Integrating Heuristic and Machine-Learning Methods for Efficient Virtual Machine Allocation in Data Centers

Ali Pahlevan, Xiaoyu Qu, Marina Zapater, Member, IEEE, David Atienza, Fellow, IEEE

Abstract—Modern cloud data centers (DCs) need to tackle efficiently the increasing demand for computing resources and address the energy efficiency challenge. Therefore, it is essential to develop resource provisioning policies that are aware of virtual machine (VM) characteristics, such as CPU utilization and data communication, and applicable in dynamic scenarios. Traditional approaches fall short in terms of flexibility and applicability for large-scale DC scenarios. In this paper we propose a heuristic-and a machine learning (ML)-based VM allocation method and compare them in terms of energy, quality of service (QoS), network traffic, migrations, and scalability for various DC scenarios. Then, we present a novel hyper-heuristic algorithm that exploits the benefits of both methods by dynamically finding the best algorithm, according to a user-defined metric. For optimality assessment, we formulate an integer linear programming (ILP)-based VM allocation method to minimize energy consumption and data communication, which obtains optimal results, but is impractical at runtime. Our results demonstrate that the ML approach provides up to 24% server-to-server network traffic improvement and reduces execution time by up to 480x compared to conventional approaches, for large-scale scenarios. On the contrary, the heuristic outperforms the ML method in terms of energy and network traffic for reduced scenarios. We also show that the heuristic and ML approaches have up to 6% energy consumption overhead compared to ILP-based optimal solution. Our hyper-heuristic integrates the strengths of both the heuristic and the ML methods by selecting the best one during runtime.

Index Terms—cloud data centers, greedy heuristic, machine learning, hyper heuristic, integer linear programming, energy-network traffic trade-offs, QoS, scalability assessment.

1. INTRODUCTION

CLOUD computing has recently been brought into focus in both academia and industry due to the increase of applications and services. Consequently, there has been a rapid growth in the number of data centers (DCs) in the world, leading to high energy consumption costs [1]. An approximate power breakdown shows that both information technology (IT) equipments (i.e., server, storage and network) and cooling system encompass over 88% of the total power of a modern DC [2]. Moreover, the explosion of data-intensive applications in current DCs, has led to uneven traffic, bandwidth and communication needs across applications [3].

Virtual machine (VM) consolidation [4] is widely used to minimize energy consumption based on the peak, off-peak, or average CPU usage of VMs [5] to pack them together into the minimum number of servers without degrading quality of service (QoS). However, the dynamic nature of cloud workloads impacts consolidation in two aspects: i) the CPU-load correlation across VMs (i.e., the similarity of CPU utilization traces and the coincidence of their peaks); and ii) the data exchange across VMs (i.e., data correlation) [6]. In this context, several works use heuristics to either address CPU-load correlation, consolidating VMs when their peak utilizations do not coincide [7], [8], or take data correlation into account [9], [10]. Nonetheless, jointly incorporating both metrics in a multi-objective optimization is an important aspect missing from prior works, as studied in a recent survey [11], which significantly increases the complexity of VM allocation.

As complexity raises, integer linear programming (ILP)-based methods become unfeasible at runtime to provide an optimal solution. Similarly, heuristics are problem-specific and less sensitive to dynamic environments, and their benefits become limited for large problems. Thus, when tackling dynamic problems with large state and/or action spaces, machine learning (ML) methods, and in particular reinforcement learning (RL), are suitable techniques [12]. However, in real DC scenarios, VM allocation faces the need to incorporate and assess a wide range of metrics (energy, QoS, network, etc.). This challenges the deployment of ML methods due to their limited configurability. The proposal of methods that balance the trade-offs across these metrics, or dynamically change the optimization goals during run-time to meet DC constraints, remains an open challenge, as it requires a deep assessment on the previous techniques, together with the integration of their strengths. Within this context, hyper-heuristics are “heuristics that choose heuristics” [13] and allow to determine which method to use depending on the current DC status, providing better trade-offs than when using the methods separately.

In this paper, we first propose and assess two different approaches to tackle the VM allocation problem: i) a two-phase greedy heuristic, and ii) a ML-based approach. Both approaches exploit CPU-load and data correlations, together with information about DC network topology. Our strategies consolidate VMs into the minimum number of servers and racks, and set dynamic voltage and frequency scaling (DVFS) appropriately. Then, we present a novel hyper-heuristic method that integrates the strengths of both heuristic and ML methods. We evaluate our approaches in terms of energy consumption, QoS degradation, network traffic, number of migrations and scalability, and compare them to an ILP-based optimal solution and state-of-the-art methods.

In particular, our main contributions are as follows:

- We propose a multi-objective hyper-heuristic method to...
determine dynamically which method among heuristic and ML is to be used at each time.

- We propose two energy- and network-aware VM allocation methods: i) a two-phase greedy heuristic, and ii) a low-complexity ML-based approach that uses the value iteration algorithm to assign VMs to servers.
- We provide an evaluation of the flexibility, scalability, benefits and drawbacks of heuristic vs. ML methods for the highly dynamic and complex VM allocation problem.
- We compare our proposed solutions with an ILP-based method, that provides an optimal solution, and also with two methods in the state-of-the-art.

Our results show that the heuristic, ML, and hyper-heuristic methods reach almost similar results in terms of energy consumption (< 2% difference). However, the ML method improves server-to-server network traffic by up to 24% and reduces execution time by up to 480x when compared to conventional approaches for large-scale problems. Also, the hyper-heuristic obtains better trade-offs for different objectives when compared to other approaches.

The remainder of this paper is organized as follows. Sect. II reviews related work. In Sect. III, we describe the system model and used application. Sect. IV provides an overview of the problem description. In Sect. V to VIII we introduce our proposed methods. Sect. IX to XI present the experimental setup and results, followed by conclusion in Sect. XII.

II. RELATED WORK

Research on VM allocation can be generally categorized in energy- and network-aware methods.

A. Energy-aware VM allocation

Regarding energy-aware methods, when deciding the allocation of VMs to physical servers, several works only check that the total size of VMs’ load does not exceed the server’s capacity [5], [14]. Hence, consolidation solutions are proposed based on per-VM workload characteristics, i.e., the peak, off-peak, and average utilization of VMs [4], [15]. Ahvar et al. [16] present a cost and carbon emission-efficient VM placement method and optimize network between DCs, using fuzzy sets. A few studies [7], [17]–[19] consider other VM’s attributes, like CPU-load correlation, to achieve further energy savings. Among the latter, Verma et al. [17] define VMs’ CPU utilization in a time series as a binary sequence where the value becomes ‘1’ when CPU utilization is higher than a threshold. However, this aggressive quantization alters the original behavior and is only applicable when VM envelopes are stationary. Meng et al. [18] propose a VM sizing technique that pairs two uncorrelated VMs into a super-VM by predicting the workloads. But, once the super-VMs are formed, this solution does not consider dynamic changes, which limits further energy savings. Kim et al. [7] present a CPU-load correlation-aware solution based on the First-Fit-Decreasing heuristic to separate CPU-load correlated VMs. The main drawback of this approach is that it cannot be used for online management at large-scale DCs due to its high computational overhead. Lin et al. [19] utilize the peak workload characteristics to measure the similarity of VMs’ workload. This method achieves better results for VMs whose workload follows a Gaussian distribution.

Ruan et al. [20] propose a dynamic migration-based VM allocation method to achieve the optimal balance between server utilization and energy consumption such that all servers operate at the highest performance-to-power levels. Wang et al. [21] also address a matching-based VM consolidation mechanism using migration such that active servers can operate close to a desirable utilization threshold. ML is a recently used technique for energy-aware VM allocation in DCs. Farahnakian et al. [22] and Masoumzadeh et al. [23] present a cooperative multi-agent learning management to minimize the number of active servers managing the overutilized and underutilized servers. Masoumzadeh et al. [24] introduce a VM selection task using a fuzzy Q-learning technique to make decisions for migration. Ravi et al. [25] also present an energy-efficient Q-learning based technique to decide on VM migrations. The main drawback of those approaches is their high VM migration overhead. Thus, as opposed to short-term decision, Chen et al. [8] propose a long-term VM consolidation mechanism such that the total demand of co-located VMs nearly reaches their host capacity during their lifetime period. This algorithm first detects the utilization pattern of each VM based on the four types of simple pulse functions. Then, a heuristic algorithm is used to place all VMs in as few servers as possible. They show a significant reduction in the number of migrations, i.e. only 4% of the total number of VMs, compared to dynamic short-term decision-based methods. Nonetheless, this work ignores the original utilization pattern of the VMs, which is usually a combination of those simple types of functions, achieving lower energy savings. Moreover, none of these approaches consider the data communication between VMs in the allocation process.

B. Network-aware VM allocation

To provide better network resource usage and improve the performance of applications, certain algorithms [16], [26], [27], take into account the communication among VMs in the DC. However, some works assume that data dependencies are given in the form of a directed acyclic graph (DAG). Differently, in practice there are often cyclic communication scenarios, where two VMs regularly exchange information in both directions. As a result, Biran et al. [10] propose two new heuristic algorithms to address bidirectional data communication under time-varying traffic demands. However, without the consideration of CPU-load correlation, both approaches are sub-optimal to minimize energy consumption.

III. DATA CENTER MODELLING AND APPLICATIONS

In this section we first define the considered DC configuration and describe the used power model. Then, we detail the type of applications tackled in this paper. For the sake of clarity, Table I summarizes the main parameters and notations used throughout this paper.

A. Data center configuration

A DC typically encompasses two main structural components of our interest: i) computing elements –IT– comprising servers and network switches, and ii) cooling systems. We
consider a typical raised-floor air-cooled DC [28] with 8 racks arranged in hot-cold aisle, as depicted in Fig. 1. Each rack contains 10 to 16 servers, each server with its dedicated power unit, fans and disks. Each rack has one Top-of-Rack (ToR) switch to which all servers in the rack are physically attached, providing a bandwidth $B_{agr}$. The ToR switch is the lowest layer of a 3-layer tree network topology [14]. Several ToR switches, connect to an aggregate switch, which consolidates traffic into a higher speed link with bandwidth $B_{agr}$. The Aggregation-layer switches connect to a core router that redirects incoming requests to servers, and tracks and routes VM migrations from one server to another server or DC in another location. The Core router operates in network backbone with the highest speed and bandwidth $B_{cr}$.

**Data center power model**

In this work, we model total DC power consumption ($P_{DC}$) as the sum of: i) IT power ($P_{IT}$), including total server ($P_s$) and network ($P_{net}$) power, and ii) the cooling power ($P_c$):

$$ P_{DC} = P_{IT} + P_c, \quad P_{IT} = P_s + P_{net} $$

Server power can be further extended as: $P_s = \sum_{j=1}^{N_s} P_j$, where $P_j$ and $N_s$ specify the $j^{th}$ server power and the total number of servers in the DC, respectively. Following the same methodology than in previous research in the area [1], [29], the major contributors to power consumption in servers are the CPU, memory, fans and disks. Among these, CPU has the largest effect on power, and previous research shows that the power-frequency relation is linear for a given CPU-intensive workload [30]. Hence, server power can be calculated as [29]:

$$ P_j = P_{j_{static}} + P_{j_{dyn}} $$

$$ \begin{align*}
P_{j_{static}} &= P_{disk} + P_{fan} + P_{leak} + P_{idle}\[4pt]P_{j_{dyn}} &= P_{cpu} \cdot (U_{cpu}/100) + P_{mem} \cdot (U_{mem}/100)
\end{align*} $$

where under $P_{j_{static}}$ we consider all the contributions to power that are workload-independent. $P_{disk}$ and $P_{fan}$ are considered constants for our particular workload, and respectively account for the power consumption of disks and fans. $P_{leak}$ refers to temperature-dependent leakage power. We consider a high fan speed and a low inlet temperature to reduce the effect of temperature-dependent leakage power, considering it as a
worse-case constant. $P_{idle}^{cpu}$ and $P_{idle}^{mem}$ are constants that show the idle power consumption of CPU and memory respectively. $P_{dyn}$ accounts for server dynamic, and is proportional to workload. $P_{cpu}$ and $P_{mem}$ are the fitted constants of our linear model for dynamic CPU and memory power, respectively. Finally, $U_{cpu}$ and $U_{mem}$ represent CPU and memory utilization of the server, and vary between 0 and 100. The fitted values of $P_{cpu}^{dyn}$ and $P_{mem}^{dyn}$ have been obtained from the models proposed in previous work [1], [29], and assuming that for our workloads (i.e., virtualized banking applications) memory utilization is proportional to the amount of memory accesses.

Network power ($P_{net}$) is the summation of power of all turned-on switches in each layer of network topology, as [14]:

$$P_{net} = P_{tor} + P_{agr} + P_{cr}$$  \hspace{1cm} (3)

where $P_{tor}$, $P_{agr}$ and $P_{cr}$ denote power consumption of ToR and aggregation-layer switches, and core router respectively.

Finally, to account for cooling power consumption $P_c$, we use a time-varying Power Usage Effectiveness (PUE) model, proportional to power consumed by IT components; i.e. $P_c = (PUE - 1) \cdot P_T$, as presented in [31]. In this model, PUE mainly depends on the DC room and outside temperature.

C. Applications description

Due to the tight dependency between the nature of the applications and the techniques to be applied, in this paper we consider business-critical applications [32] and, in particular, virtualized banking applications executing batches of tasks. To characterize the power and performance of these applications, we make use of synthetic workloads which are representative of real banking applications according to our industry partners. As realistic CPU usage and memory footprint traces, we use the publicly available traces from Bitbrains, a service provider that provides service to banks such as ING [32]. Bitbrains traces provide data every 5 minutes. Half of the VMs have low variance on CPU usage. However, around 20% of VMs have a very unstable CPU usage. Concerning memory footprint, 80% of VMs use less than 1GB of memory, and in most of them maximum is below 8GB [32].

The Bitbrains traces do not provide any information on the amount of data being exchanged across VMs. Thus, we synthetically generate the amount of data communicated between each pair of VMs using a non-uniform distribution, as detailed in Section IX-A.

IV. PROBLEM DESCRIPTION

In this section we present an overview of the problem description, including the objectives, inputs and outputs. The goal of all the methods proposed in this paper is to minimize the overall server ($P_i$) and accordingly DC ($P_{DC}$) power consumption, and network traffic ($D_{total}$) by means of efficient consolidation-based VM allocation. All the proposed approaches examined consist of two consecutive steps: i) VM characterization and ii) VM allocation, as shown in Fig. 2.

The VM characterization step is used to determine the data communication patterns between VMs, the CPU utilization, and the memory requirements for the next time slot. For the heuristic and ML approach, we use a last-value predictor to estimate these parameters. The last-value predictor considers that the CPU and memory utilization traces of current time slot (e.g., a time series of $n$ samples, each sample gathered every $5$ minutes) are exactly the same than on the previous time slot, as shown in Fig. 3. The specified areas in the figure indicate miss-predictions that can potentially lead to server overutilization and violations, when the predicted CPU utilization is lower than the real one. For the ILP-based method, we assume that, at the beginning of each time slot, all the VM characteristics for the time slot are known (oracle predictor).

The VM Allocation step takes the input VMs CPU, memory, and data communication requirements from the previous step, and the DC network topology. Every time slot, the VM allocation method re-allocates the existing VMs, migrating them if needed to the minimum number of servers such that highly data-correlated VMs are placed together, while highly CPU-load correlated VMs are placed apart. By lowering the number of active servers and racks, unused computing equipment (i.e., idle servers and network switches) can be turned off during that time slot to increase energy efficiency. Turning on/off IT equipment can be applicable to such applications when time slot duration is long enough to prevent significant performance degradation caused by the long transition latency between power modes and changes of resource demands.

After allocating all VMs to the minimum number of servers, the minimum frequency level ($Freq^j_T$) among all the samples in time slot $T$ for each turned-on server is computed as follows:

$$Freq^j_T \geq \left( \frac{U_{cpu}^T}{100} \right) \cdot f_{max}, \quad \forall n$$  \hspace{1cm} (4)

where $U_{cpu}^T$ indicates the total CPU utilization of the $j^{th}$ server at $n^{th}$ sample in time slot $T$. In this paper, we assume a homogeneous DC with all servers of the same type. Therefore, $f_{max}$ is equal for all servers and determines the maximum

---

**Fig. 2:** Overall diagram of the proposed scenario.

**Fig. 3:** Time slot and sample description.
frequency level. This equation guarantees that the selected frequency is sufficient to avoid server overutilization for all the samples in time slot $T$. Thus, in the proposed scenario, violations can only occur due to miss-predictions on the VM usage (e.g., for the case highlighted in Fig. 3), when the server utilization needs a higher frequency than the one selected. In this sense, the ILP method will never exhibit violations, as it uses an oracle predictor. After allocation, for all the proposed methods during one time slot, we update the servers frequency every $t_f$ by computing the maximum server utilization occurred in the previous $t_f$ period. Finally, after computing the total IT power consumption, we update and compute the cooling system power every $t_c$ during one time slot, based on the DC room and outdoor temperatures.

V. PROPOSED ILP-BASED OPTIMIZATION METHOD

The ILP method can be divided in two minimization problems: i) power consumption (ILP-Power), and ii) data communication (ILP-Data). Due to the varying nature of DC workloads (i.e., VMs’ utilization patterns change over time), there is an optimal VM allocation for each time slot and, thus, a need to invoke the ILP-based method. The time slot duration is a parameter that can be adjusted by the DC operator depending on the granularity of the traces used, to increase accuracy. We define the ILP formulations in a generic form regardless of the network topology constraints. Hence, an optimal server-to-server data communication is obtained that represents the total network traffic. In the following subsections, we describe these two optimization goals in detail.

A. Power consumption optimization

The proposed ILP-Power VM allocation aims to minimize overall server power based on the Eq. 2. The minimization objective is given by Eq. 5, where $P^T_s$ and $P^T_{j,n}$ denote overall and $j^{th}$ server power consumption at the $n^{th}$ sample of the $T^{th}$ time slot, respectively. $N_j$ and $N_s$ are the number of servers in the DC and number of samples in one time slot, respectively. The binary variable $X^T_{j,n}$ is defined to indicate whether the $j^{th}$ server is on ($X^T_{j,n} = 1$) or off ($X^T_{j,n} = 0$) in the $T^{th}$ time slot.

We use the binary variable $\text{Place}_j^T$ to indicate whether $k^{th}$ VM ($k \in \{1, 2, ..., N_{VM}\}$) is placed on $j^{th}$ server in $T^{th}$ time slot. $N_{VM}$ is the total number of VMs available in the DC. Matrices $\text{VMcpu}_{j,n}^T$ and $\text{VMmem}_{j,n}^T$ contain the $k^{th}$ VM’s CPU utilization and memory footprint at $n^{th}$ sample, respectively, during the $T^{th}$ time slot. Similarly, $U^T_{\text{mem},j,n}$ indicates the total memory utilization of the $j^{th}$ server.

The minimization problem is subject to the constraints given in Eq. 8 to 12. Constraint 1 forces each VM to be placed only in one server. Constraints 2 and 3 enforce that aggregated CPU and memory utilizations of the VMs in $j^{th}$ server do not exceed the maximum CPU and memory capacities, i.e., $C^s$ and $C^m$, respectively. In Constraint 4, the integer variable $e^T_j$ is used to specify the number of placed VMs on the $j^{th}$ server. This value is upper-bounded by $N_{VM}$. Constraint 5 guarantees that if no running VMs are placed on $j^{th}$ server in the $T^{th}$ time slot ($e^T_j = 0$), this server can be turned off ($X^T_{j,n} = 0$).

Server power in the objective function (Eq. 5) should be written as $P^T_j = X^T_j \cdot (P^T_{\text{cpu},j,n} + P^T_{\text{mem},j,n})$. However, this would introduce a non-linearity in the ILP problem (due to the product of variables $X^T_j \cdot U^T_{\text{cpu},j,n}$ and $X^T_j \cdot U^T_{\text{mem},j,n}$) in $P^T_{\text{cpu},j,n}$.

Constraint 4 avoids this issue as, when the number of VMs on $j^{th}$ server ($e^T_j$) as well as the server CPU and memory utilization is zero, $X^T_j = 0$. On the contrary, if $1 \leq e^T_j \leq N_{VM}$, then $X^T_j = 1$. Therefore, we can write: $X^T_j \cdot U^T_{\text{cpu},j,n} = U^T_{\text{cpu},j,n}$ and $X^T_j \cdot U^T_{\text{mem},j,n} = U^T_{\text{mem},j,n}$.

\[
\min P^T_s = \sum_{j=1}^{N_j} \sum_{n=1}^{N_s} X^T_j \cdot P^T_{j,n} = \sum_{j=1}^{N_j} \sum_{n=1}^{N_s} (X^T_j \cdot P_{\text{stat},j,n} + P^T_{\text{dyn},j,n}) = \sum_{j=1}^{N_j} \sum_{n=1}^{N_s} (X^T_j \cdot P_{\text{stat},j,n} + P^T_{\text{cpu},j,n} \cdot X^T_j + P^T_{\text{mem},j,n})
\]

(5)

\[
\text{where:}
\]

\[
U^T_{\text{cpu},j,n} = \sum_{l=1}^{N_{VM}} \text{Place}_j^T \cdot \text{VMcpu}_{j,n}^T
\]

(6)

\[
U^T_{\text{mem},j,n} = \sum_{l=1}^{N_{VM}} \text{Place}_j^T \cdot \text{VMmem}_{j,n}^T
\]

(7)

subject to the following Constraints:

1. $\sum_{j=1}^{N_j} \text{Place}_j^T = 1$

2. $\sum_{j=1}^{N_j} \text{Place}_j^T \cdot C^s \leq C^s$

3. $\sum_{j=1}^{N_j} \text{Place}_j^T \cdot C^m \leq C^m$

4. $e^T_j = \sum_{k=1}^{N_{VM}} \text{Place}_j^T$

5. $X^T_j \leq e^T_j \leq X^T_j \cdot N_{VM}$

(12)

B. Data communication optimization

The amount of data exchanged between VMs directly impacts network traffic and response time. In practice, two VMs regularly exchange a varying amount of data. Our goal is to minimize total data communication (network traffic, $D^T_{\text{total}}$) amongst the servers. In our formulation, $D^T_{j,n}$ represents the $j^{th}$ server data communication; i.e., the amount of data transferred by a server at the $n^{th}$ sample of the $T^{th}$ time slot.

To express $D^T_{j,n}$, the binary variable $\text{BinVMstatus}_{j,k}^T$ indicates whether both $k^{th}$ and $l^{th}$ VMs have been allocated to $j^{th}$ server ($\text{BinVMstatus}_{j,k}^T = 0$); otherwise, $\text{BinVMstatus}_{j,k}^T = 1$ in the $T^{th}$ time slot. The matrix $V_Mdata_{j,n}^T$ contains the amount of data transferred from the $k^{th}$ to $l^{th}$ VM at the $n^{th}$ sample during $T^{th}$ time slot.

The constraints of the problem are formulated in Equations 15 to 19, as follows. Constraints 1, 2 and 3 are the same as those of ILP-Power. Constraint 4 determines the status of the $k^{th}$ and $l^{th}$ VMs on the $j^{th}$ server. The variable $V_Mstatus_{j,k}^T$ is used to specify the status of any pair of VMs based on the status of each VM on $j^{th}$ server. As this variable is the sum of two binary variables, it can take only three different values (i.e. 0, 1, and 2): i) the $k^{th}$ and $l^{th}$ VMs are allocated to the $j^{th}$ server ($\text{Place}_j^T = 1$ & $\text{Place}_{j,l}^T = 1$), then \( V_Mstatus_{j,k,l}^T = 2 \); or ii) either the $k^{th}$ or $l^{th}$ VM is allocated to the $j^{th}$ server ($\text{Place}_j^T = 1$ & $\text{Place}_{j,l}^T = 0$ or vice versa), then \( V_Mstatus_{j,k,l}^T = 1 \); or iii) neither the $k^{th}$ and $l^{th}$ VMs are allocated to the $j^{th}$ server ($\text{Place}_j^T = 0$ & $\text{Place}_{j,l}^T = 0$), then \( V_Mstatus_{j,k,l}^T = 0 \).
The original data communication objective is written as 
\[ D_{j,n} = \sum_{k=1}^{N_{VM}} Place_{j,k} \cdot \sum_{i=1}^{N_{VM}} (1 - Place_{j,i}) \cdot VData_{j,n} \] 
To remove the non-linearity in the equation, constraint 5 is used to compute per-server data communication and demonstrates that, if VMstatus_{j,k} = 0, then BinVMstatus_{j,k,l} = 0, and '1' otherwise. In other words, if both VMs are allocated to the same server, then BinVMstatus_{j,k,l} = 0.

\[
\min \ D_{total}^{T} = \sum_{j=1}^{N_{S}} \sum_{n=1}^{N_{S}} D_{j,n}^{T}
\]
where

\[
D_{j,n}^{T} = \sum_{k=1}^{N_{VM}} \sum_{i=1}^{N_{VM}} (Place_{j,k}^{T}) \cdot \sum_{i=1}^{N_{VM}} (1 - BinVMstatus_{j,k,l}) \cdot VData_{j,n}
\]
subject to the following Constraints:
1. \[ \sum_{j=1}^{N_{S}} Place_{j,k}^{T} = 1 \]  
2. \[ U_{cpu,j,n}^{T} \leq C^{s} \]  
3. \[ U_{mem,j,n}^{T} \leq C^{m} \]  
4. \[ VMstatus_{j,k,l}^{T} = (1 - Place_{j,k}^{T}) + (1 - Place_{j,l}^{T}) \]
5. \[ BinVMstatus_{j,k,l}^{T} \leq VMstatus_{j,k,l}^{T} \leq 2 \cdot BinVMstatus_{j,k,l}^{T} \]

VI. PROPOSED TWO-PHASE GREEDY HEURISTIC METHOD

In this section, we propose a two-phase greedy heuristic algorithm (Heuristic, in what follows) that, at each time slot \( T \), jointly minimizes power consumption \( P_{total}^{T} \) and network traffic \( D_{total}^{T} \) (Phase 1), and then allocates resulting traffic in a network topology-aware fashion (Phase 2).

A. Phase 1 - VM Clustering

We split this phase in two steps and use a method similar to the one presented in [6]. First, at time slot \( T \), all VMs available in the system are represented as points in a two-dimensional (2D) plane. Based on the data and CPU load correlation properties, as highly data-correlated VMs should be clustered together while highly CPU-load correlated VMs should be placed apart, a function is defined to calculate attraction and repulsion forces between each two VMs. Nonetheless, differently from the original algorithm [6], we calculate the attraction force as a worst-case peak bidirectional data exchanged between each two VMs during the time slot. Similarly, the repulsion force is computed as a worst-case peak CPU utilization when the peaks of two VMs coincide during the last time slot. As a result, the points are remapped in the 2D plane with new coordinates based on the computed forces.

In the second step, after finding the final position of the VMs, we determine the minimum number of clusters (i.e., servers), \( \hat{N}_{server} \), as follows:

\[
\hat{N}_{server} = \max \{ \hat{N}_{cpu}, \hat{N}_{mem} \}
\]
\[
\hat{N}_{cpu} = \max \{ \sum_{k=1}^{N_{VM}} VMcpu_{k,n}^{T-1} / C^{s} \}
\]
\[
\hat{N}_{mem} = \max \{ \sum_{k=1}^{N_{VM}} VMmem_{k,n}^{T-1} / C^{m} \}
\]
where \( \hat{N}_{cpu} \) and \( \hat{N}_{mem} \) denote the minimum number of servers needed to comply with the VMs’ CPU and memory utilization requirements, respectively. Hence, \( \hat{N}_{server} \) is equal to the minimum number of servers to accommodate all the VMs while satisfying both the VMs’ CPU and memory utilization.

Then, we utilize a modified version of the k-means algorithm [6] to cluster VMs with respect to the distance between two VMs obtained from the repulsion and attraction phase in the 2D plane. Differently from the original k-means algorithm, we define a cap per cluster (i.e., \( C^{s} \) and \( C^{m} \)) when considering the VMs’ CPU and memory utilization. Moreover, the initial centroid of clusters is not set randomly, but instead calculated based on the last position of points available in the previous time slot. We start with the minimum number of clusters, \( \hat{N}_{server} \), to allocate the VMs to the clusters with shortest distance. If unfeasible, we increment the number of clusters by one. The process is iterated until all VMs are allocated to the minimum number of possible clusters. Our method guarantees that the total load of each cluster at each sample does not exceed \( C^{s} \) and \( C^{m} \) during the time slot. However, violation occurs due to miss-predictions, leading to delays in workload execution and, eventually, to their execution in the next time slot (with 100% prediction accuracy, no violation occurs).

B. Phase 2 - Clusters Allocation

In this phase, we allocate the clusters to the appropriate servers considering the DC network structure as described in Algorithm 1. This algorithm fills up the racks one by one, reducing the number of active switches, and minimizing network power, while keeping highly-communicating servers close to each other.

Algorithm 1 Cluster Allocation

Input: Network topology and data communication graph (G_data = (V,E))
\( V \) = clusters & \( W(E) \) = data communication between clusters
Output: Cluster allocation
1: \( \text{Edge}^{\text{wmax}} \leftarrow \text{Max. edge (} W(E_{u,v}) + W(E_{v,u}) \text{)} \) between any \( u \) and \( v \)
2: \( NT_{iag,ag} \leftarrow 0 \) Initial network traffic of \( h^{th} \) agr. switch
3: for \( f = 1 \) : Total racks do
4: \( NT_{iag,ag} \leftarrow 0 \) Initial ToR switch network traffic of \( r^{th} \) rack
5: Unused servers of \( r^{th} \) rack \( \rightarrow \) Total servers of \( r^{th} \) rack
6: while (Selected clusters \( \leq \) Unused servers of \( r^{th} \) rack) & (\( NT_{iag,ag} \leq B_{low} \)) do
7: if \( V_{u} \) or \( V_{v} \) has not been allocated then
8: Allocate clusters \( w \) and \( z \) to two servers in \( r^{th} \) rack
9: \( NT_{iag,ag} \leftarrow \text{Update traffic of servers of } r^{th} \text{ rack} \)
10: Update unused servers in \( r^{th} \) rack
11: Update unused servers in \( r^{th} \) rack
12: else if \( V_{u} \) or \( V_{v} \) has not been allocated then
13: Allocate \( w \) or \( z \) to one server in \( r^{th} \) rack
14: \( NT_{iag,ag} \leftarrow \text{Update traffic of servers of } r^{th} \text{ rack} \)
15: Update unused servers in \( r^{th} \) rack
16: Update unused servers in \( r^{th} \) rack
end if
17: end if
18: Combine \( V_{u} \) and \( V_{v} \)
19: Update \( W(E) \)
20: if All clusters allocated then
21: Terminate
22: end if
23: \( \text{Edge}^{\text{wmax}} \leftarrow \text{Find maximum edge weight} \)
24: Find number of selected clusters (‘1’ or ‘2’) when both clusters have not been allocated
25: end while
26: end for

The output of the modified k-means creates an edge-weighted data communication graph (G_data = (V,E)), where set \( V \) and \( W(E) \) represent the clusters and the amount of data
transferred across clusters, respectively. The algorithm first selects the maximum edge weight \((\text{Edge}^\text{max})\) and initializes the amount of traffic transferred through all aggregation-layer switches \((N^{\text{agg}}_r)\) for different groups of racks (lines 1 and 2). Then, we select the first rack and try to fill it up. For the selected rack \((r^h)\), we first initialize its ToR switch traffic \((N^{\text{agg}}_r)\) and the number of unused servers with the total number of servers available in \((r^h)\) rack (lines 4 and 5). Then, for clusters related to the selected edge (lines 6 ~ 25) if either: i) two clusters have not been allocated yet and the number of unallocated clusters is less than the unused servers in the rack, ii) \(N^{\text{agg}}_r\) of the new selected clusters is less than the \(B_{\text{tor}}\), and iii) \(N^{\text{agg}}_r\) related to the selected rack is less than \(B_{\text{agg}}\), we allocate clusters to the servers in that rack. We also update the \(N^{\text{agg}}_r\) and the number of unused servers of the \(r^h\) rack (lines 7 ~ 17). After allocation, we combine clusters \(V_w\) and \(V_c\) and update \(W(E)\) (lines 18 and 19). We repeat until violating those conditions for the rack, and then we select the next rack. This algorithm iterates until all clusters are allocated to physical servers, which guarantees that the bandwidth of switches is not exceeded.

VII. PROPOSED MACHINE LEARNING METHOD

This section describes a two-step ML approach to allocate VMs to servers. First, we generate offline different classes using k-means according to the features extracted from the VMs’ CPU utilization traces. To decide the appropriate number of classes \((K)\), we use a heuristic-based process, as explained in Section VII-A. Second, at runtime, we classify VMs into classes by determining the shortest euclidean distance to each class centroid, and then we use the value iteration algorithm, amongst the various RL methods, to allocate the VMs to physical servers. RL is particularly useful in problems with large state and/or action spaces that change dynamically over time and depend on the environment [12]. We use the first week of Bitbrains traces for class generation and for the exploration phase of the ML approach, and the second week of traces for run-time VMs classification and exploitation phase.

A. Class generation – Offline pattern detection

Class generation significantly simplifies the process of VM allocation, reducing the complexity of the value iteration algorithm. In the Bitbrains traces we observe a high-variability but also a daily periodicity in the CPU utilization traces, making them suitable for classification. As memory resources are more over-provisioned than CPU resources, and less critical, our class generation only takes into account the CPU utilization traces to generate classes. In this step, based on the time slot duration (i.e., 1 hour), we consider each VM’s CPU utilization trace per time slot as an individual pattern composed of 12 samples (1 every 5 minutes).

Feature extraction is a key point that greatly affects classification results. In this work, we select a list of features as:

- Maximum and minimum CPU utilization, to represent the range of variation and the absolute value of the traces.
- Time at which the maximum utilization happens, to enable CPU-load correlation techniques.
- Variance, as it shows the trace variability.
- Median, to account for typical values.
- Skewness, which is a measure of the trace asymmetry.
- Kurtosis, which provides an insight on the trace shape.

Then, we use the k-means method to classify CPU utilization patterns [33]. As k-means does not decide on the number of classes \((K)\), we propose the following heuristic.

**Heuristic-based process for determining the appropriate number of classes \((K)\):** First, we define a similarity score \((\phi^T_{kl})\) in Eq. 21 that expresses the similarity between any pair of VMs, \(VM_k\) and \(VM_l\) during time slot \(T\). To define this metric we use the Pearson Correlation \((\rho^l_{kl})\), which is effective to judge the similarity on the shape of the traces. However, as the Pearson Correlation cannot reflect the absolute CPU utilization value, we incorporate the euclidean distance \((\text{Dist}^T_{kl})\) over all the samples into the metric. As a result, Eq. 21 demonstrates that \(\phi^T_{kl}\) is high when two traces have both the same shape and CPU utilization absolute value. Since two VM traces may be totally the same, we consider \((\text{Dist}^T_{kl} + 1)\) in the denominator to avoid having infinite value when \(\text{Dist}^T_{kl} = 0\), and we normalize the values to \([0, 1]\).

\[
\phi^T_{kl} = \frac{\rho^l_{kl}}{\text{Dist}^T_{kl} + 1}, \quad \text{Dist}^T_{kl} = \|VMcpu_k^T - VMcpu_l^T\|_2 \tag{21}
\]

Second, for each time slot in one week, we classify the VMs into \(K\) different classes, according to their euclidean distance (exhaustively testing different values of \(K\)). After classification, we compute the per-class similarity score during each time slot \((\phi^\text{class}_{\omega,T})\), as calculated by Eq. 22, where \(N_{\omega}\) is the number of VMs available in class \(\omega\), obtaining \(K\) different scores (i.e., as many as classes). We average these \(K\) scores to obtain an average similarity score \((\phi^\text{class}_{\text{avg},T})\) per \(T\).

\[
\phi^\text{class}_{\omega,T} = \frac{\sum_{l=1}^{N_{\omega}} \sum_{i=1, l \neq k}^{N_{\omega}} \phi^T_{kl}}{N_{\omega} * (N_{\omega} - 1)} \tag{22}
\]

By increasing the number of classes \((K)\), the similarity score exhibits a logarithmic growth, achieving its highest value of ‘1’, when we have as many classes as patterns. As the number of classes directly impacts the execution time of the ML algorithm, but a high similarity is needed to achieve good accuracy, we heuristically choose a similarity score of 0.5, which leads to 150 classes \((K)\). Because managing 150 classes is still unfeasible for the ML algorithm, we first combine the centroids of classes (i.e., compute the mean of centroids that are below a certain euclidean distance). Then, we delete those classes whose number of VMs are below a certain threshold. By appropriately setting these thresholds, our method allows to reduce the number of classes from 150 to 27, i.e. \(K = 27\), without decreasing average similarity.

B. Run-time classification and value iteration algorithm

At runtime, we use the second week of traces to classify VMs into the classes resulting from the previous step in each time slot. First, we use the last-value predictor to obtain the last VMs’ patterns and extract their features. Then, we assign the VM to the class which has the shortest euclidean distance to the centroid. Finally, we use a RL technique, the value-iteration algorithm, to allocate the VMs to physical servers. Typically, RL models are composed of an agent and
an environment with a finite set of actions (A) and a state space (S). In the environment, the states are observed and the actions determined by agent are applied. The agent maps actions to states at any decision time. There is a reward function used for each state-action pair (R) which should be maximized by the agent. R(s,a,s'), thus, shows the immediate reward value obtained after performing action a in current state s, representing the next state (s') reward value [12].

The proposed ML approach consists of two phases, namely exploration and exploitation, as shown in Fig. 4. If the current state has not been previously explored, we are in the exploration phase, and the agent randomly chooses new actions for the new states, and records the new states and rewards obtained from each action separately. On the contrary, if a state has already been explored, we proceed to the exploitation phase, and use the value iteration algorithm to find the best action among the pool of actions obtained during exploration to maximize the reward. The essential idea is: if we knew the true value of each state, we would simply choose the action that maximizes the expected reward. In our case, at each communication between agent and environment, we apply a set of actions to environment (allocating only one VM to each server). Hence, we don’t initially know the state’s true value; we only know its immediate reward. For example, a state might have low initial reward but be on the path to a high-reward state. Value-iteration progressively evaluates the states until the solution converges. Nonetheless, it assumes that the agent has a great knowledge about the reward of all states in the environment. Because in a consolidation problem the reward can be clearly stated (i.e., the higher the server utilization, the higher the reward is), the value-iteration method is suitable to solve this problem.

Because the value iteration algorithm uses a vector of actions, the amount of servers active per time slot needs to be defined before VM allocation. To this end, we first compute the maximum number of VMs than can be allocated to each server (N_{VM}^{server}) and the minimum number of needed servers, i.e., \hat{N}_{server}, per time slot. Then, the agent decides the VM placement by choosing which VM class should be allocated to each server. If the selected class has no VMs, it picks a VM from the class with a shortest euclidean distance to it. After applying the actions, we update the utilization of each server, set its state and reward, and send them back to the agent. We iterate this communication until all the VMs are allocated.

Finally, to minimize traffic through the different switches considering bandwidth constraints, we fill up the racks one after the other with the servers that have higher data communication (after allocation, we sort the amount of data transferred between any pair of servers in descending order). The following subsections describe the state, actions, and the proposed reward function for the value-iteration algorithm.

1) State and action definitions

Each time slot, the agent provides one action per server to the environment. This means that \hat{N}_{server} actions are applied to \hat{N}_{server} servers. The number of communication steps in one time slot is the maximum number of VMs allowed to be allocated to one server, i.e. \hat{N}_{VM}^{server}.

The state of each server is defined based on the number of VMs allocated to it and the server utilization. In general, we use three parameters to reflect one state value. Each parameter can get a discrete value in a finite range, as shown in Table II. The first parameter, P1, indicates the number of VMs allocated to server. P2 exhibits the maximum aggregated CPU utilization of co-located VMs during the time slot. We discretize server utilization (from 0 to 100%) in 11 uniform levels. We map the predicted server utilization to the nearest higher level to minimize violations due to miss-predictions. P3 is a binary parameter that shows that if the server is active. The allocation of a VM from one class to a server is considered as an action. We choose the first VM available from the selected class. If the class is empty, we choose a VM from the non-empty class which has the minimum distance to that class.

2) Reward function

Each pair of state-action reward (next state reward) is defined per server (R_j(s,a,s') for jth server; we simply name it R_j) according to two parameters: i) the gap between the current and the maximum utilization (i.e. utilization gap, R_j^g) and, ii) the amount of data transferred by the server (R_j^d) as:

\[
R_j = R_j^g - \frac{R_j^d}{\lambda}
\]

where \lambda is a weighting factor used to keep the reward factors in the same range.

Given that we are using consolidation as a strategy for power minimization, we could have three different situations: i) fully utilized server (i.e., \geq C^0), ii) underutilized server (< C^0), and iii) overutilized server (> C^0). To minimize power consumption via consolidation, \hat{R}_j^g needs to be highest when the server is fully utilized (we choose 1000). Hence, it is enough to set lower reward values for the rest of situations. For underutilization situations, we choose a positive proportional range between 1 and C^0/10, i.e., the higher the server utilization, the higher the reward is. On the other hand, to

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Number of allocated VMs</td>
<td>0 – \hat{N}_{VM}^{server}</td>
</tr>
<tr>
<td>P2</td>
<td>Utilization of server</td>
<td>0 – 1</td>
</tr>
<tr>
<td>P3</td>
<td>Active/inactive in time slot</td>
<td>0/1</td>
</tr>
</tbody>
</table>
minimize violations and QoS degradation, we need to avoid surpassing maximum utilization ($C^i$). For this purpose, due to the higher importance of server overutilization compared to underutilized server situations, it is enough to choose a lower value to decrease the server reward. Therefore, we set a negative value to utilization above $C^i$ (we choose -1). In practice, any other negative value works, and gives the same results. Thus, $R^i_g$ can be computed as follows:

$$R^i_g = \begin{cases} C^i - U^T_{cpu_j} & U^T_{cpu_j} < C^i \\ 1000 & U^T_{cpu_j} = C^i \\ -1 & U^T_{cpu_j} > C^i \end{cases}$$

where $U^T_{cpu_j}$ represents the maximum utilization of $j^{th}$ server among all the samples in the $T^{th}$ time slot, i.e., $\max(U^T_{cpu,j,n})$.

Similarly, $R^i_d$ represents the total amount of data transferred by $j^{th}$ server, as follows:

$$R^i_d = \sum_{m=1}^{N_j} D^T_{f,m} = \sum_{k=1}^{N_M} \sum_{l=1}^{N_M} \sum_{n=1}^{N_j} VM_{data}^T_{k,l,n}$$

where $VM_{data}^T_{k,l,n} \in server_j$ & $VM_l \notin server_j$

The reward function is computed per-server, aiming to maximize server utilization while minimizing the amount of data should be exchanged between servers.

**VIII. PROPOSED HYPER-HEURISTIC METHOD**

In this section we present a hyper-heuristic algorithm to dynamically determine which method, among Heuristic and ML, should be used at each time slot $T$ to achieve a specific trade-off across the different objectives. The proposed hyper-heuristic relies on the long-term periodicity of the workloads being executed, and learns the performance of the methods over time. The considered trade-offs (objective set $O$) are power consumption ($P_{DC}$), worst-case server overutilization ($WCV$), and total network traffic of ToR ($T_{N_{tor}}$), aggregation ($T_{N_{agr}}$), and core-layer ($T_{N_{cr}}$) switches, as the most important metrics from the DC providers perspective. They are computed for each method $i$ in a cost function (CostFunction($O_i$)) as:

$$Cost_i = \alpha_1 P_{DC} + \alpha_2 WCV + \alpha_3 (\beta_1 T_{N_{tor}} + \beta_2 T_{N_{agr}} + \beta_3 T_{N_{cr}})$$

where $\alpha_l$ and $\beta_k$ are user-defined weighting factors that need to be set with respect to the importance that the user gives to a specific objective (each normalized to (0,1)), and whose value can be changed during runtime. In this work, for the sake of clarity, we decided to give the same priority to all objectives (i.e., $\alpha_1 = \alpha_2 = \alpha_3 = 1/3$). Due to the communication distance, we consider higher weight for upper network layers ($\beta_1 = 0.1$, $\beta_2 = 0.3$, and $\beta_3 = 0.6$). Thus, the lower the power consumption, overutilization and network traffic, the lower the cost value is. The proposed algorithm is as follows.

At the beginning of each time slot, one of the methods in the pool of candidate methods ($M$) (i.e., Heuristic and ML in our case) is selected. To choose which algorithm to be executed, our proposed hyper-heuristic builds a history of the performance of the Heuristic and ML methods by using the cost function (Eq. 26). As described in Algorithm 2, at the beginning of time slot $T$, we create a hash code of the previous execution ($T-1$) that is stored in a hash table (HashTable). For the first time slot, HashTable is empty and we randomly select one of the methods. We generate the hash using the function HashGenerator (line 1) that creates a binary string with the length of the selected objectives ($O$), in which each character is ‘1’ if the ML performed better in that objective than the Heuristic, and is ‘0’ otherwise. For example, hash code ”11011" shows that ML has the best results for four objectives with respect to Heuristic. In HashTable, for each observed hash, we also store two entries per method including the cost value ($Cost^t_{Hash}$, $i \in M$) and how many times we selected that method in the past ($Num^t_{Hash}$, $i \in M$). After generating the hash, the algorithm checks in HashTable whether the hash had been already observed. If it exists, we select the method with the minimum $Cost^t_{Hash}/Num^t_{Hash}$ (line 4). Otherwise, we select the method with minimum cost value for $T-1$, and we record only the observed hash (lines 6 and 7). We update the hash entries, i.e. $Cost^t_{Hash}$ and $Num^t_{Hash}$, at the end of $T$ as follows.

After executing the selected method, at the end of time slot $T$, we collect the results ($O^T$) per method (lines 9 and 10). Then, we just update the entries of the method with the minimum cost value for corresponding hash (lines 13 and 14).

**Algorithm 2 Hyper-heuristic Algorithm**

**Input:** $O^T = \{P^{T-1}_{DC}, WCV^{T-1}, T_{N_{tor}}^{T-1}, T_{N_{agr}}^{T-1}, T_{N_{cr}}^{T-1}\}$, $i \in M$

**HashTable**

**Output:** Select one method from $M$

1: $Hash ← HashGenerator(O^T)$
2: $HashObserved ← IsHashObserved(Hash, HashTable)$
3: if $HashObserved == True$ then
4: $m ←$ Select method with $\min(Cost^t_{Hash}/Num^t_{Hash})$, $i \in M$
5: else if $HashObserved == False$ then
6: $Record Hash in HashTable$
7: $m ←$ Select method with minimum CostFunction($O^T$)
8: $end if$
9: $Execute method m for time slot T$
10: Obtain $O^T$ at the end of time slot T, $\forall i$
11: $Cost_i ← CostFunction(O^T)$, $\forall i$
12: $m ←$ Find method with $\min(Cost)$
13: $Cost^t_{Hash} ← Cost^t_{Hash} + Cost_m$
14: $Num^t_{Hash} ← Num^t_{Hash} + 1$

**IX. EXPERIMENTAL SETUP AND SCENARIOS**

In this section we present the experimental setup and introduce two scenarios to compare the proposed methods.

**A. Experimental Setup**

1) Data center configuration: We consider two rows of racks in the DC. Each row consists of four 42U racks, and each rack has ten servers. We target Intel S2600GZ servers consisting of 6-core CPU (Intel E5-2620), 9 frequency levels varying from 1.3 to 2.4GHz, and 32GB of memory.

The server power consumption is modeled as in Section III-B: each server consumes constant 16W and 27.2W for disk ($P_{disk}$) and cooling fan ($P_{fan}$), respectively. We consider a high fan speed (8000rpm) and a low inlet temperature (22°C) to reduce the effect of temperature-dependent leakage power. Under this condition, leakage power ($P_{leak}$) is almost constant and 3.1W in the worst-case. Idle power for CPU ($P_{idle}$) and memory ($P_{idle}$) are 50W and 4W, respectively. The dynamic
power of CPU ($P_{P_{CPU}}$) and memory ($P_{P_{MEM}}$) are 42.5 W and 56 W at 100% utilization, respectively [29].

A three-layered tree network topology is considered. The types of ToR, aggregation-layer switches and the core router are HP5920 with 60Gbps bandwidth ($B_{Lo}$), HP6600 with 180Gbps bandwidth ($B_{Ag}$), and HP8800 with 430Gbps bandwidth ($B_{Ce}$) that dissipate 366 W, 405 W and 3500 W, respectively [14]. For cooling power consumption, we use a time-varying PUE model ranging from 1.25 to 1.55, as in [31].

2) Simulation framework: To simulate a realistic scenario, we utilized the VMs’ CPU and memory traces of Bitbrains for a time horizon of two weeks [32]. We used the first week of traces for class generation and exploration phase, and the second week for VMs classification, exploitation phase of ML and for evaluating all the methods. We also used the validated power model of Section III-B to compute DC power consumption by exploiting an in-house simulator tool written in C++, where we coded all the algorithms used in this paper.

The VM allocation and the frequency updating ($t_f$) are invoked every 1 hour, whereas the cooling system update ($t_c$) is invoked every 10 minutes. For each experiment, we have considered different number of VMs (from 50 to 1000) in DC.

Data communication between a pair of VMs is modeled by a log-normal distribution [34]. As 80% of the VMs have 800 kB/min traffic among each other while 4% of VMs have 10 times higher traffic [35], we tune log-normal distribution with a mean of 800 and uniform variance in the range of [1, 4] for each time slot. The amount of data communication between VMs varies every 5 minutes during the one-hour time slot.

3) Simulation environment: The proposed methods are carried out on a separate server equipped with a 24-core Intel CPU@1.60 GHz and 50 GB of memory. To solve the ILP, we used the CPLEX 12.3 solver available in GAMS 23.7 [36].

B. Scenarios

1) Scenario I - Optimality assessment: To evaluate the efficiency of the proposed methods we compare them to the ILP-based methods (optimal solutions), for a few number of VMs (50, 100 and 150) and a time horizon of one day. We also take advantage of the fact that in multi-service cloud scenarios, data exchange only occurs between the tenants (VMs) of each service. In other words, we can group together the VMs that exchange data between them, while these groups are isolated and do not share data to other groups. In this scenario we assume a number of groups equal to 20% of the available VMs in the DC and we uniformly distribute the VMs to groups, limiting the number of VMs per group (group size) to 10.

We generate different traffic (data communication) between VMs of the same group for each sample during one time slot. We also redistribute the VMs per time slot under the group size limitation. For instance, for the 50 VM scenario we generate 10 groups of sizes between 1-10, and ensuring that each VM is assigned to one group. Moreover, to fairly compare ILP method to other approaches, we compute the total data communication among the servers (total network traffic) for all the methods regardless of network topology constraints.

2) Scenario II - Comparison heuristic, ML and hyper heuristic in large-scale scenarios: Communication patterns contain a wide range of variations from one-to-one to all-to-all traffic between VMs [10]. As opposed to Scenario I, in application-specific private DCs, a high number of VMs communicate with each other, e.g., bank transactions between any two customers. Thus, in this scenario we consider a more general data communication pattern between VMs, assuming that half of the VMs in the DC communicate with each other. The communicating VMs are randomly selected using a uniform distribution. Then, each selected VM is set to exchange data with any other 50% of the VMs, also selected according to a uniform distribution. We analyze the network traffic through different layers as it is considered in the heuristic, ML, and hyper-heuristic methods. Moreover, we increase the number of VMs from 200 to 1000, to compare these methods in a large-scale scenario for a time horizon of one week.

X. RESULTS - OPTIMALITY ASSESSMENT (SCENARIO I)

In this section we compare the total energy consumption of DC, QoS (violations caused by overutilized servers), network traffic, number of migrations, and execution time of the algorithms for eight different methods:

- Correlation-aware VM Allocation (CVMA) [7] that is the best in its class to optimize energy consumption.
- Heuristic: our proposed heuristic (Sect. VI).
- Heuristic-Cap: for a realistic and fair comparison with ILP-based methods, we reduce the servers capacity to 80% during the VM allocation phase to guarantee that no violation occurs due to miss-prediction. This selected cap empirically represents a trade-off between energy efficiency and violation compared to Heuristic. We also assume that all the active servers use the maximum frequency level; i.e. 2.4 GHz (100%), to compute violations.
- ML: the proposed ML algorithm (Sect. VII).
- ML-Cap: the proposed ML algorithm with 80% servers capacity cap and setting maximum frequency level.
- ILP-Power: the proposed ILP-based method for DC energy optimization (Sect. V-A).
- ILP-Data: the proposed ILP-based method for data communication optimization (Sect. V-B).

A. Energy efficiency analysis

Figure 5 shows the energy consumption breakdown of the DC including both IT and cooling components. As a result of turning-off more servers, all approaches show an overall energy improvement higher than ILP-Data; on the contrary, ILP-Data further reduces (up to 66%) network energy, as it turns off switches for longer periods of time. Heuristic and ML exhibit less than 2% energy savings compared to CVMA; while providing 10 and 9% improvements compared to GH, respectively. This is because CVMA only considers CPU-load correlation between VMs; but, GH allocates the VMs with high data correlations to fewer servers. On the other hand, ILP-Power only improves up to 5% the results of ML by optimizing the number of active servers. In general, Heuristic-Cap and ML-Cap lead to higher energy consumption, due to the conservative capping approach, that increases the number of used servers.
B. QoS - Analysis of violations

Figure 6 (right y-axis) shows the total number of violations, defined as the number of overutilized servers during one day. Heuristic provides a drastic reduction of the violations, from 30 to 87% in worst and best cases compared to ML, respectively. This is because, in the ML approach for a low number of VMs, most of the classes are empty after classification. Therefore, to fill up the servers, ML chooses one VM from the nearest non-empty class, decreasing classification accuracy and leading to violation. On the other hand, Heuristic and ML achieve 94 and 65% improvements compared to CVMA, respectively. GH drastically decreases the number of violations in comparison with the other approaches due to partially filling up the servers. Due to the nature of the ILP-Power, ILP-Data, Heuristic-Cap and ML-Cap, these methods do not present any violation. Thus, they are not shown in Fig. 6.

Figure 6 (left y-axis) shows the average and worst-case amount of overutilized servers for a time horizon of one day, which determines the degree by which the negotiated QoS requirements can be violated. Basically, quality degradation is observed due to the miss-predictions, especially during abrupt workload changes. The results show, for lower number of VMs, Heuristic and CVMA provide better worst-case violation reduction compared to ML. But, for higher number of VMs, the violation of ML remains below the violation of Heuristic and CVMA because classification accuracy is improved. As a result, we obtain 70 and 19% less violations on average and in the worst case for ML compared to Heuristic, and also 88 and 10% improvements compared to CVMA, respectively, for 150 VMs. This figure also shows that GH obtains better results compared to the other approaches due to the servers’ underutilizations.

C. Network traffic analysis - data communication

Figure 7 shows total amount of data exchanged among the servers. Results demonstrate that ILP-Data (optimal solution) reduces the traffic ≈41, 42 and 46x in best case and ≈19, 17 and 21x in the worst case compared to Heuristic, ML and ILP-Power, respectively. This is due to the fact that ILP-Data distributes the non-communicating groups of VMs to different servers to minimize traffic. However, the number of turned-on servers is increased, leading to higher energy consumption. GH achieves up to 3x less network traffic compared to Heuristic and ML since its goal is to minimize the network traffic. On the contrary, the other approaches first try to use the minimum number of servers and then minimize the data communication.

In this sense, heuristic and ML increase the capability of absorbing time-varying data communication between the servers compared to ILP-Power and CVMA. These results show up to 14, 11 and 3% improvements for ML compared to ILP-Power, CVMA and Heuristic, respectively.

D. Evaluating the number of migrations

Table III shows the total number of migrations for one day. CVMA reduces the number of migrations compared to other methods since this considers only CPU-load correlation. On the contrary, ILP-Power and ILP-Data have the highest number of migrations in order to find the optimal allocation solutions. In the best case, ML obtains up to 31 and 25% improvements in the number of migrations compared to GH and Heuristic, respectively, due to the trace classification strategy. Moreover, ML-Cap outperforms Heuristic-Cap by up to 23%.

E. Execution time of proposed algorithms

The proposed methods trade-off solution optimality by execution time. To obtain the results shown in Table IV, we run the VM allocation methods for all time slots in 1 day, and we compute its average. The execution time of ILP-based methods is the highest (> 2 hours in some cases), making runtime allocation unfeasible. On the other hand, ML is the fastest

| TABLE III: Total number of migrations for one day |
|-----------------|-----------------|-----------------|
| Method          | 50VMs           | 100VMs          | 150VMs          |
| CVMA            | 429             | 1271            | 2180            |
| GH              | 868             | 2038            | 3192            |
| Heuristic       | 715             | 1862            | 3003            |
| ML              | 619             | 1410            | 2251            |
| Heuristic-Cap   | 791             | 1914            | 3180            |
| ML-Cap          | 659             | 1561            | 2464            |
| ILP-Power       | 888             | 2160            | 3120            |
| ILP-Data        | 936             | 2352            | 3528            |
TABLE IV: Execution time (sec.) of the algorithms

<table>
<thead>
<tr>
<th>Method</th>
<th>50 VMs</th>
<th>100 VMs</th>
<th>150 VMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVMA</td>
<td>0.19</td>
<td>0.93</td>
<td>2.49</td>
</tr>
<tr>
<td>GH</td>
<td>0.005</td>
<td>0.026</td>
<td>0.073</td>
</tr>
<tr>
<td>Heuristics</td>
<td>0.969</td>
<td>3.907</td>
<td>12.897</td>
</tr>
<tr>
<td>MLs</td>
<td>0.003</td>
<td>0.005</td>
<td>0.009</td>
</tr>
<tr>
<td>ILPs</td>
<td>10.087</td>
<td>287.694</td>
<td>8619.48</td>
</tr>
</tbody>
</table>

algorithm (<10 ms in the worst case) making it particularly suitable to solve large-scale problems.

XI. RESULTS - LARGE-SCALE SCENARIO (SCENARIO II)

In this section we show, for the same metrics than in the previous case, a comparison between the heuristic, ML, hyper-heuristic (Hyper) methods, and the state-of-the-arts.

A. Energy efficiency analysis

Figure 8 shows that heuristic, ML and Hyper reach almost similar results for energy consumption (< 2%). Hyper provides better energy savings compared to ML by selecting the heuristic method in some time slots where heuristic dramatically outperforms ML in terms of energy consumption. We observe up to 35 and 34% energy improvements for proposed heuristic and ML algorithms compared to Heuristic-Cap and ML-Cap, respectively, for 800 VMs. For larger scenarios, above 800 VMs, we need to turn-on a new rack and, thus, the second aggregation switch and the core router. Thereby, energy consumption increases due to the higher network energy consumption. Also, heuristic and ML result in high energy savings compared to GH and CVMA, reducing the number of active servers when the total demand of co-located VMs nearly reaches their server capacity during period.

B. QoS - Analysis of violations

Figure 9 (right y-axis) shows that ML provides a violation reduction, up to 15 and 63% compared to Heuristic and CVMA, starting from 400 VMs. On the other hand, for the lower number of VMs, Heuristic performs better than ML. In order to provide a better trade-off, Hyper reduces the number of violations by 11% compared to Heuristic while decreasing the energy consumption compared to ML. Moreover, GH performs better since it is not fully utilizing CPU resources which is not a case for energy efficiency. Also, Heuristic-Cap and ML-Cap reduce the number of violations dramatically because of the cap set on server load.

Figure 9 (left y-axis) shows that Heuristic outperforms ML in terms of the amount of violations for 200 VMs, reaching up to 18% improvement in the worst case. As the number of VMs increases, e.g. 800 and 1000 VMs, the violations of ML get closer to Heuristic and GH, inverting the trend for 600 VMs. This is because ML has less control on the worst-case violation during peak loads. Hyper outperforms ML and Heuristic by up to 23 and 14% due to selecting the best method per time slot with lower violation, while leading to only up to 8% overhead compared to both approaches over all cases.

In average, ML provides better results than the other approaches by managing the off-peak VMs load. Finally, for all the cases, Hyper is able to exploit the strengths of Heuristic and ML for providing intermediate solutions.

C. Multi-layer network traffic analysis

Figure 10 shows the total traffic through the ToR, aggregation-layer switches, and core router. From 200 to 800 VMs, when the core router is turned off, the results demonstrate that ML reduces the ToRs traffic up to 9% compared to Heuristic; while, Heuristic improves the aggregation-layer up to 4%. Note that for 200 and 800 VMs, traffic in the aggregation and core layers is very low for ML, while for Heuristic they are zero. This increase is due to turning on a server in a new rack. For 1000 VMs, ML provides less ToR and aggregation-layer traffic, but higher core traffic compared to Heuristic. Basically, Heuristic results in lower traffic in the upper layers of the network, due to its fine-tuning capabilities. Comparing the ML to CVMA and GH, we obtain significant improvements especially in upper layers for higher number of VMs, when CVMA and GH are less sensitive to dynamic environments, and their benefits become limited for large problems. By using Hyper, we achieve up to 5 and 7% improvements in ToR and aggregation compared to Heuristic; while 6 and 9% overheads compared to ML, respectively. For the core layer, Hyper improves 53% compared to ML, but presents an overhead of 16% compared to Heuristic.

Following the same trend, Heuristic-Cap outperforms ML-Cap in upper network layers. Differently, we need to turn on the core router for both methods, starting from 800 VMs, due to setting a conservative cap and consequently turning on a server from a new group of racks connected to the aggregation switch.

D. Evaluating the number of migrations

Table V represents the total number of migrations for one week. ML reduces the number of migrations by up to 36, 40, and 39% compared to CVMA, GH, and Heuristic, respectively. This happens because in ML, the classification
accuracy increases for higher number of VMs, thus allowing to place together better candidate VMs. In addition, ML-Cap improves this metric by up to 35% compared to Heuristic-Cap. On the other hand, Hyper achieves 13% improvement and 20% overhead on average over different number of VMs, trading-off the benefits of Heuristic and ML. In the worst case, i.e. 1000 VMs, we only migrate 4.8 and 6.5% of total number of VMs on average every sample for ML and Hyper, respectively, that does not lead to high live migration overhead.

E. Computational overhead (execution time) and discussion

Table VI shows the average execution time of the proposed VM allocation methods for one week of traces. The results follow the same trend than for the small-scale scenario, with Hyper exhibiting a trade-off between the execution time of Heuristic and ML. In summary, the ML method provides almost the same energy efficiency, but dramatically lowers computational overhead when compared to Heuristic, while the heuristic method obtains better network traffic and QoS.

This is because heuristic methods allow more fine-tuning on the allocation, but present a larger computational overhead, which makes them unsuitable for large-scale scenarios. Our results show that Hyper ensures a good trade-off between solution quality (energy, QoS, and network traffic) using the benefits of both Heuristic and ML approaches.

XII. Conclusion

In this paper we have proposed a two-phase greedy heuristic and a ML method to tackle the challenge of energy- and network-aware VM allocation, evaluating them in terms of energy, network traffic, QoS, migrations and scalability. We have compared them to the optimal solutions (implemented using ILP), and to two algorithms in the state-of-the-art that are the best in their areas. We have presented, for the first time in literature, a novel multi-objective hyper-heuristic method for the VM allocation problem able to find better solutions by leveraging the strengths of heuristic and ML methods, while allowing users to decide on the importance of each metric. Our experimental results have shown that heuristic and ML methods reach similar results in energy consumption (<2% difference), consuming only up to 6% more energy than the optimal solution. The ML approach obtains up to 24% server-to-server network traffic improvements when compared to all other methods, and achieving execution time speed-up up to 480x for large-scale problems. On the other hand, the heuristic algorithm results in better QoS and lower traffic in the upper layers of the network structure, due to its fine-tuning capabilities. Finally, the hyper-heuristic algorithm integrates the benefits of heuristic and ML to ensure a good trade-off between solution quality and computational overhead.

### REFERENCES


---

### TABLE VI: Execution time (sec.) of the proposed algorithms under different number of VMs

<table>
<thead>
<tr>
<th>Methods</th>
<th>200</th>
<th>400</th>
<th>600</th>
<th>800</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVMA</td>
<td>4.45</td>
<td>28.5</td>
<td>88.3</td>
<td>200.6</td>
<td>471.6</td>
</tr>
<tr>
<td>GH</td>
<td>1.01</td>
<td>4.12</td>
<td>9.65</td>
<td>17.49</td>
<td>28.6</td>
</tr>
<tr>
<td>Heuristics</td>
<td>20.3</td>
<td>88.8</td>
<td>188.7</td>
<td>341.4</td>
<td>519.9</td>
</tr>
<tr>
<td>MLs</td>
<td>0.047</td>
<td>0.19</td>
<td>0.421</td>
<td>0.711</td>
<td>1.107</td>
</tr>
<tr>
<td>Hyper</td>
<td>10.4</td>
<td>60.86</td>
<td>102.1</td>
<td>199.72</td>
<td>282.35</td>
</tr>
</tbody>
</table>

---

### TABLE V: Total number of migrations for one week

<table>
<thead>
<tr>
<th>Method</th>
<th>200VMs</th>
<th>400VMs</th>
<th>600VMs</th>
<th>800VMs</th>
<th>1000VMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVMA</td>
<td>30072</td>
<td>63578</td>
<td>96836</td>
<td>130260</td>
<td>163643</td>
</tr>
<tr>
<td>GH</td>
<td>22539</td>
<td>47437</td>
<td>65690</td>
<td>78726</td>
<td>98407</td>
</tr>
<tr>
<td>Heuristic</td>
<td>28608</td>
<td>61724</td>
<td>95014</td>
<td>128540</td>
<td>158635</td>
</tr>
<tr>
<td>ML</td>
<td>22539</td>
<td>47437</td>
<td>65690</td>
<td>78726</td>
<td>98407</td>
</tr>
<tr>
<td>Heuristic-Cap</td>
<td>29406</td>
<td>62607</td>
<td>96061</td>
<td>129465</td>
<td>163007</td>
</tr>
<tr>
<td>ML-Cap</td>
<td>23929</td>
<td>49380</td>
<td>69299</td>
<td>86689</td>
<td>105262</td>
</tr>
<tr>
<td>Hyper</td>
<td>25609</td>
<td>57471</td>
<td>83143</td>
<td>111935</td>
<td>132822</td>
</tr>
</tbody>
</table>

---

### Fig. 10: Network traffic of (a) ToR, (b) Aggregation-layer switches and (c) core router for one week.


