The Journal of Supercomputing (2023) 79:11478–11494
https://doi.org/10.1007/s11227-023-05122-5

# Network SLO-aware container scheduling in Kubernetes

**Eunsook Kim[1] · Kyungwoon Lee[2] · Chuck Yoo[3]**

Accepted: 15 February 2023 / Published online: 28 February 2023


## Abstract

In clouds, various services run on respective containers and have service-level objectives (SLO) that significantly impact service qualities. However, Kubernetes, a widely used container orchestration platform, does not schedule containers with respect to the network SLOs. This paper proposes a new container scheduling technique consisting of a cloud-level and node-level scheduler. The cloud-level scheduler selects a node that is best suited for satisfying the network SLO, and the node-level scheduler adjusts the CPU allocation for the container to satisfy SLOs on the selected node. We implement the cloud-level scheduler in Kubernetes and the node-level scheduler in the Linux kernel module and evaluate them using simulation and actual deployment. The evaluation results show that the cloud-level scheduler reduces the scheduling overhead by 22× compared to DRF, a representative multi-resource scheduling technique. Also, the node-level scheduler increases the number of containers that satisfy SLOs by 2.5× compared to native Kubernetes, which will significantly enhance the service quality of user-facing services.

**Keywords** Container scheduling · Service-level objectives · Service quality · Network performance

✉ Kyungwoon Lee
kwlee87@knu.ac.kr

✉ Chuck Yoo
chuckyoo@os.korea.ac.kr

Eunsook Kim
eden.do@kakaoenterprise.com

[1] Kakao Enterprise, Pangyoyeok-ro 235, Seongnam, Gyeonggi-do 13494, South Korea

[2] School of Electronics Engineering, Kyungpook National University, 80 Daehak-ro, Buk-gu, Daegu 41566, South Korea

[3] College of Informatics, Korea University, 145 Anam-ro, Seongbuk-gu, Seoul 02841, South Korea

## 1 Introduction

Kubernetes [1] is the de-facto container orchestration platform that dynamically creates and manages multiple containers simultaneously. Due to its simple interface and powerful functionalities, Kubernetes is utilized in many industrial fields [2, 3]. When a tenant requests to create a container with a description file that specifies the resource demand, such as the number of CPU cores and memory size for the container, Kubernetes performs container scheduling that examines nodes to find the most suitable one to run the container.

The container scheduling of Kubernetes examines the number of available CPU cores and the free memory size of nodes. Then, it selects a node with available CPU cores and memory space larger than the resource demand. However, currently, Kubernetes does not check the available network bandwidth of nodes in the node selection. When the network bandwidth demand (called SLO – service-level objectives [4, 5]) is specified, it can exceed the network capacity of the node so that the SLO is not satisfied. This can cause Kubernetes to prevent containers from satisfying network SLOs. As the network SLO determines the service quality of many cloud applications, such as user-facing services [6–9], it is crucial to take the network SLO into account in Kubernetes container scheduling.

Previous studies [10, 11] for satisfying network SLOs are based on multi-resource scheduling that allocates multiple resources such as CPU, memory, and network to containers simultaneously. The amount of resource allocation is determined by the resource demands specified by tenants. Dominant resource fairness (DRF) [11] is a representative multi-resource scheduling that enables multi-resource scheduling for high performance and efficient resource utilization [12, 13]. However, DRF-based multi-resource scheduling has a drawback: The scheduling overhead becomes very significant with the increase in the number of tenants because the DRF algorithm calculates the dominant share of every tenant [14].

This paper proposes cloud-level and node-level schedulers designed to satisfy the network SLO and incur low scheduling overhead. First, our cloud-level scheduler investigates the available network bandwidth of nodes. Then, it selects a set of nodes with available network bandwidth larger than the SLO. Then, our scheduler checks the number of available CPU cores of the selected nodes and chooses the one with the largest number of CPU cores available. This is because it has been reported that a "proper" amount of CPU has to be provisioned in order for a node to achieve network SLOs [15]. For example, the experiment results in [15] show that the CPU usage of a container increases by two times when the network SLO increases from 50 Mbps to 100 Mbps. So our scheduler considers both the network bandwidth and the number of available CPU cores. This is distinct from the current Kubernetes scheduler, which only considers the number of available CPU cores and free memory space.

Second, our node-level scheduler works on the chosen node. Because it is not known how many CPU cores are required for the network SLO, our node-level scheduler monitors the actual network bandwidth. Then, the scheduler dynamically adjusts the CPU allocation in the direction to minimize the gap between the

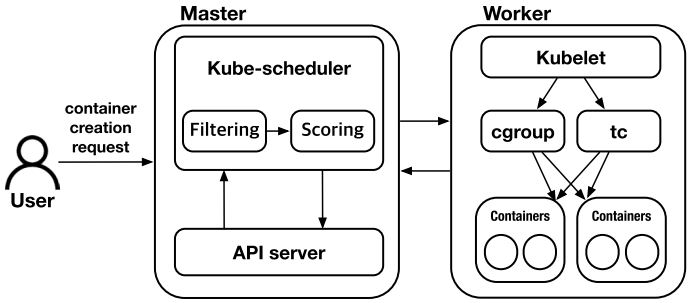



**Fig. 1** Container scheduling in Kubernetes

SLO and the actual network bandwidth. We adopt the idea of [15] and modify it to work with our node-level scheduler. Our scheduler works in hybrid architecture that divides container scheduling into cloud-level and node-level, which differs from DRF.

There are many previous studies [5, 6, 16] on scheduling network bandwidth in container virtualization. The studies mostly aim to allocate network bandwidth to respective containers. However, our technique suggests allocating network bandwidth at the cloud-level scheduling while enforcing the CPU allocation at the node-level scheduling. Thus, multiple resource scheduling seems the most appropriate study for comparison because multiple computing resources, such as CPU and network bandwidth, are allocated together.

We implement the cloud-level scheduler in Kubernetes and the node-level scheduler as a Linux kernel module[1]. Our performance evaluation results include large-scale simulations for the cloud-level scheduler and experiments using actual deployment for the node-level scheduler. The results show that our cloud-level scheduler reduces the scheduling overhead by 22× compared to DRF. Moreover, our node-level scheduler improves the number of containers that satisfy SLOs by 2.5× compared to native Kubernetes while reducing the number of containers that fail to satisfy SLOs by 7×.

## 2 Background and motivation

This section first explains the container scheduling of Kubernetes. Then, we describe dominant resource fairness (DRF) and its limitation, which motivates this paper.

### 2.1 Kubernetes container scheduling

Figure 1 illustrates the container scheduling in Kubernetes. First, when a tenant requests to create a container by specifying computing resources, the master node of Kubernetes receives the request. The request is forwarded to the API Server, and the

[1] The source code of the prototype implementation can be found at https://github.com/kiiimes/DepCon.

Kubernetes scheduler (i.e., *kube-scheduler*) recognizes the request. *Kube-scheduler* handles the filtering process to create a list of nodes that can satisfy the specification of the container. This means *kube-scheduler* searches worker nodes that can offer CPU cores and memory size to satisfy the specification. Note that *kube-scheduler* does not check the available network bandwidth of worker nodes.

Based on the list of worker nodes, the scheduler performs the scoring process. In the scoring process, the scheduler selects a worker node with the highest priority by calculating the weight of the worker nodes in the list according to the scheduling policy configured by system administrators. By default, the scheduler computes weights based on the available CPU cores and memory size of the worker nodes. After selecting a worker node, the scheduler notifies the *kubelet* agent running on the selected worker node to create the container. When *kubelet* receives a request to create a container, *kubelet* allocates computing resources for the container by using the `cgroups` API (e.g., `cpuset`, `cpu.cfs_quota_us`).

If a network bandwidth is specified, the network bandwidth is managed only by *kubelet* using tc rather than *kube-scheduler* `tc`. This indicates that the network bandwidth specified in the description file is not considered in the filtering and scoring process of *kube-scheduler* and is only applied in the worker node using `tc`. As a result, when multiple containers running on a worker node have their demand for the network bandwidth, the sum of the network bandwidth can exceed the network capacity of the worker node, which inevitably hampers the service quality.

### 2.2 Dominant resource fairness scheduling

Dominant Resource Fairness (DRF) [11] is a fair sharing model that generalizes max-min fairness to multiple resource types. DRF receives the resource demand that includes the amount of computing resources such as CPU and memory required for a job from tenants, similar to the container scheduling in Kubernetes. Then, from the resource demand, DRF finds a dominant resource with a larger fraction among multiple resources. For example, when a tenant requests to create a container with 1 CPU and 100 MB in a cluster that consists of servers equipped with 10 CPU and 10 GB memory, the dominant resource for the tenant is the CPU. This is because the CPU for the container has a larger share (i.e., 0.1) than memory (i.e., 0.01) when the container runs on the server with the requested resources.

In addition, DRF operates as follows in multi-tenant clouds: Assume that there are two tenants (t1 and t2) and the tenants have different resource demands (t1 = (100 Mbps, 200%) and t2 = (300 Mbps, 100%). When the total capacity of the two servers is (2000 Mbps, 2000%), resource share is calculated as (requested/total capacity), so t1's resource share is (1/20, 1/10) and t2's resource share is (3/20, 1/20). In other words, t1's dominant share is 1/10 (CPU resource share), and t2's

**Table 1** Notations for Algorithm 1

| Notation | Meaning | Notation | Meaning |
|---|---|---|---|
| $N$ | The total number of worker nodes | $C_{SLO}$ | Network SLO of a container |
| $S$ | A set of the entire worker nodes | $s_i$ | $i$th worker node |
| $AS$ | A set of available worker nodes | $as_i$ | $i$th available worker node |
| $s_i$.netIdle | Idle network bandwidth of $s_i$ | $s_i$.cpuIdle | The number of idle CPU cores of $s_i$ |

dominant share is 3/20 (network resource share). After calculating dominant shares for every tenant, DRF compares the dominant shares of all tenants and allocates resource demand to the tenant's job with the smallest dominant share.

Even though DRF and its variants improve resource utilization with the fairness guarantee between tenants, it brings significant scheduling overhead with large numbers of tenants. The reason is that the DRF algorithm iterates all tenants to calculate the dominant share of every resource for each tenant [14]. As a result, the scheduling delay caused by the DRF algorithm can increase as the number of tenants increases.

## 3 Design and Implementation

This paper proposes the cloud-level scheduler and the node-level scheduler as follows: (1) the cloud-level scheduler finds the suitable worker node to create containers that can offer the network bandwidth specified as the SLOs in the description file as in Algorithm 1. (2) To achieve network SLOs, the node-level scheduler allocates the "proper" amount of CPU resources to containers depending on the network SLOs.

### 3.1 Cloud-level scheduling algorithm

Cloud-level container scheduling plays a key role as it determines the application performance, resource utilization, and even power consumption of the nodes in clouds [17–19]. In this paper, we focus on container scheduling for achieving the network SLOs of containers with low overhead. To minimize the overhead of cloud-level scheduling, we construct a scheduling algorithm depicted in Algorithm 1 that consists of two phases: *bandwidth filtering* and *CPU scoring*. Note that Table 1 describes each notation and the meaning in Algorithm 1.

**Algorithm 1** Cloud-level scheduling

```
1: count, FW ← 0
2: AS, FS ← ∅
3: for i ∈ 1..N do
4:     idle ← s_i.netIdle
5:     if idle > C_SLO then
6:         as_{++count} ← s_i
7:     end if
8: end for
9: for j ∈ 1..(count − 1) do
10:    if FS = ∅ || FW < as_j.cpuIdle then
11:        FS ← as_j
12:        FW ← as_j.cpuIdle
13:    end if
14: end for
15: Return FS
```

Given that there is a cloud data center with *N* worker nodes represented by $S=\{s_1...s_N\}$ with the idle network bandwidth and the number of idle CPU cores indicated by *s.n*etIdle and *s.c*puIdle, respectively. The value of *s.n*etIdle and *s.c*puIdle ranges from zero to the maximum capacity of the node. *Bandwidth filtering* first investigates the value of *s.n*etIdle for every worker node in the cloud because the goal of *bandwidth filtering* is to find worker nodes with a larger network bandwidth than containers demand. For example, when a container requests 200 Mbps of network bandwidth, it can only run on worker nodes with available network bandwidth larger than 200 Mbps.

From lines 1 to 9 in Algorithm 1, we describe *bandwidth filtering*, which creates a list of nodes (*A*S) with available network bandwidth larger than the requested network SLO ($C_{SLO}$). In other words, the worker node, $s_i$, with $s_i$.*netIdle* larger than $C_{SLO}$ is included in *A*S. Thus, the maximum value of *c*ount would be *N*. When there is no worker node with available network bandwidth larger than the SLO, *c*ount remains as zero, and the scheduler rejects the request for creating containers.

In *CPU scoring*, the scheduler assigns a weight to each node in the list based on the number of idle CPU cores (lines 10-15). It examines the number of idle CPU cores ($as_j$.*cpuIdle*) of the worker nodes in *A*S sequentially. The value of $as_j$.*cpuIdle* is stored in *F*W where the largest number of available CPU cores among the worker nodes in *A*S is stored. *F*W is compared to the number of idle CPU cores of the next worker node ($as_{j+1}$.*cpuIdle*). Only the larger value of $as_j$.*cpuIdle* remains in *F*W, and the corresponding worker node is selected as the worker node (*F*S) that creates the container. This indicates that the worker node with the largest number of available CPU cores receives the highest priority. High priority is assigned to the worker nodes with a large number of available CPU cores because we do not know the "proper" number of CPU cores for achieving network SLOs. If the node with the largest number of available CPU cores does not satisfy the SLO, no worker node can achieve the SLO because of insufficient CPU cores.

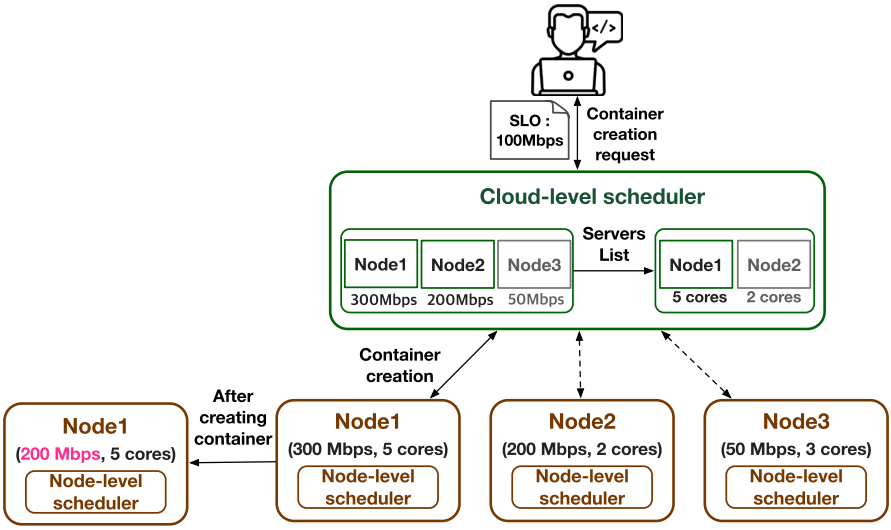



**Fig. 2** Overall design

### 3.2 Cloud-level scheduler

Figure 2 depicts how the cloud-level scheduler works. When the cloud-level scheduler receives a request to create a container with 100 Mbps network SLO from a tenant, it creates a list of nodes with available network bandwidth of more than 100 Mbps among three nodes, which are Node 1 (N1) and Node 2 (N2) in Fig. 2. Between N1 and N2, the cloud-level scheduler selects a node with the larger number of available CPU cores. So, N1 is selected as the node to create the container with a network SLO of 100 Mbps. When the container is created on N1, the available bandwidth of N1 decreases from 300 Mbps to 200 Mbps. The node-level scheduler on N1 receives the information of the created container and the network SLO from the cloud-level scheduler.

In terms of overhead, our cloud-level scheduler offers a low scheduling overhead compared to the DRF-based scheduling algorithm. The DRF-based scheduling algorithm iterates resource allocation for the number of tenants ($T$) and resource types ($R$). Hence, the time complexity of the DRF-based algorithm is $O(R^2T)$ [14] in which the scheduling overhead increases with the increase of the number of tenants and resource types. On the other hand, our method does not iterate resource allocation for the number of tenants or resource types, only for the number of nodes ($N$). Thus, the worst-case time complexity of our cloud-level scheduler is $O(N)$. For example, when there is only one worker node that has available network bandwidth larger than the network SLO of containers, the number of total operations for bandwidth filtering and CPU scoring is $N$ and 1, respectively. When the entire worker nodes have sufficient network bandwidth for containers, the number of total operations for CPU scoring increases to $N$ while that for bandwidth filtering is $N$ independent of the number of CPU scoring.

Thus, the maximum number of operations for bandwidth filtering and CPU scoring is $2N$, which leads to $O(N)$.

We implement the cloud-level scheduler of Algorithm 1 in *kube-scheduler* of Kubernetes. Note that the modified *kube-scheduler* conducts Algorithm 1 in addition to the existing native *kube-scheduler* algorithm. For example, when a container has the resource demands for memory and network bandwidth, the modified *kube-scheduler* finds a suitable worker node with sufficient free memory size and available network bandwidth. This can support containers that have resource demands for multiple computing resources such as CPU, memory, and network bandwidth. In addition, as our cloud-level scheduler selects a node with the largest available CPU cores, it assigns a higher weight to the CPU cores (i.e., 10) than the memory size (i.e., 1). For example, when there are two nodes (N1 and N2) with available CPU cores (N1=50%, N2=70%) and memory sizes (N1=70%, N2=50%), our scheduler selects N2 with the larger available CPU cores and less available memory than N1.

### 3.3 Node-level scheduler

Our node-level scheduler monitors the actual network bandwidth in a period and calculates the proper CPU as follows:

$$\mathrm{CPU}_{alloc} = \mathrm{CPU}_{prev} + \mathrm{CPU}_{prev} * k * \{(N_S - N_P)/N_S\} \quad (1)$$

where $\mathrm{CPU}_{prev}$ and $\mathrm{CPU}_{alloc}$ indicate the CPU allocation in the previous period and current period. $\mathrm{CPU}_{alloc}$ is calculated to be proportional to the difference between the SLO ($N_S$) and the actual bandwidth ($N_P$). In Eq. 1, $k$ is a tunable parameter that determines the convergence speed. For example, the large value of $k$ increases the speed for achieving the SLOs by increasing the CPU allocation in big steps. However, this can result in a large fluctuation in the network performance. When $k$ is small, the performance of containers does not fluctuate much. But, as the CPU allocation is changed at a slow pace, which leads to slow convergence to the network SLO.

We implement our node-level scheduler as a Linux kernel module (LoC is 317). Also, we modify the source code of the *kubelet* agent in order for *kubelet* to deliver the information of containers to the kernel module when the container is initialized. The container information includes the specifications in the description file, process IDs, and the virtual network interface the container utilizes in the worker node. Based on the container information, our node-level scheduler periodically monitors the number of packets processed in the virtual network interface of containers. When the actual performance becomes lower than the network SLO, our scheduler adjusts the CPU allocation using Eq. 1 by utilizing the Linux `cgroups`. When the actual performance exceeds the network SLO, our node-level scheduler adjusts CPU allocation to reduce the allocation for the containers. In our implementation, the period is set to one second, which is found to be the best empirically [15]. Note that our cloud-level and node-level schedulers are designed to be separate components in the hybrid architecture.

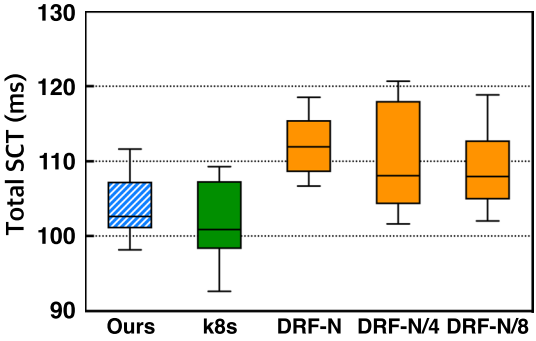



**Fig. 3** CloudSim result

## 4 Evaluation

We conduct two sets of experiments. First, we evaluate the scheduling overhead of the proposed technique using large-scale simulation. Also, we compare the overhead with that of native Kubernetes and DRF to show that our cloud-level scheduler only incurs negligible overhead, which is much lower than DRF. Second, we measure the actual performance of containers on a rack-scale cluster environment when multiple containers run concurrently.

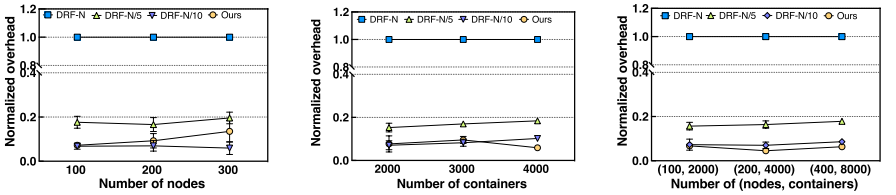
### 4.1 Scheduling Overhead Analysis

Because container scheduling deals with numerous nodes and containers, it is essential to offer low scheduling overhead for high scalability [20]. We evaluate the scheduling overhead by measuring the scheduling completion time (SCT). SCT indicates the time spent selecting the appropriate node for containers.

#### 4.1.1 CloudSim evaluation

For the experiment, we utilize a representative cloud simulator, CloudSim [21], and measure the total SCT for creating the entire containers. Figure 3 shows the results of the CloudSim where there are 400 nodes and 300 containers. For the native Kubernetes (i.e., k8s) experiments, we assign the resource demand of containers for CPU and memory in a random manner, while we (Ours in Fig. 3) use the proposed technique for network SLOs for the experiments. Also, for DRF experiments, we vary the number of tenants (i.e., x-axis) denoted as DRF-*N*, DRF-*N*/4, and DRF-*N*/8, which indicate the number of tenants in each experiment, where *N* means the number of containers. In other words, DRF-*N*/4 indicates that the number of tenants is $\frac{300}{4}$.

Figure 3 illustrates that the proposed technique only increases scheduling overhead by 2% compared to native Kubernetes on average. This is because the bandwidth filtering and CPU scoring of the proposed technique do not require complex arithmetic calculations. Moreover, our implementation based on native Kubernetes




(a) Increase nodes (b) Increase containers (c) Different configurations

**Fig. 4** Scheduling overhead of our technique and DRF with different configurations

utilizes the existing operations of native Kubernetes and embeds several functionalities for our scheduler. For example, in native Kubernetes, *kube-scheduler* retrieves the worker node information, such as the number of available CPU cores and free memory size. We modify *kube-scheduler* to include the available network bandwidth in addition to the existing worker node information. As our cloud level minimizes the additional operations in *kube-scheduler*, the overhead caused by the proposed technique becomes negligible. On the other hand, the SCT values with DRF-*N*, DRF-*N*/4, and DRF-*N*/8 increase by 10%, 8%, and 7%, respectively, compared to that with native Kubernetes. The reason is that the dominant fair share of each tenant needs to be calculated and updated to guarantee fairness between the tenants. As a result, with the largest number of tenants, DRF requires 112 ms on average to create 300 containers, while the proposed technique only consumes 104 ms on average.

### 4.1.2 Support for different configurations

Next, we conduct a simulation with various configurations, such as the number of nodes and containers. Under each configuration, we measure the total SCT for creating the entire containers. For the experiment, we had to build our simulator due to the scalability issue of CloudSim, which cannot run more than 700 containers. Also, we present the results as the normalized SCT, where one indicates the results of DRF-*N*. This is because the values of SCT in the simulation vary depending on the configurations and the experiment environments. Figure 4a shows the normalized SCT with the increasing number of nodes when the number of containers is fixed as 2,000. As in the previous experiment, the number of tenants increases from *N*/10 to *N*. The scheduling overhead of our cloud-level scheduler increases by 1.8× (from 0.072 to 0.135) as the number of nodes increases from 100 to 300. This is because the time complexity of Algorithm 1 is only dependent on the number of nodes.

With 300 nodes, our cloud-level scheduler reduces the overhead by 640% and 45% compared to DRF-*N* and DRF-*N*/5, respectively. However, our scheduler offers scheduling overhead higher than that of DRF-*N*/10. This is because the time complexity of our cloud-level scheduler and DRF is O($N$) and O($T$), respectively, where $N$ denotes the number of nodes while $T$ indicates the number of tenants. As there are 2000 containers, the number of tenants with DRF-*N*/10 is 200, less than the number

of nodes. As a result, the overhead of our cloud-level scheduler can become higher than that of DRF when the number of nodes is larger than the number of tenants.

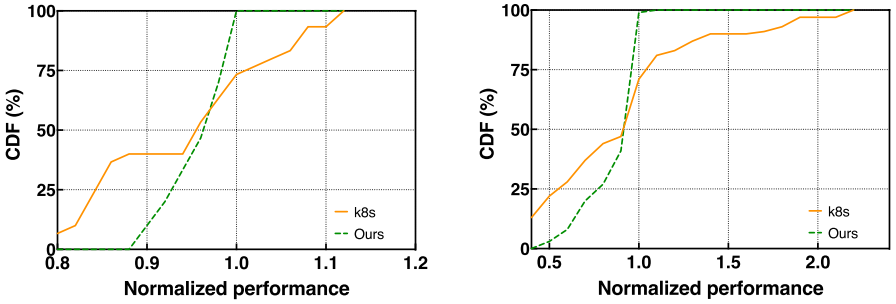
Figure 4b depicts the scheduling overhead comparison between DRF and our scheduler when the number of nodes is fixed to 200 with increasing containers from 2000 to 4000. We find that our technique outperforms all DRF cases. This is because the scheduling overhead of DRF increases as the number of containers and tenants also increases. This is different from our technique, which is not affected by the number of tenants. Moreover, when the number of containers increases, the scheduling overhead of our technique does not increase but decreases. This is because the absolute SCT of DRF-*N* increases with the increase in containers. For example, the absolute SCT of DRF-*N* increases by 3.6× (from 0.54 to 1.94) when the number of containers increases from 2000 to 4000. On the other hand, the absolute SCT of our technique only increases by 1.9×, which decreases the normalized SCT of our technique.

At last, Fig. 4c illustrates the change of scheduling overhead when the number of nodes and containers increases. We increase the number of nodes from 100 to 400 while increasing the number of containers from 2000 to 4000. The result shows that our cloud-level scheduler offers the lowest scheduling overhead independent of the number of nodes and containers. For example, with 200 nodes and 4000 containers, the proposed technique reduces the scheduling overhead by 22× compared to DRF-*N*. Even though DRF-*N*/5 and DRF-*N*/10 offer low scheduling overhead compared to DRF-*N*, they show some increase in scheduling overhead as the number of nodes and containers increases. However, our scheduler does not show any increase in scheduling overhead but decreases when the number of nodes and containers increases. Although this does not mean a decrease in the absolute SCT, it indicates that the SCT of our scheduler does not increase linearly with the increase in nodes and containers.

### 4.2 Container performance analysis

Now, we run two sets of experiments to evaluate the performance of containers running with the proposed technique. For experiments, we utilize the actual deployment that consists of 10 physical servers connected via a 10 GbE network switch. Each server runs either native Kubernetes or modified Kubernetes with our implementation on Ubuntu 18.04 and Linux kernel version 5.3. Note that the version of Kubernetes is 1.18.3, while the container runtime is Docker version 19.03. The servers are equipped with an Intel Xeon CPU E5-2650v3@2.3 GHz (10 cores), 128 GB memory, and 256 GB SSD.

First, we utilize four servers among 10 servers and configure the experimental environment with one master node, two worker nodes, and an evaluation machine. We create 30 containers on the two workers, and the containers have two different SLOs in a random manner (100 Mbps or 300 Mbps). Second, we utilize 10 servers in total and assign one master node, six worker nodes, and three evaluation machines. Then, we increase the number of containers for experiments to 100 while




(a) Two worker nodes (b) Six worker nodes

**Fig. 5** Normalized performance of containers with different SLOs

specifying four different SLOs (e.g., 100 Mbps, 200 Mbps, 300 Mbps, and 400 Mbps) randomly to each container. Note that we specify the network SLOs in the description file of containers without any resource demand for CPU or memory for both the proposed technique and native Kubernetes.

For experiments, we utilize the Netperf [22] benchmark to measure the network performance of containers and configure the containers running on the worker node to transmit 64 B TCP packets to the evaluation machine. Note that we present the cumulative distribution function (CDF) (y-axis) of normalized performance (y-axis) in which one indicates the value of SLOs. At last, we compare the results of our technique with that of native Kubernetes (i.e., k8s).

Figure 5a shows that most of the containers satisfy SLOs with the proposed technique. Among 30 containers, only three (i.e., 10%) achieve network performance less than 90% of the SLO, while the rest satisfy the SLO. On the other hand, with native Kubernetes, 40% of containers achieve network performance less than 90% of the SLO. This means that our scheduler shows 4× better performance than Kubernetes. Note that we set the degree of SLO achievement to 90% which is generally utilized as the low bound in most of the previous studies [23–25].

The major reason for the performance degradation in native Kubernetes is the lack of considerations for network SLOs in container scheduling. Native Kubernetes does not consider the network SLOs in container scheduling, but the containers have different network SLOs, such as 100 Mbps and 300 Mbps, which require different amounts of CPU cores. This can bring CPU contention in the worker node that runs containers with network SLOs of 300 Mbps. Actually, we find that the containers with network SLOs of 300 Mbps suffer from severe performance degradation compared to others with network SLOs of 100 Mbps. The average network bandwidth of the containers with network SLOs of 300 Mbps is 270 Mbps while that of the containers with network SLOs of 100 Mbps is 105 Mbps on native Kubernetes.

On the other hand, our node-level scheduler dynamically allocates CPU cores to containers depending on the network SLOs using Eq. 1. This enables containers to receive proper CPU cores to achieve the network SLO. Thus, the containers with a network SLO of 300 Mbps receive a larger number of CPU cores than those with a

network SLO of 100 Mbps. In addition, the cloud-level scheduler assigns a worker node with the largest amount of available CPU cores among the entire worker nodes. If a worker node is busy with multiple containers that require a large number of CPU cores, it is not selected by the cloud-level scheduler. As a result, the integrated scheduling of the cloud-level and node-level scheduler enables containers to achieve the network SLOs effectively.

Figure 5b depicts the normalized performance of 100 containers under native Kubernetes and our scheduler, respectively. We find that the increasing number of containers and worker nodes aggravates the performance degradation of containers. With native Kubernetes, 22 containers achieve low network performance decreasing to half of the SLO, while 44 containers achieve network performance of less than 80% of SLOs. Only 24 containers (i.e., 24%) satisfy SLOs by achieving normalized performance ranging from 0.9 to 1.0. Similar to Fig. 5a, the reason for the performance degradation is CPU contention in specific worker nodes. When native Kubernetes distributes containers to worker nodes, the CPU usage of containers varies depending on the network SLOs of the containers. We find that a worker node runs the largest number of containers with network SLOs of 400 Mbps, which allows the containers on the worker node not to receive sufficient CPU allocation.

Our scheduler mitigates the CPU contention and reduces the number of containers that achieves 50% of SLOs to three, which is a 7× reduction. The number of containers that achieve normalized performance ranging from 0.9 to 1.0 is 60 with our scheduler, which is 2.5× higher than native Kubernetes. Overall, our scheduler increases the ratio of containers that satisfy SLOs (normalized performance ranging from 0.9 to 1.0) by 1.6× compared to native Kubernetes. Also, it reduces the number of containers that fail to satisfy SLOs (normalized performance less than 0.9) by 41%. This indicates that our node-level scheduler enables containers to satisfy different SLOs simultaneously by offering a proper CPU allocation depending on the SLOs.

## 5 Related Work

This section describes relevant studies to this paper, including multi-resource scheduling that has been actively researched for cloud environments. As jobs (e.g., containers) in clouds share computing resources (e.g., CPU, memory, and network bandwidth) simultaneously, they suffer from performance interference and SLO violation. The primary goal of multi-resource scheduling is to resolve such issues and improve resource utilization and system efficiency while offering fairness between tenants.

DRF [11] and H-DRF [13] are representative multi-resource schedulers. They aim to provide fairness in resource allocation by applying the generalization of max-min

fairness that maximizes the minimum allocation received by a tenant in the system for multiple resource types. DRF considers the heterogeneous data center applications and allocates the same dominant share, the maximum among all tenant shares, to all jobs. H-DRF applies a hierarchical structure to DRF to offer multi-resource scheduling for Hadoop frameworks. However, DRF and H-DRF have high-computational complexity because they calculate resource allocation for every tenant and resource [14].

Other studies [14, 28–30] achieve fairness in resource allocation while solving several issues, such as resource utilization or overhead of DRF. DC-DRF [14] is the adaptive approximation of DRF to reduce the time complexity for multi-resource allocation at a centralized controller. It presents several optimization techniques, such as parallelism and NUMA-awareness, to improve the scheduling performance of the controller. PS-DSF [28] is a server-based DRF extension for the fair resource allocation of multiple resources in heterogeneous servers with placement constraints. PS-DSF proposes the max-min fairness of virtual dominant shares for tenants associated with each server to improve resource utilization. Carbyne [29] is an altruistic approach focusing on long-term fairness rather than immediate fairness. It improves average job completion time and cluster resource utilization by re-locating the leftover resources without violating fairness. TSF [30] is a new sharing policy that considers multi-resource shares for data center jobs with placement constraints. TSF suggests removing the placement constraint and allocating the maximum amount of resources. Even though it increases resource utilization by providing idle resources to tenants, it increases scheduling overhead when there are more than 100 tasks configured to run the job, increasing the job's total runtime.

Unlike most of the studies based on DRF, HUG [12] aims to increase resource utilization and guarantee minimal performance, which is similar to this paper. HUG limits the bandwidth utilization of each tenant to ensure optimal isolation and high network utilization for multiple tenants. Also, HUG can satisfy the network SLOs, as it reserves and allocates the minimum network resources to each tenant. Even though the technique can offer minimum network bandwidth through resource reservation, it cannot guarantee sufficient CPU cores for achieving specific network SLOs.

Recently, several Kubernetes-based container scheduling has been introduced [2, 3, 18, 26, 31]. They aim to optimize the current version of Kubernetes container scheduling by resolving performance interference and power consumption issues. However, most previous studies focus on allocating CPU and memory rather than network bandwidth. Moreover, the studies for network bandwidth allocation do not consider network SLOs. For example, a recent study [31] adopts quality of experience (QoE) as an SLO metric. The value of QoE is calculated by the mean opinion score built for video streaming services.

Table 2 demonstrates the previous studies relevant to this paper. In addition, as the proposed technique adopts a hybrid architecture that combines the cloud-level scheduler and node-level scheduler, it can support a single worker node and a cluster that consists of numerous worker nodes. This is different from previous studies [7, 15, 16, 32, 33] that focus on CPU or network bandwidth allocation on a single worker node.

**Table 2** Comparison table

| Paper | Goal | Architecture | SLO guarantee | Overhead |
|---|---|---|---|---|
| Our work | SLO guarantee | Hybrid | O | Low |
| DRF [11] | Fairness | Centralized | X | High |
| HDRF [13] | Fairness | Centralized | X | High |
| AlloX [26] | Fairness, utilization | Centralized | X | High |
| Kube-Sphere [2] | Fairness | Centralized | X | High |
| DC-DRF [27] | Fairness, utilization | Centralized | X | Low |
| PS-DSF [28] | Fairness, utilization | Hybrid | X | Low |
| Carbyne [29] | Fairness, utilization | Centralized | X | N/A |
| TSF [30] | Fairness | Distributed | X | High |
| HUG [12] | Performance isolation, utilization | Centralized | △ | High |

## 6 Discussion

This paper focuses on achieving network SLOs by selecting worker nodes with sufficient network bandwidth and available CPU cores. In order to provide sufficient CPU cores, the cloud-level scheduler assigns the highest priority to the worker node with the largest number of available CPU cores. This is because the required CPU usage of a container for achieving a specific network SLO can be found only after running the container on a worker node so we employ the dynamic CPU allocation of our node-level scheduler, which is a distinct feature of this paper.

## 7 Conclusion

This paper proposes a new container scheduling technique for achieving network SLOs in the Kubernetes environment. The proposed technique consists of the cloud-level and node-level scheduler that considers the available network bandwidth and the available CPU. The cloud-level scheduler performs *bandwidth filtering* and *CPU scoring* to find a worker node that can offer sufficient network bandwidth and CPU cores. The node-level scheduler dynamically adjusts CPU allocation for containers to achieve specified network SLOs. We design and implement the proposed technique in Kubernetes and evaluate the scheduling overhead while measuring the actual performance of containers. Our evaluation results show that our technique reduces the scheduling overhead by 22× compared to DRF. This is because the scheduling overhead of the proposed technique is independent of the number of tenants different from existing DRF-based techniques. Also, when we measure the actual network performance of containers, the number of containers that suffer from performance degradation decreases by 7× compared to native Kubernetes. This shows that the proposed technique is effective in achieving network SLOs of containers running concurrently.

**Author Contributions** EK and KL wrote the main manuscript text and EK prepared the figures. CY reviewed the manuscript.

**Funding** This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (Ministry of Science and ICT, MSIT) (No. RS-2022-00166222 and No. 2023R1A2C3004145) and Basic Science Research Program funded by the Ministry of Education (NRF-2021R1A6A1A13044830).

**Data Availability** Data and materials are available on request from the authors.

## Declarations

**Conflict of interest** The authors declare that there are no competing interests as defined by Springer, or other interests that might be perceived to influence the results and/or discussion reported in this paper.

**Ethics approval** This declaration is not applicable.

## References

1. LinuxFoundation.: Production-Grade Container Orchestration. http://Kubernetes.io/
2. Beltre A, Saha P, Govindaraju M, Kubesphere (2019) An approach to multi-tenant fair scheduling for kubernetes clusters. In, (2019) IEEE cloud summit. IEEE :14–20
3. Carrión C. Kubernetes scheduling: Taxonomy, ongoing issues and challenges. ACM Computing Surveys (CSUR). 2022;
4. Kannan RS, Subramanian L, Raju A, Ahn J, Mars J, Tang L (2019) Grandslam Guaranteeing slas for jobs in microservices execution frameworks. In: Proceedings of the fourteenth EuroSys conference. 1–16
5. Qiu H, Banerjee SS, Jha S, Kalbarczyk ZT, Iyer RK (2020) FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In: 14th USENIX symposium on operating systems design and implementation (OSDI 20). 805–825
6. Xu C, Rajamani K, Felter W. Nbwguard (2018) Realizing network qos for kubernetes. In: Proceedings of the 19th international middleware conference industry. 32–38
7. Khalid J, Rozner E, Felter W, Xu C, Rajamani K, Ferreira A, et al (2018) Iron: Isolating Network-based CPU in Container Environments. In: 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18); 313–328
8. Kim D, Yu T, Liu HH, Zhu Y, Padhye J, Raindel S, et al (2019) FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds. In: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19); 113–126
9. Guo Y, Yao W (2018) A container scheduling strategy based on neighborhood division in micro service. In: NOMS 2018-2018 IEEE/IFIP network operations and management symposium. IEEE; 1–6
10. Tembey P, Gavrilovska A, Schwan K. Merlin (2014) Application-and platform-aware resource allocation in consolidated server systems. In: Proceedings of the ACM symposium on cloud computing; 1–14
11. Ghodsi A, Zaharia M, Hindman B, Konwinski A, Shenker S, Stoica I (2011) Dominant resource fairness: Fair allocation of multiple resource types. In: 8th USENIX symposium on networked systems design and implementation (NSDI 11);
12. Chowdhury M, Liu Z, Ghodsi A, Stoica I HUG (2016) Multi-resource fairness for correlated and elastic demands. In: 13th USENIX symposium on networked systems design and implementation (NSDI 16); 407–424
13. Bhattacharya AA, Culler D, Friedman E, Ghodsi A, Shenker S, Stoica I(2013) Hierarchical scheduling for diverse datacenter workloads. In: Proceedings of the 4th annual symposium on cloud Computing; 1–15
14. Kash IA, O'Shea G, Volos S (2018) DC-DRF: Adaptive multi-resource sharing at public cloud scale. In: Proceedings of the ACM symposium on cloud computing; 374–385

15. Lee K, Lee K, Park H, Hwang J, Yoo C (2022) Autothrottle: satisfying network performance requirements for containers. IEEE Transact Cloud Comput
16. Hong CH, Lee K, Kang M, Yoo C (2018) qCon: QoS-aware network resource management for fog computing. Sensors. 18(10):3444
17. Chung A, Park JW, Ganger GR (2018) Stratus: Cost-aware container scheduling in the public cloud. In: Proceedings of the ACM symposium on cloud computing; 121–134
18. Menouer T (2021) KCSS: Kubernetes container scheduling strategy. J Supercomput 77(5):4267–4293
19. Zhao D, Mohamed M, Ludwig H (2018) Locality-aware scheduling for containers in cloud computing. IEEE Transact Cloud Comput 8(2):635–646
20. Burns B, Grant B, Oppenheimer D, Brewer E, Wilkes J (2016) Borg, omega, and kubernetes. Queue. 14(1):70–93
21. Piraghaj SF, Dastjerdi AV, Calheiros RN, Buyya R (2017) ContainerCloudSim An environment for modeling and simulation of containers in cloud data centers. Software: Practice and Experience. ;47(4):505–521
22. HewlettPackard.: Netperf. https://github.com/HewlettPackard/netperf
23. Badshah A, Ghani A, Shamshirband S, Aceto G, Pescapè A (2020) Performance-based service-level agreement in cloud computing to optimise penalties and revenue. IET Communicat 14(7):1102–1112
24. Zeng X, Garg S, Barika M, Zomaya AY, Wang L, Villari M et al (2020) SLA management for big data analytical applications in clouds: a taxonomy study. ACM Comput Survey (CSUR). 53(3):1–40
25. Zhao L, Sakr S, Liu A (2013) A framework for consumer-centric SLA management of cloud-hosted databases. IEEE Transact Serv Comput 8(4):534–549
26. Le TN, Sun X, Chowdhury M, Liu Z (2020) AlloX: compute allocation in hybrid clusters. In: Proceedings of the fifteenth european conference on computer Systems; 1–16
27. Dobrescu M, Egi N, Argyraki K, Chun BG, Fall K, Iannaccone G, et al (2009) RouteBricks: Exploiting parallelism to scale software routers. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles; 15–28
28. Khamse-Ashari J, Lambadaris I, Kesidis G, Urgaonkar B, Zhao Y (2017) Per-Server Dominant-Share Fairness (PS-DSF): A multi-resource fair allocation mechanism for heterogeneous servers. In: 2017 IEEE international conference on communications (ICC). IEEE; 1–7
29. Grandl R, Chowdhury M, Akella A, Ananthanarayanan G (2016) Altruistic scheduling in multi-resource clusters. In: 12th USENIX symposium on operating systems design and implementation (OSDI 16); 65–80
30. Wang W, Li B, Liang B, Li J (2016) Multi-resource fair sharing for datacenter jobs with placement constraints. In: SC'16: Proceedings of the international Conference for High Performance Computing, Networking, Storage and Analysis. IEEE; 1003–1014
31. Carvalho M, QoE-Aware Macedo DF (2021) Scheduler Container, for Co-located Cloud Environments. In, (2021) IFIP/IEEE international symposium on integrated network management (IM). IEEE :286–294
32. Kim YK, HoseinyFarahabady MR, Lee YC, Zomaya AY (2020) Automated fine-grained cpu cap control in serverless computing platform. IEEE Transact Parallel Distribut Syst 31(10):2289–2301
33. Lee K, Hong CH, Hwang J, Yoo C (2019) Dynamic network scheduling for virtual routers. IEEE Syst J 14(3):3618–3629