Efficient Network Slicing Orchestrator for 5G Networks using a Genetic Algorithm-based Scheduler with Kubernetes: Experimental Insights

Massinissa Ait Aba, Maya Kassis, Maxime Elkael, Andrea Araldo, Ali Al Khansa, Hind Castel-Taleb, Badii Jouaber

Efficient Network Slicing Orchestrator for 5G Networks using a Genetic Algorithm-based Scheduler with Kubernetes: Experimental Insights

Massinissa Ait Aba
Davidson Consulting
Paris, France
Massinissa.ait-aba@davidson.fr

Maya Kassis
Paris, France
Maya.kassis4@gmail.com

Maxime Elkael
SAMOVAR, Telecom SudParis, Institut Polytechnique de Paris
maxime.elkael@gmail.com

Andrea Araldo
SAMOVAR, Telecom SudParis, Institut Polytechnique de Paris
andrea.araldo@telecom-sudparis.eu

Ali Al Khansa
SAMOVAR, Telecom SudParis, Institut Polytechnique de Paris
maxime.elkael@gmail.com

Hind Castel-Taleb
SAMOVAR, Telecom SudParis, Institut Polytechnique de Paris
hind.castel@telecom-sudparis.eu

Badii Jouaber
SAMOVAR, Telecom SudParis, Institut Polytechnique de Paris
badii.jouaber@telecom-sudparis.eu

Abstract

In 5G networks, physical resources can be virtualized and allocated to separate virtual networks (or network slices), with distinct requirements. The Virtual Network Embedding (VNE) problem consists in finding the optimal mapping of virtual resources (virtual links and nodes) onto a physical infrastructure. A recent trend consists in virtualizing 5G networks using Kubernates (K8s), a popular virtualization technology.

In this paper we perform an experimental study to show the limit of using the standard K8s deployment strategy when dealing with dynamically arriving slices in a heavy loaded setting. By deploying the virtual components of a slice one by one, standard K8s is prone to wasting resources and energy due to partially deploying slices that, at the end, are found to be infeasible, due to lack of available resources. We propose an alternative K8s deployment strategy that first solves VNE via a Genetic Algorithm and then, for each slice, deploys either all its components or none. Our experimental results show a notable improvement in slice acceptance, energy efficiency and deployment time. Our work shows that it is necessary to adapt cloud native technologies to the specific requirements of telecommunication scenarios, as they are different from the cloud ones for which such technologies were originally developed.

Keywords— 5G, Network Slicing, Virtualization, Kubernetes, Energy Consumption, Acceptance Ratio, OAI, UERANSIM

1 Introduction

Network slicing consists in creating multiple independent virtual networks, called *slices* on top of a shared physical infrastructure. Each slice can have different requirements and be owned by different tenants. Network virtualization and slicing can leverage containerization. *Containers* package an application along with its dependencies into a single portable unit. The different software components of a slice can thus be packaged into containers. The network operator, owning the physical infrastructure, aims to determine the placement of the slices's components to appropriate physical nodes, in order to maximize the number of deployed slices while minimizing the overall energy

Table 1: Experimental Approaches

References	Experiment/Approach	Focus and Contributions
[1]	Containerized Applications	Designing applications for slices using K8s and Amazon services.
[2]	5G Core Emulation Platform	Experimental framework on K8s for 5G core network.
[3]	Greedy Algorithm	Deploying 5G network slices using the greedy algorithm.
[4]	VM Deployment and Scheduling	Experiments involving scheduling slices on different VM flavors.
[5]	Prototype 4G Core on K8s	Implementing 4G core networks as slices on K8s with scheduling.
[6]	Cloud-Native Telecom Functions	Introducing cloud-native telecom functions on K8s.
This Paper	Scheduling 5G VNFs using a Genetic Algorithm (GA)	Leveraging K8s for network slicing, experiments, and 5G core integration, considering the entirety of pods within the slice during the orchestration phase.

consumption, subject to physical resource capacity and slices' requirements. This problem is referred to as Virtual Network Embedding (VNE). Kubernetes (K8s) has been recently used as a virtualization technology for slicing [6]. In this framework, a slice consists of a set of virtualized network functions, each packaged into a *container* running inside a *pod*. Several slices arrive and depart over time. The deployment strategy is implemented into a K8s *scheduler*, which is the agent that decides in which physical node each pod of each slice must be placed.

We pinpoint that K8s has been developed for cloud environments, where resources are abundant and can be practically assumed unconstrained, since one can get as much resource as needed as long as one pays for it. In the cloud, the main question is thus *where to place* pods. However, such assumption may not hold in a telecommunication scenario, where network operators may not have the economic ability or willingness to over-dimension the deployed physical resources. In this case, in certain periods resources might be too scarce to accommodate all arriving slices. Therefore, the question of *whether it is possible to place* all pods of a slice arises. This imposes a change in the standard K8s scheduling discipline. While the standard K8s scheduler places the different pods of a slice one after another, we propose a strategy that takes a comprehensive view of all pods within a slice and decides whether to place them all (and where) or to place none. The decision is based on a Genetic Algorithm (GA).

We emphasize that the core idea of our approach is quite simple: before starting to deploy a slice, the scheduler must compute a feasible placement for the entire slice. If it does not exist, it does not deploy any of its pods (as standard K8s would do), as this would just waste time, energy and resources, which could instead be used for pods of other feasible slices arriving in the near future. Therefore, the contribution of this paper does not lie in some intricate algorithmic solution. It instead mainly consists in showing, in an experimental setting, that the re-use of a Cloud technology, such a K8s (popular nowadays for virtualizing 5G networks) requires an appropriate adaptation to be effective in telecommunication scenarios. The adaptation we propose is materialized in our scheduler, which brings improvement in energy efficiency, slice acceptance and deployment time, also thanks to intelligent placement decisions based on a GA. We release the code of the scheduler and the experiments as open source.[1]

The paper is organized as follows: Sec. 2 discuss the related work. Sec. 3 presents the detailed virtualization environment. In Sec. 4, we discuss slice deployment on a physical infrastructure and describe the proposed scheduling strategy. Sec. 5 shows our experiments and Sec. 6 concludes the paper.

2 State of the Art

In this section, we provide an overview of the existing research in the realm of network slicing and VNE. The first subsection discusses practical experiments and empirical investigations in the context of network slicing orchestration. The second subsection delves into the theoretical underpinnings of VNE algorithms.

[1] https://github.com/AIDY-F2N/setpod-scheduler

Table 2: Summary of Related Works in VNE

Methodology or Algorithm	References	Focus and Contributions
ILP, MILP	[7], [8]	Optimization algorithms for VNE, explores ILP and MILP formulations.
Genetic Algorithm	[9], [10], this work	Application of Genetic Algorithm for near-optimal VNE.
Simulated Annealing	[11]	Utilizing Simulated Annealing to tackle VNE challenges.
Ant Colony Optimization	[12], [13]	Investigating Ant Colony Optimization for VNE optimization.
Survey	[14], [15], [16]	Early and extended surveys providing insights into VNE problem.
Reinforcement Learning	[17],, [18]	Exploration of Reinforcement Learning for addressing VNE.
Graph Theory	[19]	Usage of graph theory-based solutions for addressing VNE.

2.1 Experimental Work

Two papers describe, from a system-level point of view, how network slicing can be realized using K8s. The platform described in [6] is able to host cloud-native telecom functions, using K8s and Openshift Operator, and to perform 5G network automation. in a cloud environment. Similar to [6], our previous preliminary paper [5] describes how to prototype 4G core network slices on a K8s cluster. Since the aforementioned two articles focus on the architectural and procedural aspects, they do not provide any performance evaluation.

In [1], an approach to realize network slicing based on K8s and Amazon's services is presented, together with a scaling algorithm. The authors of [2] introduced an experimental framework using an emulation platform with the 5G core architecture running on a K8s cluster. Their work highlighted the monitoring and lifecycle management of 5G networks. While we also use K8s for containerizing a 5G network (using the Open Air Interface (OAI) code [20]), we focus on scheduling algorithms rather than scaling or lifecycle management strategies, i.e., we are interested in correctly deciding whether to deploy a slice and, if yes, in which nodes to deploy each of its components. A greedy algorithm for mapping Virtual Network Functions (VNFs) on physical machines is presented in [3]. Evaluation is performed in simulation, which cannot capture the waste of energy and deployment time related to the scheduler decision, which is instead central to our work.

In [4], the authors conduct a series of experiments to test different virtual machines' capabilities that may suit different slices. Slices are deployed on different virtual machine flavors using OAI 4G Evolved Packet Core (EPC) code, including the radio part and Commercial Off The Shelf (COTS) UEs. However, in their work slices are considered fixed, while we assume a dynamic scenario, where slices arrive and depart. It is in such scenario that we can assess the unsuitability of standard K8s scheduling for a resource constrained telecommunication operator.

Previous experimental efforts are summarized in Table 1. We observe that experimental work on the performance of scheduling, in terms of energy, time-of-deployment and slice acceptance, in a scenario with dynamic slice arrivals and departures and in a cloud-native environment is missing. This paper aims to fill this gap.

2.2 Virtual Network Embedding (VNE)

Virtual Network Embedding (VNE) consists in finding the optimal mapping of virtual network functions (VNFs) and the virtual links connecting them onto a physical network. A VNF is mapped onto a physical node (representing, for instance, a machine) and a virtual link is mapped into a path of physical links. The objective is usually to minimize resource usage, i.e., bandwidth and/or CPU cycles [18], or energy [21]. The mapping should satisfy the requirements of the slices, which can be expressed in terms of minimum resources to be allocated [18] or in terms of minimum latency requirements [21]. Moreover, the CPU and the bandwidth capacities of the physical nodes and links must not be exceeded.

Among the large corpus of surveys on VNE, we bring to the reader's attention some representative ones, such as [14–16].

The state-of-the-art solutions for VNE problem are mainly based on optimization algorithms such as Integer Linear Programming (ILP) [7], Mixed Integer Linear Programming (MILP) [8]. However, the VNE problem is proven to be NP-hard [14, 16], which precludes exact solutions in large scale cases. Metaheuristics such as Genetic Algorithms (GAs) [9, 10], Simulated Annealing (SA) [11]

and Ant Colony Optimization (ACO) [12,13] are thus employed to find near-optimal mappings. Other solutions are based on graph theory [19]. Recently, Montecarlo Tree Search [18] and Reinforcement Learning [17] have been applied to VNE.

Observe that in VNE, slice tenants are usually assumed to specify a-priori the amount of CPU and bandwidth they require to be allocated to their virtual functions and virtual links. This is also the assumption we will adopt in this paper. It is also possible to assume, instead, that tenants just require a certain latency requirements to be satisfied and that it is up to the network operator to decide how much CPU cycles and bandwidth to allocate to each virtual components in order to meet such requirements. This point of view, complementary to VNE, is adopted in [22].

Table 2 summarizes the work on VNE.

3 Virtualization environment

Our work is based on K8s. K8s is a widely used and popular orchestration platform for managing containerized applications. Other container orchestration tools are available, such as Docker Swarm, Apache Mesos, and Amazon ECS. K8s may be preferred over the others due its large community and ecosystem and its support for multi-cloud or hybrid cloud deployments. Although K8s is the orchestrator to which we refer, this choice does not represent a limit in the generality of the proposed approach, which could be replicated in other platforms.

In K8s, a node is a worker machine that runs containerized applications. It is a physical or virtual machine that is part of a cluster managed by the K8s control plane. Each node in a physical network infrastructure (K8s cluster) requires resources like CPU, memory, and storage, and also incurs operational costs such as electricity, cooling, and maintenance. Nodes are responsible for running one or more containers, which are packaged and deployed as pods. A pod in K8s is the smallest and simplest unit in the K8s object model. It represents a single instance of a running process in a cluster and can contain one or more containers that share the same network namespace and storage volumes and are deployed on the same host. Containers within a pod are scheduled to run on the same physical or virtual machine. Pods provide an isolated environment for containers to run in, encapsulating the application's processes, storage, and networking. Pods can be created, updated, and scaled by using K8s manifests, which are configuration files written in YAML or JSON format. Pods can be automatically rescheduled on different nodes in case of node failure or during scaling operations. K8s manages pods by scheduling them to appropriate nodes, monitoring their health, scaling them, managing their networking, handling updates and rollbacks, and performing clean-up tasks during termination.

In a network slicing scenario, where a single physical network infrastructure is partitioned into multiple virtual networks (slices), in order to cater to different services or applications, pods can be used to deploy and manage containerized applications within those slices. The pods can be configured with the necessary resources, such as CPU and memory requests, based on the service requirements of the network slice. The CPU request specifies the minimum amount of CPU required for the pod to run, while the memory request specifies the minimum amount of memory required. These requests are used by the K8s scheduler to ensure that pods are deployed on nodes with sufficienrandomlyt reserved resources to run them. By reserving the requested resources on the node, the scheduler ensures that the pod has access to those resources when it is scheduled to run. Additionally, pods can be used to enable multi-tenancy within network slices, where different tenants or users can have their own isolated containerized applications running within separate pods while sharing the same network slice infrastructure. This allows for efficient resource utilization, isolation of tenant-specific applications, and simplified management of containerized applications within network slices.

In this work, we assume that a slice consists of a set of pods that host network functions. Pods may have dependencies that require a specific order of deployment to ensure proper communication between them. We assume that we have several slices to deploy on a physical network infrastructure managed by a network operator. A slice is deployed if all the pods of the slice have been deployed in the physical network infrastructure, respecting the appropriate order of deployment and the CPU and memory requests. Our objective is to maximize the acceptance ratio by deploying as many slices as possible while minimizing the energy consumption of the physical network infrastructure. It is worth noting that modeling the problem of network slicing can vary depending on the specific requirements, constraints, and goals of the network slicing scenario, and may require further customization or refinement based on the specific context of the network infrastructure and services being provided. Furthermore, the optimization of the network slicing process may be based on other goals, such as minimizing resource utilization and minimizing latency.

4 Deploying slices on a K8s cluster

In this section, we discuss slice deployment on a physical network infrastructure and describe the K8s standard scheduler and our proposed scheduler. Kube-scheduler is a key component of the K8s control plane responsible for scheduling pods to run on specific nodes within a K8s

cluster. It ensures that pods are deployed to appropriate nodes based on the defined constraints, such as resource requirements, affinity rules, and other scheduling policies. In the following, the default scheduling algorithm used by Kube-scheduler is presented.

4.1 Default K8s scheduler

The default scheduling algorithm used by Kube-scheduler in K8s is known as the "default scheduler" or "predicate-based scheduler" [23]. This algorithm follows a two-step process:

1. Predicates: During the first step, the scheduler applies a series of predefined filtering rules called "predicates" to determine which nodes are eligible for placing a pod. Predicates consider factors such as resource requirements (e.g., CPU, memory), node conditions (e.g., node capacity, node readiness), and other scheduling constraints (e.g., node selectors, taints/tolerations) [24].

2. Priorities: After applying predicates, the scheduler assigns a priority score to each eligible node based on a set of predefined rules called "priorities". Priorities are used to rank the eligible nodes by order of preference.

After ranking the nodes, the scheduler selects the highest-scoring node as the optimal placement for the pod. If multiple nodes have the same highest score, the scheduler may use additional tie-breaking rules to make the final decision [23].

4.2 Disadvantage of deploying a slice pod by pod

For a given slice of *ns* pods, the "default scheduler" deploys the slice pod by pod, deciding for each pod the node on which it will be deployed independently of the other pods. When deploying a given slice using the "default scheduler", some problems may arise:

- *Inappropriate resource exploitation*: Deploying pod by pod can lead to inappropriate resource exploitation. Indeed, because the scheduler makes decisions based on incomplete information, local optima are easily reached. The placement of a pod may seem optimal in terms of resource usage at a given instant, but it may turn out to be inefficient when other pods need to be scheduled in the future and may lead to the impossibility of scheduling a full slice using the remaining resources. For example, assuming we want to deploy a slice consisting of 4 pods to a platform of two physical nodes, each with the capacity to provide 8 cores. Pods 1 and 2 have a CPU request of 2 cores, and pods 3 and 4 have a request of 5 cores. Figure 1 shows a first deployment, pods 1 and 2 are deployed on node 1 and pod 3 on node 2.

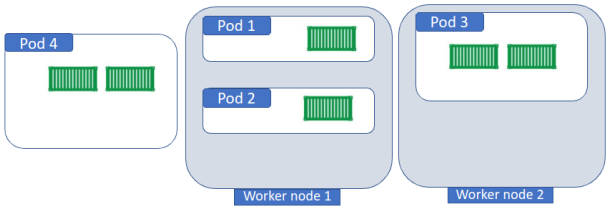

Figure 1: Deployment of 3 pods out of 4.

The new CPU capacities of the two nodes after deploying the first 3 pods are 4 cores for node 1 and 3 cores for node 2. The deployment of pod 4 is therefore impossible because neither of the two nodes has the 5-core capacity required by pod 4. A better assignment of pods to nodes would be the deployment shown in Figure 2. The remaining CPU available on the two nodes after deploying the slices is 1 core for node 1 and 1 core for node 2.

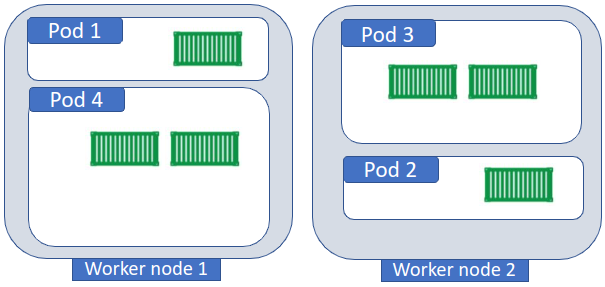

Figure 2: Deployment of 4 pods out of 4.

This example shows that the "default scheduler" focuses only on the pod level. However, such a method, no matter what strategy is used, can lead to sub-optimal solutions from the perspective of a slice deployment. In other words, a feasible slice deployment might need to consider the deployment of all the pods within a slice holistically.

- More nodes than necessary are used: Deploying pod by pod may lead to the utilization of more nodes than necessary, because the choice of deploying some pods can saturate the resources of some nodes in a way that the rest of the pods of the slice will need other nodes for their deployments, thus increasing the overall cost of operating the cluster. In the example shown in Figure 1, pod 4 would need a 3rd node for its deployment, while it would have been possible to deploy all the pods using only two nodes (Figure 2).

- Deployment of unnecessary pods: a slice is deployed if all its pods are deployed. However, by deploying the slice pod by pod, it is possible that after the deployment of the first pods, the available resources will not be sufficient for the deployment of the remaining pods, the slice is then refused. However, some pods have already been deployed and thus have to be deleted as they are not needed by any valid slice. This implies a waste of time and unnecessary energy consumption related to the deployed pods.

4.3 Proposed scheduler

K8s uses a scheduling framework that allows different scheduling algorithms to be implemented and used with the Kube-scheduler component. To avoid the above-mentioned problems, we have proposed a new scheduler. Unlike the "default scheduler", the proposed scheduler first checks whether it is possible to find a feasible assignment of the pods to the cluster nodes respecting resources and labels constraints. Only if such an assignment is found, the scheduler starts the deployment of the pods composing the slice. Otherwise, it rejects the slice and no pod is deployed.

4.4 Virtual Network Embedding problem

We consider a simplified VNE model and we do not consider the links between pods and the links between nodes. Scheduling pods on a K8s cluster can thus be viewed as a bin packing problem [25] where the main goal is to deploy a set of items (the pods of the slices in our case) using a minimum number of bins (physical nodes in our case). However, while bin packing problems are generally defined in a static scenario, where all the items to be placed are known in advance, we consider instead a dynamic scenario, where slices (which can be viewed as a set of pods) arrive and depart. In such a dynamic scenario, our goal is to maximize the acceptance ratio. The acceptance ratio is 1 minus the rejection ratio, which is the fraction of slices that have arrived but have not been placed. This occurs when no mapping of virtual nodes onto physical nodes has been found that is feasible, i.e., that respects the node capacity constraints.

4.5 Genetic algorithm run by the proposed scheduler

The algorithm used for pod assignment to the nodes is a Genetic Algorithm (GA). GA is a metaheuristic that takes inspiration from the biological processes that drive evolution. We chose GA since they are powerful in optimizing complex problems and also relatively easy to implement. Their ability to efficiently explore large solution spaces makes them an ideal choice for tackling intricate optimization challenges with impressive runtime performance. Recall that the main focus of this paper is not in the algorithm used for deciding pod mapping, rather it is on studying a deployment discipline, alternative to standard K8s, which takes into consideration the requirements of the entire slice before deploying nodes one after another. In order to study such deployment discipline, it is sufficient to adopt any intelligent algorithm to decide mapping, and GA was a perfect fit for this. Other work may focus, in the future, in replacing GA with any other mapping algorithm, complex or advanced at will. The details of each step are explained in the following.

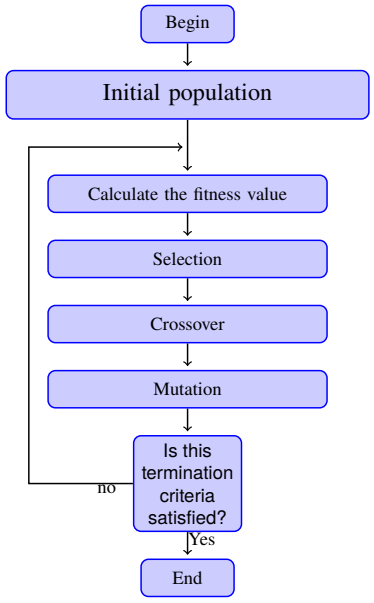

Figure 3: The different steps of a classical genetic algorithm.

4.5.1 Initial population

We define a chromosome as a solution that contains an assignment of each pod to a physical node. The selection of the initial population of chromosomes is a very important step for the GA, which could significantly impact the solutions obtained in the next steps. A chromosome is built by following two steps:

- Firstly, we select the set of possible physical nodes for each pod. To be selected, the physical node must have sufficient available resources, such as CPU and memory, and must have the necessary software dependencies installed to support the containers of the pod. The physical node must not already be running any other pods that would cause conflicts or exceed its capacity and must be labeled with the appropriate labels that match the pod's selector so that it can

be scheduled to the correct node. Labels in K8s are key-value pairs that are attached to resources such as nodes, pods, services, and deployments. They are used to organize and categorize resources in a meaningful way, making it easier to manage and manipulate them. Labels can be used to select a specific set of resources based on their characteristics, such as their role, environment, version, or owner. A default number of labels is defined by K8s for each pod (hostname, role, OS, etc.). If a pod specification includes a node selector or specific rules that require specific labels to be present on the node, the scheduler will only consider nodes that have those labels as candidates for scheduling the pod. Finally, the set of selected nodes must be in a healthy state and able to communicate with the rest of the cluster to ensure the proper functioning of the pod.

- Secondly, we assign each pod to exactly one physical node randomly using a uniform probability distribution.

4.5.2 Calculate the fitness value

The fitness function takes a chromosome as input and evaluates how "fit" or "good" the solution is with respect to the different constraints of the problem. The fitness value of each chromosome is calculated according to the number of pods that can be deployed without violating the resource constraints, and the number of labels of the selected nodes to which each pod is deployed. For each chromosome ch_k, first, we calculate $n(ch_k)$ the number of pods that can be deployed according to the assignment defined in the previous step without violating the resource constraints. If all pods can be deployed, the chromosome represents a feasible solution. Secondly, for the $n(ch_k)$ pods that can be deployed without violating the resource constraint, we compute $l(ch_k)$ the sum of the labels of the selected nodes for their deployment. For each chromosome ch_k, the fitness value $f(ch_k)$ is then defined as follows:

$$f(ch_k) = n(ch_k) + \frac{1}{l(ch_k)}. \quad (1)$$

We consider that the best chromosome is the one with the highest value $f(ch_k)$. So, if several chromosomes represent feasible solutions, the chromosome that uses nodes with the minimum number of labels is the best. The idea is to minimize the usage of nodes that have a high pod coverage (that can host pods with many labels), in order to maintain sufficient specific resources for future slices.

4.5.3 Parent selection

Parent selection is the process of selecting the fittest chromosomes and allowing them to pass their genes to the next generation. We select the best $\frac{N}{2}$ chromosomes as parents for a population of N chromosomes.

4.5.4 Crossover

Crossover is a genetic operation used to make the programming of some chromosomes vary from one generation to the next. Two chromosomes (parents) are selected using selection operators, while crossover sites are chosen randomly. The genes at these crossover sites are then exchanged, creating a completely new chromosome. From the best $\frac{N}{2}$ chromosomes previously selected, $\frac{N}{2}$ new chromosomes are generated following the steps outlined below:

- First, we form $\frac{N}{2}$ pairs of chromosomes from the best chromosomes randomly.
- For each pair, we randomly generate a crossover point c between 1 and ns, where ns is the number of pods in the slice. We then concatenate the c first selected physical nodes of the first parent and the last $ns - c$ selected physical nodes of the second parent to form the selected physical nodes of the new chromosome.

4.5.5 Mutation

Mutation occurs to maintain diversity within the population and prevent premature convergence. The key idea is to insert random genes into offspring to maintain population diversity. At each iteration, we generate a random number k between 1 and $\frac{N}{4}$. Then, k chromosomes are selected randomly from the new population to be modified in this iteration. For each selected chromosome, $\frac{ns}{4}$ mutation pods are chosen randomly, and the selected physical nodes of these pods are modified (generating a new node assignment for each pod), thus creating a new chromosome.

4.5.6 Termination

First, we check whether the set of possible physical nodes for each pod contains at least one physical node. If this is not the case, we stop the algorithm because it is impossible to find a feasible solution. Otherwise, after selecting the initial population, the different steps of the algorithm (Fitness value, Selection, Crossover, Mutation) are executed over It iterations. The algorithm terminates when we reach the number of iterations It. For a slice of ns pods and a cluster of nc nodes, the population size N is set to $N = VAR \times nc \times ns$, and the number of iterations It is set to $It = 2 \times VAR \times nc \times ns = 2N$. The parameter VAR plays a crucial role in determining the trade-off between solution quality and computation time. By default, VAR is set to 15. Adjusting the value of VAR allows for a fine-tuning of the algorithm's behavior. If VAR is set

to a larger value, the genetic algorithm may yield a more accurate solution due to an increased exploration of the solution space. However, it is important to note that this may come at the cost of extended computation time. On the other hand, setting VAR to a smaller value can expedite the convergence of the algorithm, leading to faster results. Nevertheless, there is a potential risk that the algorithm may not find a good solution, as a smaller VAR may limit the exploration of the search space.

If a feasible solution is found, the scheduler deploys the slice pod by pod respecting the imposed deployment order. The proposed scheduler is designed to manage applications composed of a set or group of pods. We have named it Setpod-scheduler (check Algorithm 1).

Algorithm 1 GA-Based algorithm run by Setpod-scheduler

Initial Populations: For each pod,

- Select the set of possible physical nodes.
- Assign the pod to exactly one physical node randomly.

if (each pod contains at least one physical node) **then**
  for ($i = 0$ to It) **do**
    Calculate the fitness value of each chromosome
    Select the best chromosomes as parents
    Perform crossover
    Perform mutation
  end for
else
  No feasible solution.

5 numerical results

To evaluate our scheduler, we compare Setpod-scheduler and Kube-scheduler in several scenarios, using a K8s cluster, and a 5G platform composed of a 3GPP-Compliant 5G core network and a simulated RAN. Figure 4 shows the experimental scheme, we detail each part in the following.

5.1 Description of physical network infrastructure

In our lab, the physical network infrastructure consists of a K8s cluster consisting of one master and two workers. Each worker consists of an Intel Core i7-6600U processor and 16GB of system memory. The master includes an 11th Gen Intel(R) Core(TM) i7-11850H processor, which is equipped with 16 cores, and features 32GB of system memory.

5.2 Description of the platform used

We deploy in our platform the code of the Open Air Interface (OAI 5G) Core Network [20]. All the features of the OAI 5G core network components are continuously tested with professional testers, commercial gNBs (with COTS UE), and open-source RAN simulators. Furthermore, we are considering slices with multiple pods, each pod representing a VNF of the 5G core. OAI code is open-source and cloud-native. It implements the 5G Release 16 Core Network virtual functions as pods on the top of K8s cluster [20]. A total of 8 pods ensure the operations of the OAI Core Network (MySQL, NRF, UDR, UDM, AUSF, AMF, SMF, SPGW-U) [26]. In addition, we connect the code of the UERANSIM simulator [27] to the 5G OAI Core Network. UERANSIM is the open source 5G UE and RAN (gNodeB) simulator considered as a 5G mobile phone and a base station in basic terms. We use it to generate the traffic inside the slice and measure the performance and energy consumption. The two worker nodes are used to run the OAI Core Networks. The master node is used to run the simulated GNB and the UEs of UERANSIM with traffic generated for each UE. A wattmeter is used to measure the power consumption of both workers, as shown in Figure 4. The used code is available online [28].

Nevertheless, due to the novelty of 5G and network slicing concepts, we are implementing a slice as a whole 5G core network including one control plane and one data plane. Therefore, the notion of multiple data paths sharing the same control plane is not considered in this paper.

5.3 Description of the emulation scenario

We compare Kube-scheduler and Setpod-scheduler across 50 distinct scenarios involving arrivals and departures of independent 12 slices. To generate the arrivals and departures of the slices, a uniform random variable x is sampled from the range $[0, 1]$. We introduce a parameter P to serve as a threshold for distinguishing between new arrivals and departures. Specifically, if $x > P$, we generate an 'arrival of a slice' event; otherwise, we generate a 'departure of a slice' event. The parameter P can be interpreted as the departures/arrival ratio. We vary P across different scenarios to demonstrate its impact on the acceptance rate.

Figure 4 shows a scenario containing the arrivals and departures of 5 slices. (0,'arrival',5) means that the slice number 5 arrives at time 0.

For each slice, we randomly generate the CPU request of each pod between $100m$(equivalent to 0.1 CPU or 100 milliCPU) and $300m$ using a uniform probability distribution. The memory request of each pod is generated between 128 Mebibytes and 512 Mebibytes using a uniform probability distribution. In practice, the CPU demands of pods in cloud or containerized environments

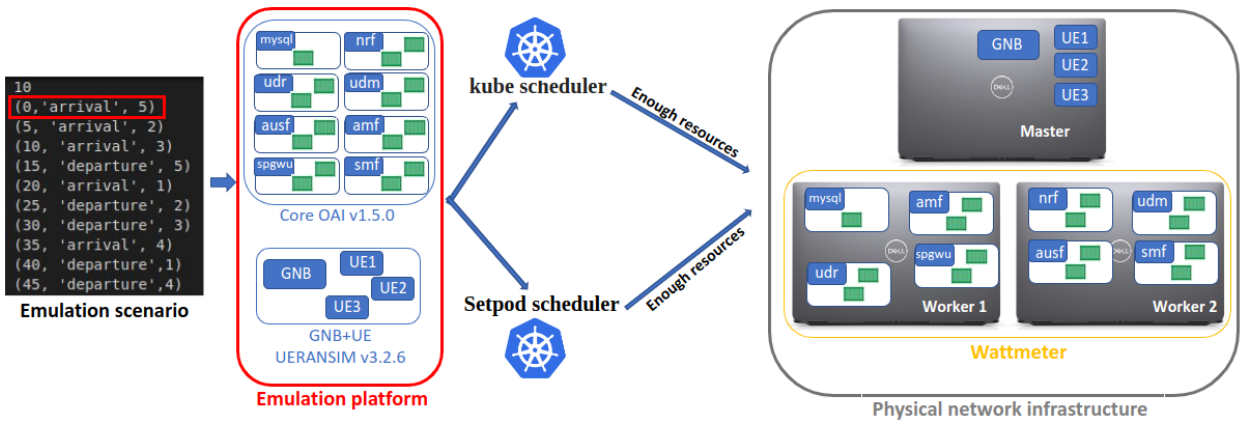

Figure 4: Experimental scheme used to compare the two schedulers.

often vary between minimal usage (e.g., for lightweight applications) and higher usage (e.g., for resource-intensive computations). Therefore, the range of $100m$ to $300m$ for CPU requests allows for the simulation of a diverse set of pod workloads, encompassing both lightweight and more computationally intensive applications. Similarly, for memory, the selected range of 128 Mebibytes to 512 Mebibytes aligns with the diverse memory requirements of pods in practical deployments. This range accommodates the needs of pods ranging from those with modest memory footprints to those with more substantial memory demands, reflecting the heterogeneity often encountered in real-world scenarios. The use of a uniform probability distribution within these specified ranges ensures that each potential value is equally likely, providing a fair representation of the spectrum of possible resource demands. This approach aims to capture the inherent variability in workload characteristics, contributing to a robust and realistic simulation environment for our experiments.

We set the VAR parameter value to 15 for the genetic algorithm. Leveraging just two nodes for deploying the core network of each slice, comprising 8 pods, the population size is equal to $N = 15 \times 2 \times 8 = 240$, and the number of iterations It is equal to $It = 2 \times N = 480$.

5.4 Experimental results

Figures 5, 6 and 7 show the comparison results between Kube-scheduler and Setpod-scheduler for $P = 1$, $P = 3$ and $P = 5$ respectively.

We employ three criteria for conducting comparisons:

- Acceptance ratio: The percentage of slices successfully deployed on the physical network infrastructure.
- Average energy consumption: The energy consumption generated by the deployment of a slice.
- Average deployment time: Comparison of the average time taken for deployment per slice.

We notice that by decreasing the value of P, the acceptance ratio decreases for both schedulers, moving from 19% for $P = 1$ to 64% for $P = 5$. This is quite logical, because the smaller the value of P, the higher the number of simultaneous slice arrivals. Deploying several slices without releasing them saturates the resources of both workers, which implies the rejection of several slices, thus giving a low acceptance ratio. The acceptance ratio of the two schedulers is identical for $P = 1$, with a small advantage for Setpod-scheduler for $P = 3$ and $P = 5$. Regarding energy consumption, it is crucial to highlight the potential impact of the deployment strategy on resource efficiency. The risk of deploying unnecessary pods poses a challenge to resource optimization. When employing Kube-scheduler, which tends to deploy slices even in resource-constrained scenarios, this may contribute to increased energy consumption, averaging around 80 kJ. In contrast, the Setpod-scheduler, with its more efficient deployment approach, demonstrates a lower average energy consumption of approximately 55 kJ. This efficiency not only results in energy savings but also correlates with a shorter average deployment time per slice, taking only about 155 seconds, compared to the approximately 240 seconds required by Kube-scheduler. The observed time difference is not attributed to the genetic algorithm which gives a solution in less than 0.1 second for deploying a slice on the used platform; instead, it is associated with kube-scheduler deploying pods that are deemed unnecessary when the available resources cannot accommodate all pods within the slice. This phenomenon leads to time inefficiencies, resulting in an elevated average deployment duration. It is worth noting that the genetic algorithm's remarkable speed in generating solutions is attributed to its inherent parallelism and efficient exploration of the solution space. Genetic algorithms, by nature, excel in parallel

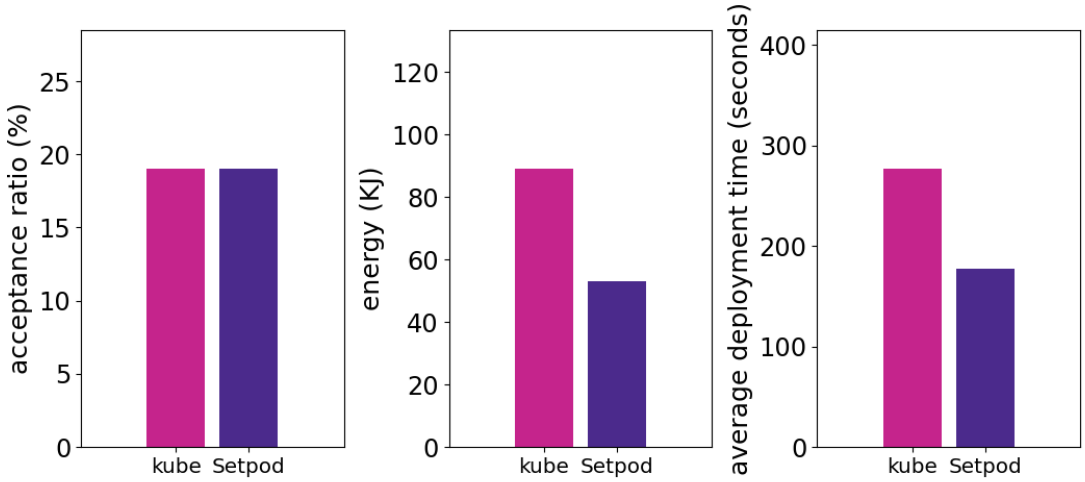

Figure 5: Kube-scheduler vs Setpod-scheduler for $P = 1$.

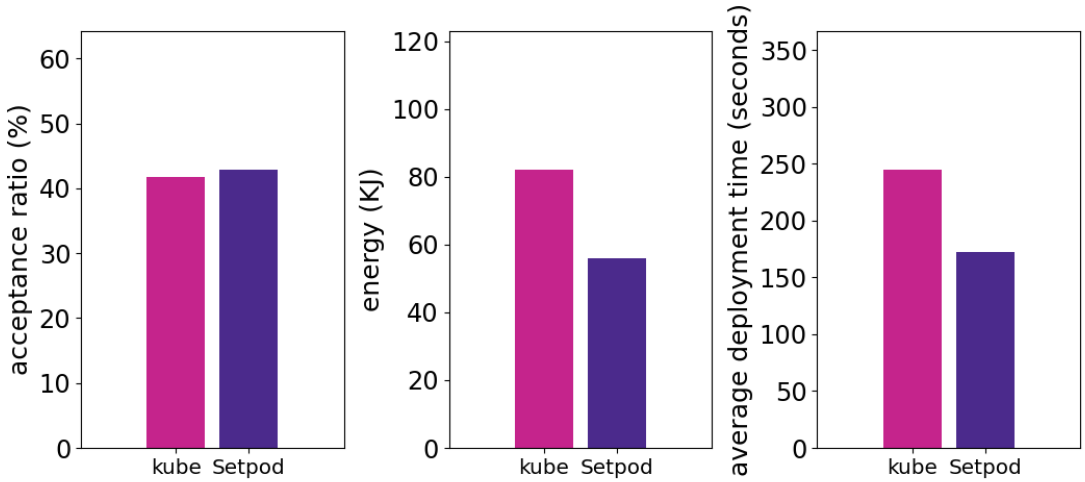

Figure 6: Kube-scheduler vs Setpod-scheduler for $P = 3$.

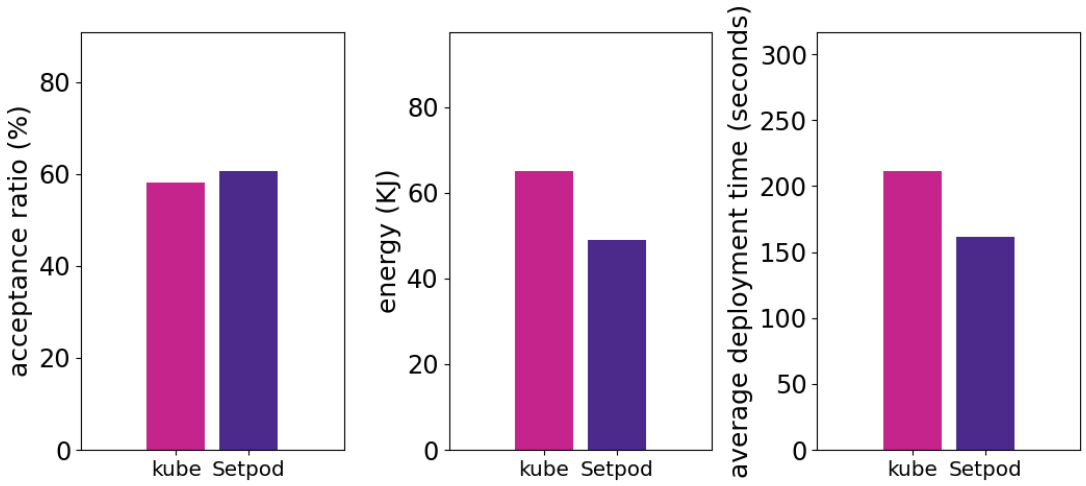

Figure 7: Kube-scheduler vs Setpod-scheduler for $P = 5$.

processing and can swiftly converge towards feasible solutions. However, as the complexity of the optimization problem increases or the solution space becomes more intricate, the algorithm's execution time may experience variations. In our specific scenario, the relatively simple deployment task contributes to the genetic algorithm's rapid performance. Nevertheless, the efficiency of genetic algorithms remains subject to the intricacy and scale of the optimization problem at hand.

6 Conclusion and Future Perspectives

In this work, K8s is used as a virtualization technology to abstract physical resources into virtual resources, and two schedulers are compared to deploy the virtualized resources of different slices on a common physical infrastructure. The goal is to maximize the acceptance ratio by deploying as many slices as possible while minimizing the energy consumption of the physical infrastructure. The experimental results demonstrate the importance of using a good slice scheduler for efficient network slicing orchestration. Unlike the default K8s scheduler that deploys the pods of a slice one by one, Setpod-scheduler deploys all the pods within a slice holistically (all pods or nothing). Setpod-scheduler has shown its efficiency in terms of energy consumption and average deployment time per slice compared to Kube-scheduler, with a good acceptance ratio. Setpod-scheduler is available online and can be tested on various platforms [29].

There are several extensions that we can see to this work. In the context of network slicing, it would be interesting to add more constraints, such as bandwidth and latency on the links between pods, and to observe the behavior of the scheduler in terms of QoS. Moreover, we used only a small cluster of 3 machines to compare the two schedulers. We plan to conduct larger-scale experiments in the future to determine the behavior of the GA by increasing the population size and the number of iterations. Also, we are interested in investigating other use cases that have a dependency within the pods of a slice. This helps in ensuring the generality of our scheduler to any use case of a multi-pod slice deployment (other than the 5G core). Finally, a dedicated scheduler for network slicing should manage the orchestration of slices without having the information related to the arrival and departure of slices in advance. Thus, an efficient scheduler should deploy a slice in a way that facilitates the scheduling of later arriving slices. It would therefore be interesting to integrate highly adaptable and intelligent behavior algorithms, capable of learning and making decisions through trial-and-error interactions with their environment, such as reinforcement learning-based methods.

References

[1] R. Botez, J. Costa-Requena, I.-A. Ivanciu, V. Strautiu, and V. Dobrota, "Sdn-based network slicing mechanism for a scalable 4g/5g core network: A kubernetes approach," *Sensors*, vol. 21, no. 11, p. 3773, 2021.

[2] S. Barrachina-Muñoz, M. Payaró, and J. Mangues-Bafalluy, "Cloud-native 5g experimental platform with over-the-air transmissions and end-to-end monitoring," in *2022 13th International Symposium on Communication Systems, Networks and Digital Sig-*

nal Processing (CSNDSP). IEEE, 2022, pp. 692–697.

[3] Y. Li, J. Zhang, H. Xue, J. Ma, J. Wu, M. Zhao, C. Han, and X. Dang, "5g core network slices embedding and deploying based on greedy algorithm in smart grids," in *2022 IEEE 14th International Conference on Advanced Infocomm Technology (ICAIT)*. IEEE, 2022, pp. 31–35.

[4] I. Afolabi, M. Bagaa, T. Taleb, and H. Flinck, "End-to-end network slicing enabled through network function virtualization," in *2017 IEEE Conference on Standards for Communications and Networking (CSCN)*. IEEE, 2017, pp. 30–35.

[5] M. Kassis, M. A. Aba, H. Castel-Taleb, M. Elkael, A. Araldo, and B. Jouaber, "Integrated deployment prototype for virtual network orchestration solution," in *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2022, pp. 1–3.

[6] O. Arouk and N. Nikaein, "5g cloud-native: Network management & automation," in *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2020, pp. 1–2.

[7] S. R. Chowdhury, R. Ahmed, N. Shahriar, A. Khan, R. Boutaba, J. Mitra, and L. Liu, "Revine: Reallocation of virtual network embedding to eliminate substrate bottlenecks," in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE, 2017, pp. 116–124.

[8] G. Sun, H. Yu, V. Anand, and L. Li, "A cost efficient framework and algorithm for embedding dynamic virtual network requests," *Future Generation Computer Systems*, vol. 29, no. 5, pp. 1265–1277, 2013.

[9] P. Zhang, H. Yao, M. Li, and Y. Liu, "Virtual network embedding based on modified genetic algorithm," *Peer-to-Peer Networking and Applications*, vol. 12, pp. 481–492, 2019.

[10] K. T. Nguyen and C. Huang, "Distributed parallel genetic algorithm for online virtual network embedding," *International Journal of Communication Systems*, vol. 34, no. 4, p. e4691, 2021.

[11] P. Zhang, Y. Hong, X. Pang, and C. Jiang, "Vnehpso: Virtual network embedding algorithm based on hybrid particle swarm optimization," *IEEE Access*, vol. 8, pp. 213 389–213 400, 2020.

[12] X. Guan, X. Wan, B.-Y. Choi, and S. Song, "Ant colony optimization based energy efficient virtual network embedding," in *2015 IEEE 4th International Conference on Cloud Networking (CloudNet)*. IEEE, 2015, pp. 273–278.

[13] M. Diallo, A. Quintero, and S. Pierre, "An efficient approach based on ant colony optimization and tabu search for a resource embedding across multiple cloud providers," *IEEE Transactions on Cloud Computing*, vol. 9, no. 3, pp. 896–909, 2019.

[14] A. Fischer, J. F. Botero, M. T. Beck, H. De Meer, and X. Hesselbach, "Virtual network embedding: A survey," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 4, pp. 1888–1906, 2013.

[15] H. Cao, S. Wu, Y. Hu, Y. Liu, and L. Yang, "A survey of embedding algorithm for virtual network embedding," *China Communications*, vol. 16, no. 12, pp. 1–33, 2019.

[16] A. Hashmi and C. Gupta, "A detailed survey on virtual network embedding," in *2019 International Conference on Communication and Electronics Systems (ICCES)*. IEEE, 2019, pp. 730–737.

[17] H.-K. Lim, I. Ullah, Y.-H. Han, and S.-Y. Kim, "Reinforcement learning-based virtual network embedding: A comprehensive survey," *ICT Express*, 2023.

[18] M. Elkael, M. A. Aba, A. Araldo, H. Castel-Taleb, and B. Jouaber, "Monkey business: Reinforcement learning meets neighborhood search for virtual network embedding," *Computer Networks*, vol. 216, p. 109204, 2022.

[19] M. Elkael, B. Jouaber, H. Castel-Taleb, A. Araldo, D. Olivier *et al.*, "A two-stage algorithm for the virtual network embedding problem," in *2021 IEEE 46th Conference on Local Computer Networks (LCN)*. IEEE, 2021, pp. 395–398.

[20] OAI, "Oai-5g-basic v1.5.0, https://gitlab.eurecom.fr/oai/cn5g/oai-cn5g-fed/-/tree/master," 2023.

[21] M. Elkael, A. Araldo, S. d'Oro, H. Castel-Taleb, M. A. Aba, and B. Jouaber, "Joint placement, routing and dimensioning at the network edge for energy minimization," in *IEEE Globecom*, 2023.

[22] W. Huang, A. Araldo, H. Castel-Taleb, and B. Jouaber, "Dimensioning resources of network slices for energy-performance trade-off," in *ISCC 2022: 27th IEEE Symposium on Computers and Communications*. IEEE, 2022.

[23] "kube-scheduler:v1.26.3, https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/," 2023.

[24] "Kubernetes documentation, https://kubernetes.io/docs/home/."

[25] M. Iori, V. L. De Lima, S. Martello, F. K. Miyazawa, and M. Monaci, "Exact solution techniques for two-dimensional cutting and packing," *European Journal of Operational Research*, vol. 289, no. 2, pp. 399–415, 2021.

[26] "5GCN, https://gitlab.eurecom.fr/oai/cn5g/oai-cn5g-amf/-/wikis/home."

[27] A. Güngör, "Ueransim v3.2.6, https://gitlab.eurecom.fr/oai/cn5g/oai-cn5g-fed/-/tree/master."

[28] "OAI 5GC deployment, https://github.com/aidy-f2n/oai-ueransim," 2023.

[29] "Setpod-scheduler, https://github.com/aidy-f2n/setpod-scheduler," 2023.