Energy Optimization through a Multidimensional Distributed Scheduling Approach

Valera Humberto, Giraldo Leonel Isaac Guerrero, Muñoz Carlos Alejandro Sivira, Marroquin Alan Alfredo Rojas, Damas Lucas Aguilar, Marc Dalmau, Philippe Roose, Cardinale Yudith

Energy Optimization through a Multidimensional Distributed Scheduling Approach

Humberto Valera*, Leonel Isaac Guerrero Giraldo†, Carlos Alejandro Sivira Muñoz†,
Alan Alfredo Rojas Marroquin‡, Lucas Aguilar Damas§, Marc Dalmau¶,
Philippe Roose¶, and Yudith Cardinale§
*Technopôle DOMOLANDES, Saint-Geours-de-Maremne, France †Universidad Simón Bolivar, Caracas, Venezuela
‡Universidad Católica San Pablo, Arequipa, Peru §Universidad Internacional de Valencia, Spain
¶E2S UPPA, University of Pau, Anglet, France
Emails: humberto.valera@domolandes.fr, {leonelisaacguerrero, carlos.csivira}@gmail.com, alan.rojas@ucsp.edu.pe,
{laguilard, ycardinale}@universidadviu.com, {Marc.Dalmau, Philippe.Roose}@iutbayonne.univ-pau.fr

***Abstract*—Distributed systems, spanning pervasive, cloud, and edge computing, require robust scheduling for tasks and data management, facing challenges like hardware diversity, QoS, localization, and energy efficiency. Current solutions often lack a comprehensive approach, failing to achieve balanced load management, QoS, and notably in minimizing application energy use, a key modern concern. We propose a distributed scheduler using multidimensional spaces and data structures aimed at comprehensive load balancing, energy efficiency, and QoS. The scheduler indexes devices within the structure based on diverse criteria such as hardware availability, GPS, and energy features. It then organizes applications into abstract graphs, where nodes are containers and their data items (units of data accessed by the containers), and edges are the transfer rates among them. To migrate containers, the scheduler performs range queries within the structure for optimal devices. To organize data items, it finds the barycenter of the devices executing the containers linked to them. We evaluated our scheduler's effectiveness using the PISCO simulator, comparing it against other energy-efficient schedulers.**

***Index Terms*—distributed systems, power, energy, scheduling**

I. Introduction

Modern distributed schedulers in cloud and edge environments orchestrate resource allocation to ensure consistent Quality of Service (QoS) as defined by Service Level Agreements. They adapt to changing workloads and device failures using middleware and virtualization (VM, containers) to enable task migration and duplication, improving service continuity and resilience [1]–[3].

The growing complexity of these infrastructures highlights new critical goals in scheduling operations [4]. These include the physical location of devices and the nodes where data elements are deployed (e.g., volumes in Kubernetes or HDFS in Hadoop), along with energy consumption considerations. The latter is vital for both economy and sustainability. To achieve the Zero-Emissions goal by 2050, electricity emissions must be reduced by 55% by 2030, yet computing devices' energy use is increasing 7% faster than the global average[1].

To manage multiple complex variables and reduce energy consumption, we introduce an online dynamic scheduling method based on a previously proposed approach existing in the literature [5]. It implements an abstract hardware resource space, indexing devices in a distributed multidimensional data structure by their hardware availabilities. The scheduler enables a 'query-to-migrate' process, properly identifying the most energy-efficient nodes for containers execution. The latter are profiled considering CPU, hard disk, and network usage as primary variables influencing energy consumption, while RAM usage is treated as a static factor, not impacting energy usage. Moreover, this method only considers containers without considering the transfer of information between them nor the data items they utilize.

We expand this approach to include connection entities (container-to-container and container-to-data package interactions) and data items (like database files and raw files) as schedulable elements. Applications are thus abstract graphs, with nodes being containerized processes and data items, and edges being average transfer rates among them. Furthermore, we broaden the scope of energy variables to also encompass RAM activity, GPU load, and the physical positioning of devices as energy-related factors. Our scheduler performs region queries for energy-efficient hardware resources to organize the containers. Then, to schedule data items, it considers the barycenter of the subspace of hosts executing the containers linked to these data items. For this, we introduce the corresponding energy-distance functions.

The approach is abstract and applicable across various orchestration technologies, enabling the optimization of granular hardware resource usage and energy consumption, among other variables, with optimal computational complexity. While this heuristic is inherently technology-agnostic, it is implementable in orchestration systems such as Kubernetes (first implementation steps[2]). However, in this article, we demonstrate our approach's energy and QoS effectiveness through a simulation environment, comparing it versus methods based on hardware availability, nodes' profiles, and historical execution data.

The rest of this article is structured as follows: Section II

[1]https://moderndiplomacy.eu/2022/01/17/surging-electricity-demand-is-putting-power-systems-under-strain-around-the-world/

[2]https://github.com/Humberto-Valera/Distributed-Energy-Scheduling-for-K8s.git

explores related work; Section III explains data structures in schedulers; Section IV outlines our multidimensional space implementation; Section V introduces our scheduler engine and algorithm; Section VI presents experiments and results validating our method's effectiveness; and Section VII concludes with future directions.

II. Related Work

For cloud setups, scheduling approaches address VM overload, energy, cost efficiency, tasks' timing constraints, and adaptive learning strategies [6]–[13]. They improve energy efficiency through VM consolidation, noting that while the majority focus primarily on CPU energy consumption, only a few also consider memory, and even fewer account for bandwidth [14]. For containers, some efforts focus on redeploying POD within Kubernetes environments for hardware and energy improvement to scale down active nodes [15].

In the Edge, some initiatives optimize energy and bandwidth by clustering nodes and profiling tasks [16]. They also focus on migrating containers in IoT and P2P networks to save energy, employing k-means and hierarchical clustering algorithms for identifying efficient hosts [17], or utilizing simple negotiation among peer nodes for energy-aware container migrations [18], [19].

While all these strategies improve energy efficiency, often through predictive models, they may not be compatible with high workload fluctuations and overlook the full range of hardware necessary for understanding energy consumption. In contrast, our abstract approach holistically addresses three crucial aspects: **energy efficiency, dynamic adaptability, and hardware diversity**. For this purpose, our approach leverages multidimensional data structures, enabling efficient resource indexing, rapid task-resource matching, and predictable updates during node/application load changes. This technique can be implemented in different infrastructures, such as K8s[3].

III. Data structures as a scheduler kernel

Our scheduler relies on a multidimensional space and structure for energy savings and QoS optimization. It manages factors such as hardware resources and the physical location of devices, enabling complex queries for precise load balancing. Such a space is defined in Def. 1.

Definition 1 (The Multidimensional Space): The space U is made up of 7 dimensions/axis: 1) CPU computational capacity in GHz; 2) RAM capacity expressed in MB; 3) Network transfer rate expressed in MB/s; 4) Storage speed expressed in MB/s; 5) GPU computational capacity expressed in GHz, 6) GPS position of devices, and 7) Power consumption of the network connection in watts. Given a set of devices connected to a shared network, they are indexed in the space U in terms of their current average hardware resources' availability, GPS position, and connection-based power consumption.

Given the heterogeneous characteristics of space U, a custom algebra for query operations is essential [20]. We adopt the methods described by Gaede et al. [20] as follows:

[3]https://github.com/hhumbertoAv/DistributedEnergyScheduling-for-K8s

- Custom Region Query: This query comprises four intervals – $[CPU_i, CPU_f]$, $[RAM_i, RAM_f]$, $[NET_i, NET_f]$, $[I/O_i, I/O_f]$ - and identifies all devices D that fall within these intervals.
- Custom Nearest Neighbor Query(n): Given a device D with specific network-related power consumption, this query identifies the nearest n devices with the shortest distance to D. This distance calculation incorporates CPU and RAM usage during data transmission over the network as Eq. 1 shows.

$$\begin{aligned} &\text{Power}(Dev)_{\text{net}} = \text{power}(CPU)_{\text{net}} + \text{power}(RAM)_{\text{net}} + \\ &\text{power}(NET)_{\text{net}}; \\ &\text{dis}(Dev(A), Dev(B)) = \text{Power}(A)_{\text{net}} + \text{Power}(B)_{\text{net}} \end{aligned} \tag{1}$$

IV. The Data Structure and the U Space Implementation

Several approaches deploy distributed structures like advanced R-trees [21] and KD-trees [22] for network management. While a detailed study of these structures exceeds this paper's scope, we adopt a non-centralized approach [5] for its proven energy efficiency in scheduling and its avoidance of centralized failures. Thereby, we implement our space as a non-centralized structure based on the MAAN (Multi-Attribute Addressable Network) overlay [23]. MAAN generalizes the Chord approach [24] to deploy a multidimensional space in as many circular device-sorted lists as the space has dimensions. That enables multidimensional queries and element insertion/deletion in an efficient logarithmic complexity.

In the Chord protocol, devices are organized into an ordered circular list, arranged clockwise based on their ID. Each device maintains a 'Finger table' comprising network addresses, typically IP addresses or unique identifiers of other peers. The table's size equals the bit string length for the system's maximum ID. Each entry in this table is calculated as node ID$+2^i$, where i ranges from 0 to the bit string length minus one. This setup enables efficient routing in the network's circular ID space. When a node N searches for an element E, it utilizes its Finger table to find the successor of E's ID, streamlining the search process in the distributed network.

A. The node insertion and deletion operation in MAAN

In the Chord protocol, a new node nN joins the network by first obtaining a unique ID through hashing its IP address. It then connects to an existing node nN_1, identified as its immediate successor in the network. nN copies the finger table from nN_1 and sets its predecessor to nN_1's predecessor. Subsequently, nN_1's predecessor is updated to nN. Finally, nN proceeds to update the finger tables of nodes within a specific range. This range typically includes the $2^n - 1$ (where n is the finger's table size) closest preceding nodes in the network. These updates are crucial for maintaining network consistency and ensuring efficient routing, as nN integrates itself into the existing network structure.

When a node lN needs to leave the network, it first notifies its predecessor node plN to consider lN's immediate

successor, lN_1, as the new direct successor. Then, lN_1's predecessor reference is updated to plN. Following this, an updating process similar to the join operation is carried out, but this time with lN_1's ID. This update ensures that the finger tables of nodes counter-clockwise in the circle remain current. Finally, the responsibility for the keys (data in the node) managed by lN is transferred to lN_1.

MAAN extends the Chord protocol's approach for adding or removing devices, adapting it to a multidimensional context. This involves applying similar steps as Chord for node updates, but across multiple dimensions, ensuring each list remains coherent and accurately reflects the system's structure.

B. Implementing the space U

To index and operate devices within the resource space U, our system implements the MAAN operations but, rather than relying on IP addresses for ID generation, it uses metrics based on the hardware resource availability and network-related power consumption of devices. We employ a specially designed locality-preserving formula to convert a numerical value V (availability of a device's resource or its network connection's power consumption) into an ID that fits coherently within our multidimensional space. As shown in Eq. 2, this process involves scaling V by the maximum number of nodes N our system can support at any given time, and then normalizing this product by the highest limit value SL for device resources or consumption. Values of SL and N are tailored to the specific constraints of each dimension, ensuring that the generated ID accurately represents the resource or power profiles of the devices in each dimension.

$$ID = \left\lceil \frac{V \times N}{SL} \right\rceil \tag{2}$$

When a new node acquires an existing value within a specific dimension, it becomes part of a $twin-list$ implemented for each device. All $twin-nodes$ share the same predecessor and successor pointers within the overlay network.

C. Information frames and the reindexing process

Our structure indexes devices by the dimensions of the space U, whose values can fluctuate significantly due to OS scheduling or device movement. To manage this, the middleware on each device implements '*data frames*', FIFO-like collections for each dimension. They periodically store information, average this data, and then apply Eq. 2 to determine the ID for each device in every dimension. This method enables controlled reindexing of nodes within the system.

D. The querying process implementation

Our system uses MAAN's iterative method for queries, starting with one dimension to create a candidate list X, and then refining it by considering other dimensions. To prevent resource conflicts—when multiple queries select the same node, causing uncoordinated actions—we introduce a "candidate lock" so each device in X is locked to one query at a time. Devices remove locked candidates from X. We also tackle "circular saturation," where device overload from migrations can strain the system, by predicting resource and power needs before migration/duplication. These predictions inform our scheduling algorithm to maintain system stability. They are detailed in the next section.

V. The scheduler

Our energy-aware scheduling method utilizes multidimensional spaces to allocate hardware resources efficiently, allowing for future customizations in various data structures, frameworks, and optimization dimensions. This paper presents a general system configuration for our scheduler implementation, outlined as follows: (i) all devices are indexed within space U; (ii) a device D may run multiple containerized applications $[C_i \dots C_j]$, each with initial requirements for CPU, RAM, NIC, GPU, and storage rate; (iii) containerized applications, identifiable by PIDs, have their energy usage profiled by equations with values sourced from GNU/Linux interfaces; (iv) each container can connect to others at an average transfer rate determined by an *information frame*; (v) D stores data units, each identified by an ID and described by its size in MB; (vi) containers access data units across the network at an average rate set by an *information frame*; (vii) a composite application comprises a set of containerized applications and data units linked in an incomplete graph; (viii) the OS on each device runs a single-priority round-robin scheduling algorithm, ensuring that in overload scenarios, competing processes proportionally receive resources based on their requests.

Upon joining the space U, a device's supervisor Microservice **(SM)** independently launches a scheduling algorithm instance, potentially enabling devices to operate under different configurations of the scheduling parameters. The **SM** is a daemon-based middleware deployed in each device with three main objectives: (i) maintain an updated container list that can be sorted by each axis of the space U; (ii) track each device's component load information; and (iii) implement the technical overlay aspects (ID, finger tables, and the information frame) and the scheduling mechanisms, such as the mentioned projection operations. Let's first explore the latter mechanisms before delving into scheduling details.

A. Power calculation and projection

Power and energy metrics for hardware components in space U, derived from equations using pre-computed values, OS interfaces, or datasheets have been validated for changes in power consumption relative to overall computer consumption [4]. While these approximations may oversimplify hardware specifics, typical of software power tools, they align with our goals to track power variations across components and predict consumption before deployment or migration. This is why we do not use tools like PowerJoular [25] or Scaphandre[4], which are effective only for single components and do not support power-prediction approaches.

[4]https://github.com/hubblo-org/scaphandre

*1) **CPU energy estimation.**:* Power and energy consumption across CPU cores for a process is estimated by Eq. 3, which incorporates CPU capacitance (calculated from TDP, V_{TDP}, and f_{TDP} with a 0.7 multiplier [26]), current frequency (f_i), and voltage (V_i) [26], the ratio of process-specific CPU usage to total CPU usage ($U^{PID}_{CPU_i}/U_{CPU_i}$), and CPU fan energy ($fans_s$) that adjusts based on core frequency thresholds ($f_i > f_x > f_j$).

$$\sum_{i=0}^{nCores} E^{PID}_{CPU_i}(t) = \frac{0.7 \times TDP}{f_{TDP} \times V^2_{TDP}} \times f_i \times V_i^2 \times \frac{U^{PID}_{CPU_i}}{U_{CPU_i}}(t) + fans_s(t), where : \quad fans_s = J \forall K(f_i > f_x > f_j) \tag{3}$$

a) Power projection/prediction:: To estimate the power consumption of a container C before its migration from device D to $D2$, we follow the steps in Eq. 4.

Given:

1. $I_{C_D} = \frac{X_{C_D} \times I_D}{100}$, where I_D is the number of instructions D can execute in time $T1$, and C uses $X\%$ of D's CPU time X_{C_D} in $T1$; AND
2. $X_{C_{D2}} = \frac{X_{C_D} \times I_D}{I_{D2}}$, estimating C's CPU load percentage on $D2$, from it's CPU's instruction capacity; THEN
3. $E_{C_{D2}} = \frac{X_{C_{D2}} \times I_D \times C_{D2} \times V^2_{D2} \times F_{D2}}{100 \times I_{D2}} + fans_s(t)$, is the energy $D2$'s CPU would consumes processing C.

(4)

*2) **RAM energy estimation.**:* For RAM energy, Eq. 5 [27] covers both background (E_{BK}) and active process energy ($E_{act_{app}}$) across its different working states and its operation quantities.

The RAM energy consumption combines:

1. Background energy, E_{BK}, accounts for:
   $E_{BK} = T_{sf} \times 0.35\,\text{W}$ (self refresh) $+ T_{ckeoff} \times 0.89W$ (CKEOFF state) $+ T_{act} \times 1.56\,\text{W}$ (active state) $+ \sum_{i \in \{ranks\}} T_{act} \times 0.098\,\text{W}$ (per rank),
   where T represents time in each state,
2. Active energy, E_{act}, for read/write operations:
   $E_{act} = N_R \times 6.6\,\text{nJ}$ (page read) $+ N_W \times 8.7\,\text{nJ}$ (page write),
   where N_R and N_W are the number of read and write operations.

(5)

a) Power projection/prediction:: To project consumption from device D to D_2, we adapt the multipliers of Eq. 5 for D_2, considering RAM-module differences based on datasheet specifications.

*3) **Network interfaces, storage devices, and GPU estimation.**:* For these components, we provide a model that considers the load generated by a process P (L_P/L_{MAX_D}) and the device's active power consumption (W_{u_D}) (see Eq. 6).

$$E_{D_M} = (W_{u_D} \times \frac{L_P}{L_{MAX_D}}) \times T_D \tag{6}$$

a) Power projection/prediction:: We use Eq. 6 for projecting power consumption to a device $D2$, adjusting PC and L_{MAX_D} for $D2$'s load by a process P ($L_P/L_{MAX_{D2}}$).

B. The communication protocol

Each **SM** in our system uses a descriptor vector in peer negotiations, detailing six metadata types per device: resource capacity, current hardware load, resource availability, container C_i execution requirements, load average by C_i, and operation type (request/answer). This ensures a cost-effective communication method for scheduling (see Algorithm 2, line 1).

C. The scheduling algorithm

Upon joining, devices' **SM** run a scheduling algorithm (Algorithm 1) considering three main scenarios:

In the first scenario (line 6 to line 13), if a device's component load hits an overload condition (lines 6-13), the system migrates containers to prevent cooling systems from maxing out and driving up power consumption, a situation rooted in a nonlinear load-power relationship. Containers, ranked by their load on the most taxed component (line 4), are migrated in iterations based on three scenarios (line 8) —reflecting the best, average, and worst cases— to avoid network overload. This process continues until the device exits the overload state or migration becomes unfeasible for even the smallest container.

The second scenario addresses a device's underload condition (lines 14 to 19), where the supervisor aims to minimize the device's workload. The objective is to lower all component loads below a set threshold, enabling the device to enter a suspended state. This saves energy across all the device's internals, including the motherboard, extra fans, etc. Upon all hardware components reaching minimal load levels (line 14), the **SM** seeks to migrate all active containers to proper devices (line 15). If successful, the device enters a suspension state, reactivating only upon receiving a certain number of migration requests within a given timeframe (lines 16 and 17).

In both scenarios, the **SM** follows Algorithm 2's migration policy, using region queries ($O(log(n))$, where n is the device count) to find candidates with sufficient resources (lines 2-8). It then chooses the destination with minimal power consumption across all hardware components (lines 12-17).

The third scenario (lines 20-32) addresses stable deployments where load balancing is impossible due to resource constraints or doesn't fit the first two scenarios' criteria. The **SM** determines the device's stability by analyzing its maximum downtime over the last million operations, tracked via the $MovementsWindow$. Optimizing data package locations is crucial due to the energy and time costs of data migrations. The algorithm selects the most energy-efficient connection device as the centroid for data placement (line 24),

migrating data there if hardware resources are sufficient and data replicas are preserved (line 26). Failing that, the algorithm seeks alternatives through the nearest neighbor query (line 30), iterating until migration success or return to the starting node.

The three scenarios implement an energy recovery method, as described in Algorithm 3. Each **SM** assesses the energy expended on its last operation (line 1) and the power saved by it (line 2). Once enough time has passed to ensure that the saved energy is at least double the energy expended, the algorithm attempts to execute another operation.

Algorithm 1: Main algorithm for device **D**

```
1  timeSlice ← 1min;
2  MovementsWindow ← [];
3  size ← size(containerList);
4  sortByfactors(containerList);
5  while true do
6      if isOverLoadSituation then
7          while isOverLoadSituation() do
8              for M_i in [containerList[size − 1],
                 containerList[size/2], containerList[0]] do
9                  if MigrateContainer(M_i) = true then
10                     containerList.delete(M_i);
11                     break;
12             if No container moved then break;
13         WaitUntilEnergyIsRecuperated() # see Algorithm 3;
14     else if isUnderLoadSituation then
15         migrateAllContainers(containerList);
16         if containerList.empty() then
17             suspendUntilExternalCall();
18         else
19             WaitUntilEnergyIsRecuperated() # see
                 Algorithm 3;
20     else if NoMovements(timeSlice) then
21         dataUnit ← selectMostUsed();
22         centroid ← D;
23         while centroid isDifferentTo D do
24             centroid ←
                 lessNetworkConsumption(dataUnit.connectedPeers());
25             if move(dataUnit, centroid) then
26                 sendAndDeleteData(D, centroid);
27                 MovementsWindow.add(now −
                     lastOperationTime);
28                 break;
29             else
30                 centroid ←
                     CustomNearestNeighborQuery(centroid);
31         WaitUntilEnergyIsRecuperated();
32         timeSlice ← max(MovementsWindow);
```

Algorithm 2: MigrateContainer(M) for device **D**

```
1  V ← buildVector(M);
   ResourceCandidateLists ← CustomRegionQuery(V);
   CandidateList ← intersect(ResourceCandidateLists, V);
2  foreach D_l in CandidateList do
3      sendVector(V, D_l);
4      WaitForCandidateReponse(20);
5      if isCandidatePossible(D_l, M) then
6          lock(D_l);
7      else
8          DeleteFromCandidateList(D_l);
9  if isEmpty(CandidateList) then
10     return false;
11 cheapestDevice ← CandidateList[0];
12 foreach D_l in CandidateList do
       // Project consumption for CPU, RAM, NIC, Storage,
          and GPU
13     c ← projectConsumption(V, D_l);
14     if c < projectConsumption(V, cheapestDevice) then
15         cheapestDevice ← D_l;
16 sendAndDeleteContainer(M, cheapestDevice);
17 MovementsWindow.add(now − lastOperationTime);
18 return true;
```

Algorithm 3: WaitUntilEnergyIsRecuperated for a device **D**

```
1  usedWatts = getOperationsPower();
2  timeSavings = 2 × (usedWatts × operationsTime) / (InitialWatts − FinalWatts);
3  sleep(timeSavings);
```

VI. Experiments and Results

To demonstrate our scheduler effectiveness, we evaluated three main variables through experiments: **total and average hardware component energy consumption across all devices, total containers runtime, and average containers' QoS**. We considered two scenarios: one comparing our algorithm **(Multidimensional Distributed Scheduling - MDS)** against an initial deployment based on resource availability (as most schedulers do today), and another comparing our approach to two other energy-aware scheduling strategies, explained further below. We made this distinction because, to the best of our knowledge, our method uniquely considers all elements of the U space comprehensively, including connections to data items that containers may have, an aspect not comparable with other algorithms. Each scenario deployed 50 devices and executed a sequence of 100, 200, 300, 400, 500, and 600 containers to evaluate scalability and performance under varying workloads.

Regarding the experimentation environment, we utilized the PISCO simulator [28] for its capacity to dynamically abstract and deploy any network topology and scheduling heuristic. We chose PISCO due to our physical infrastructure limitations and our interest in technology neutrality. To our knowledge, PISCO is the only simulator capable of such comprehensive abstraction.

A. Power, load, and QoS values

Using PISCO, we deployed two scenarios by considering delimited random values for hardware capabilities and power consumption, based on appropriate metrics collections[5,6,7,8,9] [7]. Additionally, we explored a range of workloads to assess their needs for different computing capabilities and data package sizes, as outlined in the Figure 1. For **QoS**, we considered the average ratio of computational capacities required by a workload versus what is provided across all

[5]https://www.samsung.com/semiconductor/global.semi.static/Samsung_SSD_860_EVO_Data_Sheet_Rev1.pdf

[6]https://a1dev.com/sd-bench/stats/cpu-frequency/

[7]https://www.memorybenchmark.net/amount-of-ram-installed.html

[8]https://nationalpcbuilder.com/graphics-card-gpu-power-consumption-comparison-chart/

[9]https://www.tomshardware.com/features/ssd-vs-hdd-hard-drive-difference/

hardware components [4]. The first scenario, which is more comprehensive, considers the transfer rate of containers to their data items, influenced by the physical distance between devices (the 6th dimension of space U, benchmarked against internet standards[10]). Here, we equally weighted hardware resources and connections.

CPU	RAM	NIC	DISK	GPU
1.2GHz - 4.8GHz	2000MB - 32000MB	100 Bb/s - 1000 Mb/s	SSD: 101.0-800 Mb/s HDD: 80-160 Mb/s	0.4GHz - 3.8 GHz
Cap. 10pF Volt.12v	3-5W	Idle: 0.4944W Working: 1.1349W	Idle (SSD/HDD): 0.05/5.4W Working(SSD/HDD): 2.2/8.0W	75W-346W

Resources	Workload (Min - Max)
CPU (GHz)	0.8 - 3.2
RAM (MB)	150 - 300
Network Upload (MB/s)	2 - 7
Network Download (MB/s)	3 - 9
Disk (MB/s) (both op.)	0.1 - 200
Workload's size (MB)	10 - 500
Data Package (Size)	1MB - 1000MB
Connections demand (average transfer rate)	1MBps - 50MBps

Fig. 1. Consumption and load values

B. The first Scenario

Our first goal is to assess our algorithm against an initial deployment based solely on resource availability. As Figure 2 depicts, when workloads increase, MDS maintains QoS improvement, for example, from 48.1% to 55.5% at the 600-container level. MDS also shows gains in time efficiency, decreasing an average of 46% across all cases. Energy consumption analysis reveals that MDS globally saves energy. For instance, CPU energy dropped from 105459.26 to 64213.93 joules and GPU energy from 90470.04 to 49050.85 joules at the 600-container scale. MDS saves energy also for storage devices. At the maximum scale, it decreases from 5310.47 to 3311.49 joules, highlighting the importance of storage optimization for scheduling strategies. For NIC energy, MDS shows an increase in energy consumption in specific cases (e.g., 200, 300, and 600 containers). This is attributed to MDS's workload migrations; however, it does not detract from the overall strategy, as these migrations enable significant energy savings in other critical hardware components, thus enhancing global energy efficiency.

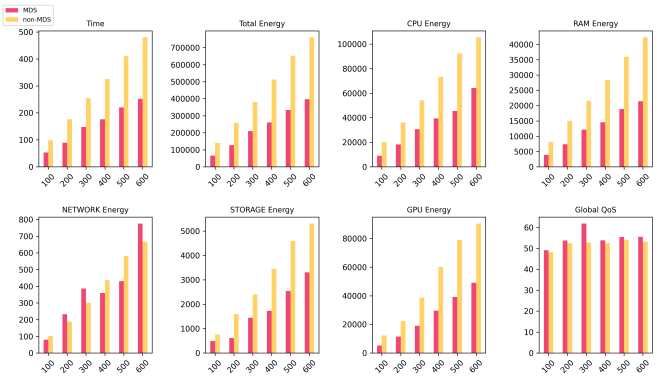

Fig. 2. MDS vs. HW availability-based Initial deployment

C. The Second Scenario

For comparative purposes, we evaluated our algorithm against other energy-aware approaches: the algorithm proposed by Ferere et al. [6], and the scheduler proposed by Li et al. [8]. We selected these energy-conscious approaches to compare the effectiveness of using data structures versus an energy-aware method selective of hardware components and a strategy based on processing historical data. **Ferere's** algorithm uses a two-step approach to assign queries to servers, optimizing for energy and resource efficiency. It first profiles servers and queries by their CPU, RAM, GPU, and network latency. Then, it allocates tasks using a Server Ability to Answer a Query **(SAAQ)** score, choosing servers that fulfill query needs using the least resources to save energy. **Li et al.** propose an energy-efficient Spark scheduler (**EASAS** in this paper) that aligns with Big Data SLA. It uses a dynamic strategy that records task times and energy, adjusting task assignments to prioritize energy efficiency across a varied cluster. Tasks are ranked in a queue by their energy consumption rate, with lower rates preferred, promoting even execution times.

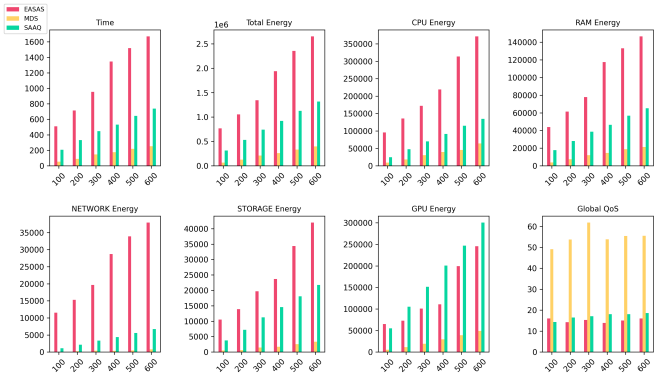

Fig. 3. MDS vs. EASAS and SAAQ

As Figure 3 shows, MDS improves QoS more than the other two approaches, for instance, from 48.1% to 55.5% at the 600-container level, surpassing EASAS and SAAQ, which cap at 15.98% and 18.64%, respectively. In terms of execution time, MDS demonstrates superior efficiency, reducing time by an average of 46% across all cases, with a notable decrease to 252 seconds at the 600-container scale, compared to EASAS's 1671 seconds and SAAQ's 738 seconds. This enhanced efficiency is further mirrored in the overall energy consumption, where MDS reports a total energy usage of 397040.47 joules at the 600-container workload, significantly lower than EASAS's 2652380.47 joules and SAAQ's 1315532.51 joules. These findings highlight MDS's capacity to not only optimize execution time and QoS performance but also significantly improve energy efficiency across all the hardware components.

VII. Conclusions and Future Work

We have presented a distributed scheduling algorithm based on multidimensional spaces and data structures. This approach indexes devices by various heterogeneous criteria, such as hardware availability, hardware characteristics, or the physical location of devices. In this work, our primary goal has been comprehensive energy saving, demonstrating that by managing

[10]https://wondernetwork.com/pings

various variables such as the characteristics of hardware components and the location of devices, we have been able to save time and energy and gain in QoS compared to an initial deployment that only considers availability and other schedulers based on hardware type or historical execution of workload charges. We are working on the actual implementation of this approach in K8s architectures, as part of a custom scheduler.

VIII. Acknowledgment

This research was supported by the internal project **UbiC-Data: Data Science for Ubiquitous Computing Environments**, Number VIU23008, financed by Valencian International University, Spain and partially supported by the "GreenSE4IoT: Towards Energy-efficient Software for Distributed Systems" project whose code is STIC-AMSUD 22-STIC-04.

References

[1] M. Adhikari, T. Amgoth, and S. N. Srirama, "A survey on scheduling strategies for workflows in cloud environment and emerging trends," *ACM Computing Surveys*, vol. 52, no. 4, pp. 1–36, 2019.

[2] X. Liu and R. Buyya, "Resource management and scheduling in distributed stream processing systems: a taxonomy, review, and future directions," *ACM Computing Surveys*, vol. 53, no. 3, pp. 1–41, 2020.

[3] Q. Luo, S. Hu, C. Li, G. Li, and W. Shi, "Resource scheduling in edge computing: A survey," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 4, pp. 2131–2165, 2021.

[4] A. V. H. Humberto, "An energy saving perspective for distributed environments: Deployment, scheduling and simulation with multidimensional entities for software and hardware," Ph.D. dissertation, UPPA, https://www.theses.fr/s342134, 2022.

[5] H. H. Alvarez Valera, M. Dalmau, P. Roose, J. Larracoechea, and C. Herzog, "An energy saving approach: Understanding microservices as multidimensional entities in P2P networks," in *SAC*, ser. SAC '21. ACM, 2021, p. 69–78.

[6] D. Ferere, I. Dongo, and Y. Cardinale, "SAAQ: A characterization method for distributed servers in ubicomp environments," *Sensors*, vol. 22, no. 17, p. 6688, 2022.

[7] B. Gul, I. A. Khan, S. Mustafa, O. Khalid, S. S. Hussain, D. Dancey, and R. Nawaz, "Cpu and ram energy-based sla-aware workload consolidation techniques for clouds," *IEEE Access*, vol. 8, pp. 62 990–63 003, 2020.

[8] H. Li, H. Wang, S. Fang, Y. Zou, and W. Tian, "An energy-aware scheduling algorithm for big data applications in spark," *Cluster Computing*, vol. 23, pp. 593–609, 2020.

[9] F. Pop, V. Cristea, N. Bessis, and S. Sotiriadis, "Reputation guided genetic scheduling algorithm for independent tasks in inter-clouds environments," in *Internat. Conf. on Advanced Information Networking and App Workshops*, 2013, pp. 772–776.

[10] H. Yuan, J. Bi, W. Tan, M. Zhou, B. H. Li, and J. Li, "TTSA: An effective scheduling approach for delay bounded tasks in hybrid clouds," *IEEE Transactions on Cybernetics*, vol. 47, no. 11, pp. 3658–3668, 2017.

[11] Z. Zhong, J. He, M. A. Rodriguez, S. Erfani, R. Kotagiri, and R. Buyya, "Heterogeneous task co-location in containerized cloud computing environments," in *IEEE 23rd Internat. Symposium on Real-Time Distributed Computing*, 2020, pp. 79–88.

[12] Z. Zhong, M. Xu, M. A. Rodriguez, C. Xu, and R. Buyya, "Machine learning-based orchestration of containers: A taxonomy and future directions," *ACM Computer Surveys*, 2022.

[13] X. Zuo, G. Zhang, and W. Tan, "Self-adaptive learning pso-based deadline constrained task scheduling for hybrid iaas cloud," *IEEE Transactions on Automation Science and Engineering*, vol. 11, no. 2, pp. 564–573, 2014.

[14] N. Hamdi and W. Chainbi, "A survey on energy aware vm consolidation strategies," *Sustainable Computing: Informatics and Systems*, vol. 23, pp. 80–87, 2019.

[15] Z. Zhong and R. Buyya, "A cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources," *ACM Trans. Internet Technol.*, vol. 20, no. 2, 2020.

[16] Y. Hao, J. Cao, Q. Wang, and J. Du, "Energy-aware scheduling in edge computing with a clustering method," *Future Generation Comp. Systems*, vol. 117, pp. 259–272, 2021.

[17] P. Chhikara, R. Tekchandani, N. Kumar, and M. S. Obaidat, "An efficient container management scheme for resource-constrained intelligent iot devices," *IEEE Internet of Things Journal*, vol. 8, no. 16, pp. 12 597–12 609, 2021.

[18] H. H. Alvarez Valera, M. Dalmau, P. Roose, and C. Herzog, "The architecture of Kaligreen V2: A middleware aware of hardware opportunities to save energy," in *IOTSMS*, 2019, pp. 79–86.

[19] H. H. Alvarez Valera, P. Roose, M. Dalmau, C. Herzog, and K. Respicio, "Kaligreen: A distributed scheduler for energy saving," *Procedia Computer Science*, vol. 141, pp. 223–230, 01 2018.

[20] V. Gaede and O. Günther, "Multidimensional access methods," *ACM Computer Surveys*, vol. 30, no. 2, p. 170–231, 1998.

[21] C. du Mouza, W. Litwin, and P. Rigaux, "Sd-rtree: A scalable distributed rtree," in *IEEE 23rd Internat. Conf. on Data Engineering*, 2007, pp. 296–305.

[22] P. Ganesan, B. Yang, and H. Garcia-Molina, "One torus to rule them all: Multi-dimensional queries in p2p systems," in *7th International Workshop on the Web and Databases: Colocated with ACM SIGMOD/PODS 2004*. ACM, 2004, p. 19–24.

[23] M. Cai, M. Frank, J. Chen, and P. Szekely, "Maan: a multi-attribute addressable network for grid information services," in *First Latin American Web Congress*, 2003, pp. 184–191.

[24] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," vol. 31, 01 2001, pp. 149–160.

[25] A. Noureddine, "Powerjoular and joularjx: Multi-platform software power monitoring tools," in *18th Internat. Conf. on Intelligent Environments*, 2022.

[26] A. Noureddine, R. Rouvoy, and L. Seinturier, "Monitoring energy hotspots in software," *Automated Software Engg.*, vol. 22, no. 3, p. 291–332, 2015.

[27] A. Karyakin and K. Salem, "An analysis of memory power consumption in database systems," in *13th Internat. Workshop on Data Management on New Hardware*. ACM, 2017.

[28] P. Roose, M. Dalmau, and H. H. A. Valera, "Système et procédé de planification de traitement de programme," Patent Pattent N°2 107 199, 2021.