained in the followings.

2) Function Replacement: Fig 8 shows the number of replacements per scheduler. While *local* and *default* are not benefiting from a dynamic mechanism, the *proposed* algorithm (#57), approaching the *optimal* (#47), can significantly reduce replacements over the *random* (#427) algorithm. The *proposed* algorithm achieves this by virtue of effectively restricted offloading decisions applied by *beneficial offloading* and *sticky offloading* strategies. Such strategies helped the *proposed* algorithm performs consistency.

3) Throughput: Fig. 9 evidences that the system's throughput relatively increases in line with the improvement in cluster's operational availability, where *optimal* (0.231) and *proposed* (0.222) algorithms enable the cluster nodes to execute up to 15% more tasks, compared to the *default* (0.198) Kubernetes scheduler. Another key insight is that although benefiting the resource sharing tends to compromise well-powered nodes' energy, it does not reduce the overall system's throughput when handled properly. That is, effective resource sharing—identifying well-powered and low-powered nodes and using excessive energy of one for another, practiced by the *proposed* algorithm—increases the throughput. The opposite effect of resource sharing on throughput occurs for the *random* (0.185) algorithm due to unplanned offloading decisions.

4) Response Time: Response time represents the impact of schedulers on QoS. That is, the round trip of sensor/actuation is expected not to be significantly affected for the sake of energy management. Fig. 10 confirms that the *proposed* algorithm gives comparative figures, way shorter than that of the *random* algorithm, preserving the QoS. In other words, in terms of the average response time and tail of latency (95^{th} percentile), the *proposed* algorithm exhibits a narrow margin of difference to that of the *local* and *default* algorithms. The *local* scheduler shows the shortest response times, since it removes the need for remote access to the sensor data imposed by offloading in other algorithms.

E. Impact of Scheduling Intervals

Fig 11 demonstrates that varied scheduling intervals, i.e., short- (15 min.), mid- (30 and 60 min.) and long-term (180 min.), cannot influence the *proposed* algorithm, and its superiority over comparative ones, in maximizing operational availability of the bottleneck node. Precisely, mid-term intervals appeared of the most appropriate ranges. However, short-term intervals seem excessively interferential, particularly for the *random* algorithm with the least availability of bottleneck nodes. Long-term intervals appear slightly less efficient for energy-aware algorithms, i.e., *optimal* and *proposed*, due to delayed actions while *random* moderately benefits.

F. Contributions of Sticky Offloading and Warm Scheduling

How do sticky offloading and warm scheduling affect the performance of energy-aware scheduling?. Fig. 12 provides a detailed analysis of the *proposed* scheduler contributions in a sample experiment, taken in 48 rounds (every 30 minutes). scheduling rounds with no actions are excluded from the beginning and at the end. Also, only replacements are highlighted and locally placed functions are not mentioned.

Chronologically, over the first 12 scheduling rounds, nodes are powerless and functions are considered locally. After that (the starting point in Fig. 12), nodes perceive SoC increase and while Node 1 and 2 obtain enough energy for locally hosting their functions, the warm scheduling policy pre-schedules `powerless` functions on Node 1 and 2 during rounds 14–15. This allowed Node 3, 4 and 5 serving sensors data straight away after joining `lowPowered` zone in rounds 15, without having to wait until the next scheduling round. After a couple of replacements during rounds 15–16, the first sticky offloading was applied to functions of Node 4 in round 16–18, indicated by yellow oval. As Node 4 later on is powered sufficiently and joins the `wellPowered` zone, the scheduler moves its functions back locally by round 20. Given the

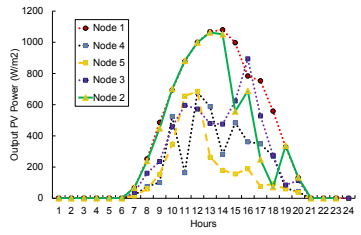

Fig. 6: Solar energy input to nodes

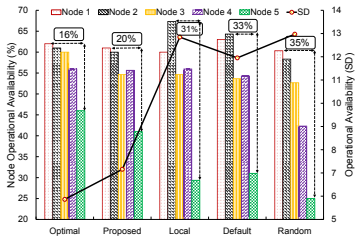

Fig. 7: Operationally per scheduler.

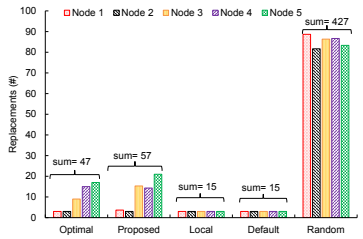

Fig. 8: Replacements per scheduler

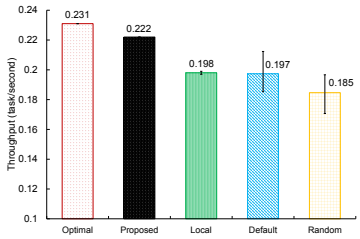

Fig. 9: Throughput per scheduler

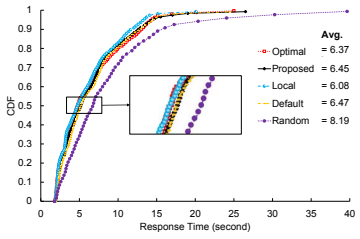

Fig. 10: Response time per scheduler

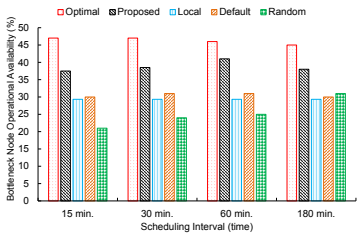

Fig. 11: Impact of scheduling interval

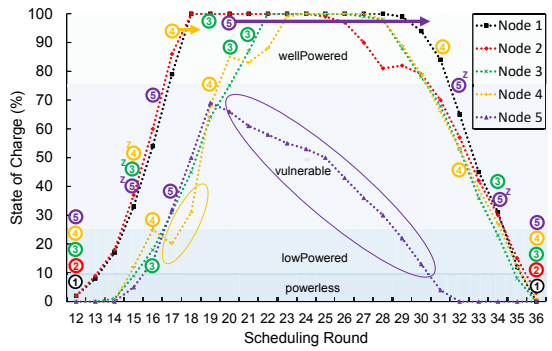

Fig. 12: Deep analysis of the proposed scheduler's performance. Circled numbers, z and $\rightarrow$, respectively, mean the functions of Node X, zero scaled functions by warm scheduling and sticky offloading duration.

unstable situation for Node 3 in round 21–23, its functions are relocated two times until it joins the `wellPowered` zone. Instead, Node 5 fails to join the `wellPowered` zone and keeps loosing energy. Recognizing this, its functions are offloaded to Node 1 and 2 by `Offloading` in round 21 and supported by `StickyOffloading` remains untouched up to round 31, despite the attraction offers of Node 3 and 4. Node 5 cannot survive beyond round 31, despite offloading, and leaves `vulnerable` and then `lowPowered` zones (indicated by a purple oval). Particular attempts for warm scheduling thereof are taken in round 33 and 35, but no energy input is provided. During round 17–31 some minor offloading actions are also taken for other nodes.

By approaching the evening, all nodes exhaust their energy and become powerless. Note that functions of Node 1 and 2 efficiently kept working locally throughout the experiment with zero replacements. Such consistency and efficient stickiness resulted in less intervention of the scheduler, thus less power wastage, higher serviceability and throughput and sufficient preservation of QoS.

G. Threats To Validity

Reproducibility: Experiments are repeated 5 times for consistency and the prototype platform is open-sourced for reusability and extensions. *Real-world and large-scale scenarios*: Given real-world experiments, it will effectively operate in practise. Also, scalability, inside the communication range, is achievable thank to the centralized design of extreme edge, with a computationally powerful enough controller node. Many centralized use cases such as Industry 4.0, Wearable Devices, Augmented Reality and Virtual Reality, as well as the smart farming demonstrated in this paper can benefit.

VI. Related Work

To the best of our knowledge, there is no existing work that studies the energy-aware scheduling at Serverless edge computing, due to its originality. Further there are even a few general schedulers for Serverless edge computing. Hence, we broadly review scheduling efforts on Serverless in cloud computing, regardless of their goal.

In a cloud-based attempt [3], authors propose a scheduler for cache hit ratio improvements and evaluated it using simulations. They believe that large packages, e.g., large function images, could delay functions' instantiating. In a similar effort [6], another package-aware scheduler (PASch) is proposed and is evaluated empirically. However, migrating from cloud that handles millions of packages, to the edge, specifically extreme edge with only a few microservices, it appears not as emergent as energy challenges. We eliminated container caching effects by pre-caching.

In a cloud to edge continuum [10], task placement solutions are proposed to optimize latency and cost by adopting Pi 3 B+ as edge nodes, as we adopted. A similar approach, but for function placements, is proposed as ActionWhisk [8] that aims at cost minimization, not energy. In another latency-focused study [20], moving both task and function in a cluster of heterogeneous nodes, including Pis, is studied.

Particular non-Serverless attempts are also worth mentioning. A node grouping strategy, akin to zones, is followed in [11] for handling energy variability in the cloud through a prediction-based task scheduling. Regardless of Serverless inconsideration, resource-intensive predictions at the edge on computation-constraint devices will not be always a viable option. The other non-Serverless attempt for energy efficiency is made by [21] where they similar to us recognized the significance of computation than communication and targeted a fair energy distribution through scheduling. In [2], a distributed controller, e.g., orchestrator, is encouraged for energy efficiency while we employed a central one and we acknowledge that the efficiency of only edge nodes has been our aim. Finally, a rather identical edge environment powered by renewable energy is considered in [14], for task scheduling which performs beneficial for particular use cases, but for privacy-critical ones requires further investigations.

VII. Conclusions

This paper investigated the challenges of reinforcing energy awareness at Serverless edge computing through resource scheduling. To the best of our knowledge, we are the first to consider such. We proposed resource scheduling algorithms to place functions on edge nodes powered with battery and renewable energy sources such as solar owing to their energy availability. Our proposed algorithms aim to maximize the operational availability of edge nodes while minimizing the variation thereof without compromising the throughput and QoS. We proposed innovative techniques such as *sticky offloading* and *warm scheduling* to reduce recurrent function replacements. We evaluated the proposed algorithms in a real test-bed, valuable insights were obtained that confirm the effectiveness of the proposed scheduler. Results show that our proposed energy-aware scheduler can improve operational availability, throughput, and serviceability over the benchmark algorithms and close to optimal while maintaining QoS, i.e., response time, acceptable.

As future work, we cover a broader range of Serverless edge systems, including heterogeneous edge nodes. Not only CPU but also GPU and FPGA will be involved with both wireless and wired communications. In addition to CPU-intensive IoT applications used in this research, we evaluate I/O-, memory, and data-intensive applications. We will also consider simulations to assess the scalability challenges of the proposed approach. Furthermore, to cover the mobile edge computing area, decentralization of the scheduler is essential.

References

[1] "Bureau of Meteorology." [Online]. Available: http://www.bom.gov.au/climate/how/ newproducts/IDCJAD0111.shtml

[2] E. Ahvar, A. C. Orgerie, and A. Lebre, "Estimating Energy Consumption of Cloud, Fog and Edge Computing Infrastructures," *IEEE Trans. on Sustain. Comput.*, vol. 3782, no. c, 2020.

[3] P. Andreades, K. Clark, P. M. Watts, and G. Zervas, "Experimental demonstration of an ultra-low latency control plane for optical packet switching in data center networks," *Optical Switching and Networking*, vol. 32, pp. 51–60, 2019.

[4] M. S. Aslanpour, A. N. Toosi, R. Gaire, and M. A. Cheema, "WattEdge: A Holistic Approach for Empirical Energy Measurements in Edge Computing," in *International Conference on Service-Oriented Computing*. Springer, 2021, pp. 531–547.

[5] M. Aslanpour, A. Toosi, C. Cicconetti, B. Javadi, P. Sbarski, D. Taibi, M. Assuncao, S. Gill, R. Gaire, and S. Dustdar, "Serverless Edge Computing: Vision and Challenges," in *AusPDC*, 2021.

[6] G. Aumala, E. Boza, L. Ortiz-Aviles, G. Totoy, and C. Abad, "Beyond load balancing: Package-aware scheduling for serverless platforms," *CCGRID*, pp. 282–291, 2019.

[7] P. J. Basford, S. J. Johnston, C. S. Perkins, T. Garnock-Jones, F. P. Tso, D. Pezaros, R. D. Mullins, E. Yoneki, J. Singer, and S. J. Cox, "Performance analysis of single board computer clusters," *FUTURE. GENER. COMP. SY.*, vol. 102, pp. 278–291, 2020.

[8] D. Bermbach, J. Bader, J. Hasenburg, T. Pfandzelter, and L. Thamsen, "AuctionWhisk: Using an Auction-Inspired Approach for Function Placement in Serverless Fog Platforms," pp. 1–48, 2021.

[9] A. Computing, "An architectural blueprint for autonomic computing," *IBM White Paper*, vol. 31, 2006.

[10] A. Das, S. Imai, S. Patterson, and M. P. Wittie, "Performance Optimization for Edge-Cloud Serverless Platforms via Dynamic Task Placement," *CCGRID*, no. 1, pp. 41–50, 2020.

[11] J. Gao, H. Wang, and H. Shen, "Smartly Handling Renewable Energy Instability in Supporting A Cloud Datacenter," *IPDPS*, pp. 769–778, 2020.

[12] Y. Han, S. Shen, X. Wang, S. Wang, and V. C. Leung, "Tailored learning-based scheduling for kubernetes-oriented edge-cloud system," *IEEE INFOCOM*, vol. 2021-May, 2021.

[13] C. Jiang, T. Fan, H. Gao, W. Shi, L. Liu, C. Cérin, and J. Wan, "Energy aware edge computing: A survey," *Computer Communications*, vol. 151, no. 2018, pp. 556–580, 2020.

[14] A. Karimiafshar, M. R. Hashemi, M. R. Heidarpour, and A. N. Toosi, "Effective Utilization of Renewable Energy Sources in Fog Computing Environment via Frequency and Modulation Level Scaling," *IEEE Internet Things J.*, vol. 7, no. 11, pp. 10 912–10 921, 2020.

[15] W. Li, T. Yang, F. C. Delicato, P. F. Pires, Z. Tari, S. U. Khan, and A. Y. Zomaya, "On Enabling Sustainable Edge Computing with Renewable Energy Resources," *IEEE Communications Magazine*, vol. 56, no. 5, pp. 94–101, 2018.

[16] Q. Luo, S. Hu, C. Li, G. Li, and W. Shi, "Resource Scheduling in Edge Computing: A Survey," *IIEEE Commun. Surveys Tuts.*, vol. 48202, no. c, pp. 1–36, 2021.

[17] T. Ojha, S. Misra, and N. S. Raghuwanshi, "Internet of Things for Agricultural Applications: The State-of-the-art," *IEEE Internet Things J.*, p. 1, 2021.

[18] P. Patros, J. Spillner, A. V. Papadopoulos, B. Varghese, O. Rana, and S. Dustdar, "Toward sustainable serverless computing," *IEEE Internet Computing*, vol. 25, no. 6, pp. 42–50, 2021.

[19] D. W. Pentico, "Assignment problems: A golden anniversary survey," *Eur. J. Oper. Res.*, vol. 176, no. 2, pp. 774–793, 2007.

[20] T. Rausch, A. Rashed, and S. Dustdar, "Optimized container scheduling for data-intensive serverless edge computing," *Future Generation Computer Systems*, vol. 114, pp. 259–271, 2021.

[21] C. Wang, X. Wei, and P. Zhou, "Optimize Scheduling of Federated Learning on Battery-powered Mobile Devices," *IPDPS*, pp. 212–221, 2020.

[22] L. Wojciechowski, K. Opasiak, J. Latusek, M. Wereski, V. Morales, T. Kim, and M. Hong, "NetMARKS: Network metrics-AwaRe kubernetes scheduler powered by service mesh," *IEEE INFOCOM*, vol. 2021-May, 2021.

[23] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue, "All one needs to know about fog computing and related edge computing paradigms: A complete survey," *Journal of Systems Architecture*, vol. 98, pp. 289–330, 2019.