Tenant-Grained Request Scheduling in Software-Defined Cloud Computing

Huaqing Tu, *Student Member, IEEE*, Gongming Zhao, *Member, IEEE*, Hongli Xu, *Member, IEEE*, and Xianjin Fang

Abstract—Cloud providers host various services for tenants' requests (e.g., software-as-a-service) and seek to serve as many requests as possible for revenue maximization. Considering a large number of requests, the previous works on fine-grained request scheduling may lead to poor system scalability (or high schedule overhead) and break tenant isolation. In this article, we design a tenant-grained request scheduling framework to conquer the above two disadvantages. We formulate the tenant-grained request scheduling problem as an integer linear programming and prove its NP-hardness. We consider two complementary cases: the offline case (where we know all request demands in advance), and the online case (where we have to make immediate scheduling decisions for requests arriving online). A normalization-based algorithm with an approximation factor of $O(1)$ is proposed to solve the offline problem and a primal-dual-based algorithm with a competitive ratio of $[(1-\epsilon), O(\log 3 \cdot n + \log(1/\epsilon))]$ is designed for the online scenario, where $\epsilon \in (0,1)$ and n is the number of racks in the cloud. We also discuss how to integrate our proposed algorithms with the previous (fine-grained) request scheduling mechanism. Extensive simulation and experiment results show that our algorithms can obtain significant performance gains, e.g., the online algorithm reduces the scheduler's overhead more than 90% and achieves tenant isolation, while obtaining similar network performance (e.g., throughput) compared with the fine-grained request scheduling methods.

Index Terms—Software-defined cloud, cloud computing, request scheduling, scalability, approximation

1 Introduction

Cloud computing [1], [2], [3] has transformed a large part of the internet industry by providing services, such as infrastructure-as-a-service (IaaS) and software-as-a-service (SaaS) [4], [5], making it attract more and more attention from both academic and industry communities. The global cloud service market is expected to grow to nearly $\$528.4$ billion by 2022 [6]. In fact, the revenue from enterprise tenants (i.e., large-scale tenants) accounts for more than 90% of the total revenue [7]. In practice, enterprises will buy services from SaaS cloud providers for employees. Then the employees submit their requests to the cloud and the cloud scheduler is responsible for scheduling requests to proper servers [8], [9]. For clarity, each individual request represents a specific task, e.g., a model training task. Scheduling at the granularity of individual requests is referred to as *fine-grained request scheduling,mann2017resource*[11].

To pursue profit maximization, cloud providers seek to design efficient scheduling algorithms to serve as many requests as possible [9], [12], [13], [14]. With the help of software-defined technology [15], the scheduler implements the fine-grained request scheduling with high flexibility and efficiency. This problem has been widely studied in recent years for different targets, such as saving energy [10] [16], [17] [18], fault tolerance [9], [19] [20] and max-min fairness [21], [22], [23]. Though fine-grained request scheduling helps maximize the system profit and achieve better resource utilization, it still faces the following two challenges.

The first challenge is *system scalability*. Cloud services have already become an essential part of our life and more and more enterprises move their workloads to the cloud. For example, the number of active enterprise tenants on the Alibaba Cloud exceeds 500 thousand[24]. Even if a tenant submits only 100 individual requests per second, the scheduler needs to schedule 50 million individual requests in one second. Thus, if we perform fine-grained request scheduling, it will occur a massive amount of overhead, including scheduling decision delay and message consumption, on the scheduler, which may encounter the risk of overload.

The second challenge is *tenant isolation*. In a cloud, each tenant will generate a large number of individual requests. To increase resource utilization, cloud providers may dispatch multiple individual requests to the same server [19], no matter which tenants they come from. The interference on shared resources breaks tenant isolation and often leads to security vulnerabilities. For instance, Delimitrou et al. [25] present an application, called Bolt, that can detect the types

• Huaqing Tu, Gongming Zhao, and Hongli Xu are with the School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui 230027, China, and also with Suzhou Institute for Advanced Research, University of Science and Technology of China, Suzhou, Jiangsu 215123, China. E-mail: thq527@mail.ustc.edu.cn, {gmzhao, xuhongli}@ustc.edu.cn.
• Xianjin Fang is with the College of Computer Science and Engineering, Anhui University of Science and Technology, Huainan, Anhui 232000, China. E-mail: xjfang@aust.edu.cn.

Manuscript received 4 November 2021; revised 9 August 2022; accepted 10 August 2022. Date of publication 16 August 2022; date of current version 2 September 2022.
This work was supported in part by the National Science Foundation of China under Grants 62132019 and 62102392, in part by the Open Research of Projects of Zhejiang Lab under Grant 2022QA0AB04, and in part by the Natural Science Foundation of Jiangsu Province under Grant BK20210121.
(Corresponding author: Gongming Zhao.)
Recommended for acceptance by J. Zhai.
Digital Object Identifier no. 10.1109/TPDS.2022.3199031

and characteristics of applications sharing a server. The testing results show that Bolt can correctly identify the characteristics of 385 out of 436 diverse workloads on Amazon Web Services [25]. Leveraging this information, malicious tenants can enable a wide spectrum of network attacks, including denial of service attacks (DoS) [26] and co-residency attacks [27] with high success probability [25]. Thus, without considering tenant isolation, once a tenant generates malicious requests, it may affect the security of other tenants sharing resources on the same server.

To overcome the above challenges, in this paper, we intend to schedule requests at the tenant granularity (i.e., coarse-grained request scheduling) and try to make each server only serve requests of one tenant for better scalability and tenant isolation. Specifically, from the perspective of the cloud provider, if a tenant generates a set of individual requests, e.g., submitting multiple requests for training deep neural networks at the same time, these requests from the same tenant can be treated as a *tenant-grained request*. Since the number of tenants is much smaller than the number of individual requests, tenant-grained request scheduling can reduce the schedule overhead, thereby improving the system scalability. Moreover, since most large-scale enterprise tenants require far more resources than one server, we try to allocate the resources of each server to only one tenant for purpose of tenant isolation. Through extensive simulations, the performance gap between tenant-grained request scheduling and fine-grained request scheduling is within 3%, but tenant-grained request scheduling method can improve the security of tenants through tenant isolation and reduce the scheduling overhead by more than 90% (i.e., achieving better system scalability).

One may think that the existing scheduling algorithms [9], [12], [14], [16] for individual requests may be extended to implement tenant-grained request scheduling with some small modifications. However, it is not the case. Specifically, existing fine-grained request scheduling methods usually assume that each server can provide services for multiple individual requests. Thus, it can be modeled as a bin packing problem in which the bins and items correspond to the servers and individual requests, respectively [9], [12], [16]. However, in many practical scenarios, e.g., enterprise-oriented SaaS cloud [8], large-scale multi-tenant GPU clusters [28] and big data analysis cloud [29], the resources requested by one tenant may exceed the capacity of a single server. In other words, we may need to allocate several servers for one tenant-grained request. Thus, *tenant-grained request scheduling is fundamentally different from fine-grained request scheduling*. Moreover, since the power or bandwidth overload in a rack will affect the performance of the cloud (will be explained in detail in Section 2.3), this paper also considers the bandwidth and power constraints of the racks, which will make the problem far from trivial.

In this paper, we study the problem of tenant-grained request scheduling in software-defined cloud. To the best of our knowledge, this is the first work to schedule tenant-grained requests that require exclusive use of multiple servers while considering the bandwidth/power constraints on racks. With the help of tenant-grained request scheduling, we can significantly improve the system scalability and tenant security in the software-defined cloud. Note that, the proposed tenant-grained request scheduling algorithm can also be used in conjunction with the existing fine-grained scheduling algorithms, which will be discussed in Section 3.3. The main contributions of this paper are as follows:

1) We define the problem of tenant-grained request scheduling (TRS) in the software-defined cloud and prove its NP-hardness.
2) We present a normalization-based offline algorithm for the TRS problem, which guarantees that the maximum bandwidth load on any rack will not exceed the optimal solution by a factor of 3.3, and the allocated server and power resources will not exceed the rack's capacity by a factor of 3.3.
3) We propose an online algorithm based on the primal-dual method for TRS. The proposed algorithm can achieve $[(1-\epsilon), O(\log 3 \cdot n + \log(1/\epsilon))]$ competitiveness, where ϵ is an arbitrary positive value and n is the number of racks.
4) We show the high efficiency of the proposed algorithms through extensive simulations. For example, the online algorithm can reduce the schedule overhead more than 90%, while achieving tenant isolation, compared with the fine-grained request scheduling methods.

The rest of this paper is organized as follows. Section 2 introduces preliminaries and the definition of tenant-grained request scheduling. In Section 3, we propose an offline normalization-based tenant-grained request scheduling algorithm for the offline scenario. In Section 4, we design an online primal-dual request scheduling algorithm for the online scenario and analyze its approximate performance. Section 5 presents the extensive simulation and experiment results of the proposed algorithms, which show the superior performance of the tenant-grained request scheduling method. Section 6 reviews the related works. Section 7 concludes the paper.

2 Preliminaries

2.1 A Motivation Example

In this section, we give a motivation example to illustrate the contrast between fine-grained request scheduling and tenant-grained request scheduling. In this example, the cloud consists of a scheduler and four servers, as shown in Fig. 1. The resource capacity of each server is set as 10. Besides, there are two tenants marked with different colors, and each tenant submits six individual requests. We use numbers from one to twelve to label all individual requests, and "individual request" is abbreviated as "IR" in Fig. 1 for simplicity. The corresponding resource (e.g., CPU) demand of each individual request is marked after it.

If we perform the fine-grained request scheduling method [9], which selects a server with the least load, the scheduling results are shown in Fig. 1a. Specifically, individual requests 1/7/11 are scheduled to server 1; requests 2/8/12 are scheduled to server 2; requests 3/5/9 are scheduled to server 3; requests 4/6/10 are scheduled to server 4. From the scheduling results in Fig. 1a, we can get the following two observations: 1) For each individual request, the fine-grained request scheduling method needs to make a

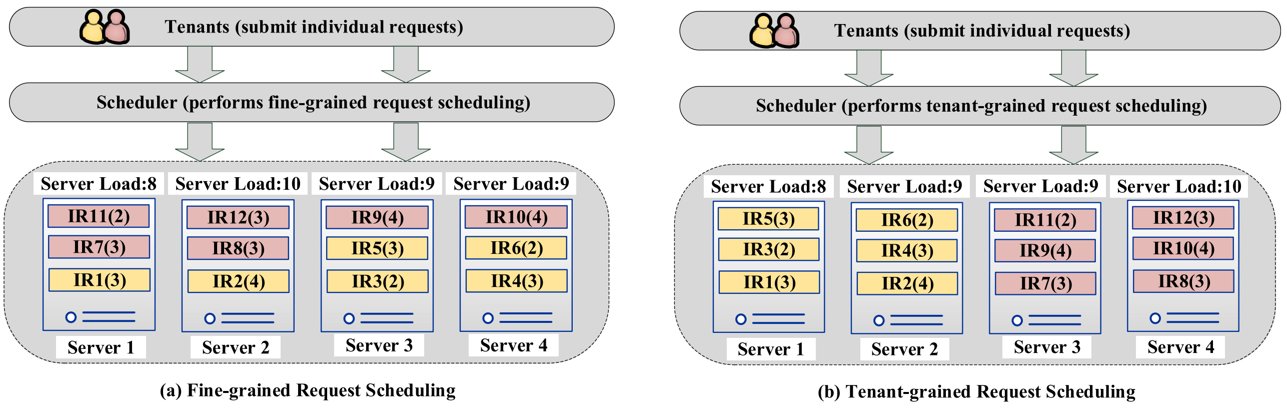


Fig. 1. Motivation Example. A tiny cloud consists of a scheduler and four servers. The resource capacity of each server is set as 10. There are two tenants, and each tenant submits six individual requests. We use numbers from one to twelve to label all individual requests, and “individual request” is abbreviated as “IR” in Fig. 1 for simplicity. The corresponding resource (e.g., CPU) demand of each individual request is marked after it. Fig. 1(a) and 1(b) show the scheduling results of fine-grained request scheduling and tenant-grained request scheduling, respectively.

scheduling decision and select a server among all servers in the system to handle it. When the number of individual requests is too large, the fine-grained request scheduling will cause high scheduling overhead, thereby reducing system scalability. 2) The individual requests of each tenant are distributed among all servers, and each server is shared by these two tenants, resulting in poor tenant isolation. A malicious tenant may attack the servers assigned to it, affecting other normal tenants.

These two observations make us think: Can we design a request scheduling method with *low scheduling overhead* and *high tenant isolation*? To this end, we propose the tenant-grained request scheduling method in this paper. This method first aggregates the individual requests from the same tenant into a tenant-grained request, then performs the proposed tenant-grained request scheduling algorithms (introduced in Sections 3 and 4) to allocate servers for each tenant-grained request. Fig. 1b exhibits the scheduling results of tenant-grained request scheduling. Since the number of tenants is much smaller than that of individual requests, tenant-grained request scheduling can greatly reduce scheduling overhead. Moreover, the scheduler tries to assign an entire server to a tenant-grained request for tenant isolation. Specifically, the former two servers are assigned to tenant 1, and the latter two servers are assigned to tenant 2. In this way, tenant-grained request scheduling can reduce scheduling overhead and guarantee tenant isolation. One may think that tenant isolation reduces resource utilization. However, the simulation results in Section 5 show that the tenant-grained request scheduling has a limited impact on the resource utilization of large-scale tenants. The discussion in Section 3.3 introduces how to improve the resource utilization of small-scale tenants.

2.2 Application Scenarios for Tenant-Grained Request Scheduling

We give some examples to illustrate that our algorithm can be applied to multiple scenarios. 1) In multi-tenant GPU clusters [28], tenants rent GPUs to train their models. GPUs represent a monolithic resource that cannot be shared at a fine granularity across users, which means that a GPU can only be used by one tenant at any time [28]. Our algorithm can be used to allocate GPUs to tenants. 2) For large-scale enterprise users, they purchase SaaS products (e.g., online video and group call conference) [7] from cloud service providers to facilitate enterprise management. Our algorithm can schedule requests at the granularity of tenants to improve the scalability of the scheduler. 3) In an enterprise's data center, such as Facebook [30], some servers process requests from Instagram users, and some servers process requests from WhatsApp users. With the help of our algorithm, the scheduler can implement request scheduling with low overhead.

2.3 System Model

Cloud Infrastructure Model: A software-defined cloud usually consists of multiple server racks $\mathcal{K} = \{k_1, k_2, ..., k_n\}$, with $n = |\mathcal{K}|$. Let C_k represent the number of servers in rack $k \in \mathcal{K}$. Since most of the servers are purchased at the same time by cloud service providers, the hardware specifications of servers in the cloud are similar and servers in a rack usually support the same service [31]. The assumption of server isomorphism is adopted by previous work like [32], [33]. Thus, we assume that all servers have similar hardware specifications and the services provided by the servers in the same rack are homogeneous. The heterogeneous scenario that the servers in the same rack have different hardware specifications will be studied in the future. The cloud provides a set of services $S = \{s_1, s_2, ...s_{|S|}\}$ for tenants. We use $\mathcal{K}_s$ to represent the set of racks with service $s \in S$.

We consider two resource constraints for a rack, bandwidth constraint and power constraint, respectively. First, to realize the communication between servers, each rack is equipped with a Top-of-Rack (ToR) switch [34], [35]. The traffic in and out of a rack needs to pass through the ToR switch, whose bandwidth capacity will significantly impact the throughput of a rack. Thus, we should consider the bandwidth capacity constraints of racks and such bandwidth capacity of rack k is denoted as B_k. Second, the power consumption of a rack is related to the load of servers on this rack and the power supply of each rack is also limited. If we schedule power-consuming requests that require a lot of computing resources to the servers in the same rack and the power load of the rack exceeds its power capacity, all servers in this rack will be affected, and the hardware may even be damaged [36], so the power capacity constraints of racks cannot be ignored. Let E_k represent the power capacity of rack k.

TABLE 1
Key Notations for Problem Definition

Parameters	Description
$\mathcal{K}$	a rack set
n	the number of racks
C_k	the number of servers in rack $k \in \mathcal{K}$
B_k	the bandwidth capacity of rack $k \in \mathcal{K}$
E_k	the power capacity of rack $k \in \mathcal{K}$
S	a network service type set
$\mathcal{K}_s$	the set of racks with service type $s \in S$
T	a tenant set
R	a tenant-grained request set
$p_s(r)$	the number of servers with service $s \in S$ required by request $r \in R$
$b_s(r)$	the bandwidth demand of request $r \in R$ for service $s \in S$
$e_s(r)$	the power demand of request $r \in R$ for service $s \in S$
y_r^k	the number of servers assigned to request $r \in R$ on rack $k \in \mathcal{K}$

Multi-Tenant Model: We consider a set of tenants renting various services from cloud providers and such tenant set is denoted as $T = \{t_1, t_2, ..., t_m\}$, where m is the number of tenants. In practice, a tenant $t \in T$ usually generates a great number of individual requests with different service requirements, especially for enterprise tenants. For clarity, we will regard all individual requests generated by the same tenant as a *tenant-grained request* in this paper. For ease of reference, the key notations used in this paper are summarized in Table 1.

2.4 Definition of Tenant-Grained Request Scheduling

This section provides a more precise description of the Tenant-grained Request Scheduling (TRS) problem. We aggregate all requests from the same tenant t_i into a tenant-grained request, denoted as r_i. Let $R = \{r_1, r_2, ...r_m\}$ represent the tenant-grained request set. Each tenant-grained request r may require multiple types of services with different resource requirements. The actual resource consumption of requests will be collected by the system as historical information to assist request scheduling. Using historical information, the scheduler can evaluate the resource demand of each request based on service type and collected historical information [11]. Specifically, we use $p_s(r)$ to represent the number of servers with service s required by request r. Let $b_s(r)$ and $e_s(r)$ denote the bandwidth demand and power demand of request r for service s, respectively. We schedule each request on servers as evenly as possible through ECMP. Thus, for any request $r \in R$ and service $s \in S$, the power and bandwidth consumption will be evenly distributed on the allocated servers. That is, each allocated server of service type s for request r needs to consume bandwidth resource of $\frac{b_s(r)}{p_s(r)}$ and power resource of $\frac{e_s(r)}{p_s(r)}$. High traffic volume leads to long queuing delay of data packets inside the ToR switch, thereby reducing the data transmission rate. Therefore, we take the bandwidth load balancing among ToR switches as the optimization goal. We formulate the TRS problem as follows:

$$\min \lambda$$

$$S.t. \begin{cases} \sum_{k\in\mathcal{K}_s} y_r^k = p_s(r), & \forall r \in R, s \in S \\ \sum_{r\in R} y_r^k \le C_k, & \forall k \in \mathcal{K} \\ \sum_{r\in R} y_r^k \cdot \frac{e_s(r)}{p_s(r)} \le E_k, & \forall k \in \mathcal{K}_s, s \in S \\ \sum_{r\in R} y_r^k \cdot \frac{b_s(r)}{p_s(r)} \le \lambda \cdot B_k, & k \in \mathcal{K}_s, s \in S \\ y_r^k \in \{0, 1, 2...\}, & \forall r \in R, k \in \mathcal{K} \end{cases} \tag{1}$$

Note that y_r^k means the number of servers assigned to request r on rack k. The first set of equations denotes that each request r needs to be allocated with $p_s(r)$ servers of service type s. The second set of inequalities denotes the constraints of the number of servers in each rack k. The third and fourth sets of inequalities means the power capacity constraint and bandwidth constraint in each rack, respectively. Our objective is to achieve bandwidth load balancing among ToR switches on racks, this is, min λ.

Theorem 1. *The TRS problem is NP-hard.*

To prove this theorem, we first give the definition of identical parallel machines scheduling problem.

Definition 1 (Identical Parallel Machines Scheduling (IPMS)[37]). *Given p parallel machines and q independent jobs, each job is to be assigned to one of the machines. All the parallel machines are identical in terms of their processing speed. Thus, every job will take the same amount of processing time on each machine. The objective is to find a schedule that minimizes the makespan.*

Proof. We prove NP-hardness of the TRS problem through a reduction from the IPMS problem. It means that TRS cannot be solved in deterministic polynomial time unless P = NP. Consider an instance of the IPMS problem: let $M = \{m_1, m_2, ..., m_p\}$ be a set of machines and $J = \{j_1.j_2, ..., j_q\}$ be a set of jobs. Moreover, the processing time of job $j \in J$ on an arbitrary machine is denoted as t_j. Now, we construct a special case of the TRS problem, in which each rack has unlimited resources, including power and bandwidth, and is equipped with enough servers. Under this case, the IPMS problem with the objective of minimizing the makespan is equivalent to the TRS problem with the objective of bandwidth load balancing among racks. Specifically, each machine m and job j can be regarded as a rack k and request r, respectively. The makespan on each machine can be regarded as the bandwidth load on each rack. Apparently, the reduction process above can be finished in polynomial time, since the IPMS problem is equivalent to the TRS problem under the special case. Note that the IPMS problem is a well-known NP-hard problem. Since IPMS is reducible to the special case of TRS, our TRS problem is NP-hard as well.pt?> □

3 Offline Algorithm Description

In this section, we propose an offline normalization-based request scheduling algorithm, called NTRS. Then we analyze the approximation performance of the NTRS algorithm.

TABLE 2
Key Notations for the NTRS Algorihtm

Parameters	Description
M_r^s	the unit request set of tenant-grained request r that requires service $s \in S$
M^s	the set of unit requests that requires service $s \in S$
M	the unit requests of all tenant-grained requests
$b_{r,k}^s$	the normalized bandwidth resource required by request r for service s
$e_{r,k}^s$	the normalized power resource required by request r for service s
$c_{r,k}^s$	the normalized server resource required by request r for service s
M_k^1	the set of unit requests that require more normalized bandwidth resources than the other two types of normalized resources
M_k^2	the set of unit requests that require more normalized power resources than the other two types of normalized resources
M_k^3	the set of unit requests that require more normalized server resources than the other two types of normalized resources

3.1 Algorithm Description

Due to the NP-hardness, it is difficult to optimally solve the TRS problem in polynomial time. This section presents a normalization-based tenant-grained requests scheduling algorithm, called NTRS, to solve the TRS problem. The basic idea of NTRS is to divide each tenant-grained request $r \in R$ into $\sum_{s\in S} p_s(r)$ subsets, each of which is called *unit request* and occupies one server. Let M_r^s denote the set of unit requests of request r that requires service $s \in S$. Obviously, $|M_r^s| = p_s(r), \forall r \in R, s \in S$. Let M^s denote the unit request set that requires service $s \in S$, i.e., $M^s = \bigcup_{r\in R} M_r^s$. The total unit requests from all tenant-grained requests are denoted by M, i.e., $M = \bigcup_{s\in S} M^s$. Each unit request $m \in M_r^s$ requires the same resource consumptions, that is, a single server with the bandwidth of $\frac{b_s(r)}{p_s(r)}$ and power of $\frac{e_s(r)}{p_s(r)}$. For ease of reference, we summarize the main notions used in the NTRS algorithm in Table 2.

The proposed NTRS algorithm mainly consists of two steps. The first step normalizes the resource requirements for each unit request $m \in M$ on different racks. Specifically, if we allocate one server on rack $k \in \mathcal{K}$ for a unit request $m \in M_r^s$, it will consume some server/power/bandwidth resources of this allocated rack. Thus, let $b_{r,k}^s$, $e_{r,k}^s$ and $c_{r,k}^s$ be the normalized bandwidth, power and server consumption of rack k, respectively, if one of the servers in rack k is allocated to the unit request in set M_r^s. These normalized resource requirements can be calculated as follows:

$$b_{r,k}^s = \frac{b_s(r)/p_s(r)}{B_k \cdot \widetilde{\lambda}}, \quad e_{r,k}^s = \frac{e_s(r)/p_s(r)}{E_k}, \quad c_{r,k}^s = \frac{1}{C_k}. \tag{2}$$

By replacing $y_r^k \in \{0,1,2...\}$ with $y_r^k \in [0, C_k^{max}]$ at the last line in Eq. (1), we can obtain the linear program LP_1 of Eq. (1), where C_k^{max} is the maximum number of servers placed in a rack, that is, $C_k^{max} = \max\{C_k, \forall k \in \mathcal{K}\}$. $\widetilde{\lambda}$ is the solution of LP_1, which can be solved in polynomial time. Let λ^* be the optimal solution of integer linear program Eq. (1). $\widetilde{\lambda}$ is the lower bound of λ^*, that is, $\widetilde{\lambda} \le \lambda^*$, since LP_1 relaxes the value range of variable y_r^k. Since $\widetilde{\lambda}$ is extremely close to the optimal solution λ^*, we use $\widetilde{\lambda}$ to replace λ^* in the following. Note that, if request $r \in R$ does not need the support of service $s \in S$ or servers in rack k does not support service s, we have $b_{r,k}^s = e_{r,k}^s = c_{r,k}^s = 0$. According to the normalized resource consumption in Eq. (2), we estimate the bottleneck resource when a unit request $m \in M_r^s$ is scheduled on rack k. Specifically, for each rack $k \in \mathcal{K}_s : s \in S$, we can divide the unit request set M^s into three subsets M_k^1, M_k^2 and M_k^3. Unit requests in M_k^1, M_k^2 and M_k^3 require more normalized bandwidth, power and server resources than the other two types of normalized resources, respectively. In other words, for each rack $k \in \mathcal{K}_s : s \in S$, we divide unit requests set M^s into three subsets as follows:

$$\begin{cases} M_k^1 = \{m | b_{r,k}^s \ge e_{r,k}^s, b_{r,k}^s \ge c_{r,k}^s, m \in M_r^s, r \in R\} \\ M_k^2 = \{m | e_{r,k}^s \ge b_{r,k}^s, e_{r,k}^s \ge c_{r,k}^s, m \in M_r^s, r \in R\} \\ M_k^3 = \{m | c_{r,k}^s \ge b_{r,k}^s, c_{r,k}^s \ge e_{r,k}^s, m \in M_r^s, r \in R\} \end{cases}. \tag{3}$$

The second step of NTRS allocates resources for each unit request according to the normalized resource consumptions. We use B_k^i, E_k^i and C_k^i to denote the cumulative normalized bandwidth, power and server consumption generated by unit requests in M_k^i, $i \in \{1,2,3\}$, when they are assigned to rack k, respectively. We rank racks in $\mathcal{K}_s$ with the increasing order of their bandwidth load. For each rack $k \in \mathcal{K}_s$, if 1) $m \in M_k^1$ and $B_k^1 < 1$ or 2) $m \in M_k^2$ and $E_k^2 < 1$ or 3) $m \in M_k^3$ and $C_k^3 < 1$, NTRS schedules the unit request m to the server in rack k and updates the corresponding cumulative value. The NTRS algorithm is formally described in Algorithm 1.

Algorithm 1. NTRS: Normalization-Based Tenant-Grained Request Scheduling

```
1:  **Step 1: Normalizing the resource demand**
2:  for each tenant-grained request $r \in R$ do
3:      for each service $s \in S$ do
4:          Devise unit request set $M_r^s$
5:  Construct a linear program $LP_1$ of Eq. (1) by replacing $y_r^k \in \{0,1,2...\}$ with $y_r^k \in [0, C_k^{max}]$
6:  Obtain the optimal solution $\lambda^*$ of $LP_1$
7:  for each tenant-grained request $r \in R$ do
8:      for each rack $k \in \mathcal{K}_s : s \in S$ do
9:          Normalize the server demand $c_{r,k}^s$, power demand $e_{r,k}^s$ and bandwidth demand $b_{r,k}^s$ with Eq. (2)
10: for each rack $k \in \mathcal{K}_s : s \in S$ do
11:     Divide the unit requests in $M^s$ into three sets $M_k^1$, $M_k^2$ and $M_k^3$ with Eq. (3)
12: **Step 2: Allocating Resources for Unit Requests**
13: for each request $r \in R$ and each service type $s \in S$ do
14:     for each unit request $m \in M_r^s$ do
15:         Rank racks in set $\mathcal{K}_s$ with the increasing order of bandwidth load
16:         for each $k \in \mathcal{K}_s$ do
17:             if $m \in M_k^1$ and $B_k^1 < 1$ then
18:                 Allocate resources for unit request $m$
19:             else if $m \in M_k^2$ and $E_k^2 < 1$ then
20:                 Allocate resources for unit request $m$
21:             else if $m \in M_k^3$ and $C_k^3 < 1$ then
22:                 Allocate resources for unit request $m$
23:             if unit request $m$ is allocated successfully then
24:                 Break
```

3.2 Performance Analysis of NTRS

This section analyzes the approximation performance of the proposed NTRS algorithm. By the definition in Eq. (3), we have the following lemma:

Lemma 2. *At any time in the execution of Algorithm 1, for the cumulative load from the unit requests in M_k^1 on rack $k \in \mathcal{K}$, we have $E_k^1 \leq B_k^1$, $C_k^1 \leq B_k^1$. For the cumulative load from the unit requests in M_k^2 on rack $k \in \mathcal{K}$, we have $B_k^2 \leq E_k^2$, $C_k^2 \leq E_k^2$. For the cumulative load from the unit requests in M_k^3 on rack $k \in \mathcal{K}$, we have $B_k^3 \leq C_k^3$, $E_k^3 \leq C_k^3$.*

Proof. On the one hand, by the definitions of M_k^1 in Eq. (3), we know that unit requests in set M_k^1 require more normalized bandwidth resources than the other two types of normalized resources. On the other hand, E_k^1, B_k^1 and C_k^1 denote the cumulative bandwidth, power and server consumption generated by unit requests that are in M_k^1 and assigned to rack k, respectively. Thus, we derive $E_k^1 \leq B_k^1$, $C_k^1 \leq B_k^1$. We can derive the other two conclusions in a similar way. □

Let constant h denote the number of servers that a rack can contain at least (and meet the power and bandwidth requirements). In other words, $h = \min\{\frac{1}{b_{r,k}^s}, \frac{1}{e_{r,k}^s}, \frac{1}{c_{r,k}^s}, \forall r \in R, k \in \mathcal{K}_s, s \in S\}$.

Lemma 3. *During the running of the NTRS algorithm, We have* $\max\{B_k^i, E_k^i, C_k^i, i \in \{1,2,3\}, k \in \mathcal{K}\} \leq 1 + \frac{1}{h}$.

Proof. We prove this lemma by induction on the number of unit requests. At the beginning of the NTRS algorithm, Lemma 3 is certainly satisfied. We assume that this lemme can hold before scheduling to a unit request $m \in M_r^s$. Let $k \in \mathcal{K}_s$ be the rack in which a server is allocated for unit request m. Unit request m may belong to any one of M_k^1, M_k^2 and M_k^3. Since these three cases are similar, we discuss the case where unit request m belong to M_k^1.

When $m \in M_k^1$, before scheduling, $B_k^1 \leq 1$ (otherwise unit request m would not be scheduled on rack k). After scheduling, its cumulative bandwidth load is $B_k^1 \leq 1 + b_{r,k}^s$. According to Lemma 2, $E_k^1 \leq 1 + b_{r,k}^s$ and $C_k^1 \leq 1 + b_{r,k}^s$. The other two cases can be discussed similarly. Thus, after scheduling unit request $m \in M_r^s$, we have

$$\begin{cases} \max_{k \in K}\{\max\{B_k^1, E_k^1, C_k^1\}\} \leq 1 + b_{r,k}^s \\ \max_{k \in K}\{\max\{B_k^2, E_k^2, C_k^2\}\} \leq 1 + e_{r,k}^s \\ \max_{k \in K}\{\max\{B_k^3, E_k^3, C_k^3\}\} \leq 1 + c_{r,k}^s \end{cases} \tag{4}$$

With the definition of h, Lemma 3 is proved. □

Theorem 4. *The proposed NTRS algorithm guarantees that the maximum bandwidth load on any rack $k \in \mathcal{K}$ will not exceed the optimal solution by a factor of $3 \cdot (1 + \frac{1}{h})$, and the allocated server and power resources will not exceed the rack's capacity by a factor of $3 \cdot (1 + \frac{1}{h})$.*

Proof. We use a variable ν_r^k to denote the number of servers that have been allocated for request $r \in R$ on rack $k \in \mathcal{K}$. According to Lemma 3, on any rack $k \in \mathcal{K}_s : s \in S$, the bandwidth load factor is as follows

$$\begin{aligned} &\sum_{r \in R} \nu_r^k \cdot b_{r,k}^s \\ &= \sum_{r \in M_k^1} \nu_r^k \cdot b_{r,k}^s + \sum_{r \in M_k^2} \nu_r^k \cdot b_{r,k}^s + \sum_{r \in M_k^3} \nu_r^k \cdot b_{r,k}^s \\ &= B_k^1 + B_k^2 + B_k^3 \leq 3 \cdot \left(1 + \frac{1}{h}\right) \end{aligned}$$

According to Eq. (2), it follows

$$\sum_{r \in R} \nu_r^k \cdot \frac{b_s(r)/p_s(r)}{B(k) \cdot \lambda^*} \leq 3 \cdot \left(1 + \frac{1}{h}\right)$$

Then, we have

$$\sum_{r \in R} \nu_r^k \cdot \frac{b_s(r)/p_s(r)}{B(k)} \leq 3 \cdot \left(1 + \frac{1}{h}\right) \cdot \lambda^*$$

Thus, the maximum bandwidth load factor on each rack will not exceed $3 \cdot (1 + \frac{1}{h})$ times as the optimal solution. Similarly, the allocated server and power resources will not exceed the rack's capacity by a factor of $3 \cdot (1 + \frac{1}{h})$. □

To deal with the situation where the resource constraints are violated, we can greedily remove the requests whose resource requirements are not satisfied from racks and servers until all resource constraints are not violated. In a request scheduling system, there is a queue for those requests waiting to be scheduled [29]. The removed request can re-enter the queue and wait for the next scheduling opportunity when there are enough resources.

Approximation factor: Following our analysis, by scheduling all requests, the NTRS algorithm can guarantee that the bandwidth load on any rack will not exceed the optimal solution by a factor of $3 \cdot (1 + \frac{1}{h})$. Similarly, the resource constraints will not be violated by a factor of $3 \cdot (1 + \frac{1}{h})$. This is also called bi-criteria approximation [38], [39], [40]. In a data center, a rack usually contains at least ten servers and provide bandwidth and power resources for these ten servers. Under this case, $h \geq 10$ holds and we have $3 \cdot (1 + \frac{1}{h}) \leq 3.3$. That is, the maximum bandwidth load through the NTRS algorithm on any rack will not exceed the optimal solution by a factor of 3.3, and the resource constraint will not be violated by a factor of 3.3 at most in many practical situations. In other words, our NTRS algorithm can achieve almost the constant bi-criteria approximation for the offline TRS problem in many practical situations.

Algorithm Running Example: For simplicity, we assume that there are two racks (k_1 and k_2), and four servers in each rack. All servers provide the same service (say, s). In addition, the bandwidth capacity and power capacity of each rack are set to 10 and 8, respectively. The resource demand of a tenant-grained request is represented by 3-tuple, i.e., (server's number, bandwidth demand, power demand). We assume that there are total three tenant-grained requests: $r_1 = (2, 4, 5)$, $r_2 = (2, 5, 4)$ and $r_3 = (3, 6, 6)$.

Following lines 2-4 of the NTRS algorithm described in Algorithm 1, we divide each tenant-grained request into a set of unit requests. Let M_r^s be the unit request set of tenant-grained request r and each unit request is also be represented as $m_r =$ (server's number, bandwidth demand, power demand). Thus, we have $M_{r_1}^s = \{m_{r_1}^1 = (1, 2, 2.5)$, $m_{r_1}^2 =$

$(1,2,2.5)\}$, $M^s_{r_2} = \{m^1_{r_2} = (1,2.5,2),\ m^2_{r_2} = (1,2.5,2)\}$ and $M^s_{r_3} = \{m^1_{r_3} = (1,2,2),\ m^2_{r_3} = (1,2,2),\ m^3_{r_3} = (1,2,2)\}$. According to lines 5-6, we set C^{max}_k as 4 and solve the linear program version of Eq. (1) by replacing $y^k_r \in \{0,1,2,...\}$ with $y^k_r \in [0, C^{max}_k]$. We obtain $\lambda^* = 0.75$. Following lines 7-9, we normalize the resource requirements of all unit requests using Eq. (2). The unit requests in $M^s_{r_1}$, $M^s_{r_2}$ and $M^s_{r_3}$ are normalized into (0.25, 0.26, 0.3125), (0.25, 0.33, 0.25) and (0.25, 0.26, 0.25), respectively. Then we divide unit requests into three subsets according to Eq. (3) for each rack. For rack k_1, we have $M^1_{k_1} = \{m^1_{r_2}, m^2_{r_2}, m^1_{r_3}, m^2_{r_3}, m^3_{r_3}\}$, $M^2_{k_1} = \{m^1_{r_1}, m^2_{r_1}\}$ and $M^3_{k_1} = \emptyset$. Similarly, for rack k_2, we have $M^1_{k_2} = \{m^1_{r_2}, m^2_{r_2}, m^1_{r_3}, m^2_{r_3}, m^3_{r_3}\}$, $M^2_{k_2} = \{m^1_{r_1}, m^2_{r_1}\}$ and $M^3_{k_2} = \emptyset$. Now, we follow lines 13-24 in Algorithm 1 to allocate resources for each unit request. We start from unit requests belonging to tenant-grained request r_1. The sequence of racks sorted according to bandwidth load is $[k_1, k_2]$. Since $m^1_{r_1} \in M^2_{k_1}$ and the cumulative power load $B^2_{k_1} < 1$, a server in k_1 will be assigned to $m^1_{r_1}$. Then, $B^2_{k_1}$, $E^2_{k_1}$ and $C^2_{k_1}$ are updated to 0.26, 0.3125, 0.25. The sequence of racks sorted according to bandwidth load is updated to $[k_2, k_1]$. Since $m^2_{r_1} \in M^2_{k_2}$ and the cumulative power load $B^2_{k_2} < 1$, a server in k_2 will be assigned to $m^2_{r_1}$. Similar to the steps above, we allocate servers for the unit requests of tenant-grained requests r_2 and r_3. The final scheduling result is as follows: $m^1_{r_1}$, $m^1_{r_2}$, $m^1_{r_3}$ and $m^2_{r_3}$ are scheduled on rack k_1; $m^2_{r_1}$, $m^2_{r_2}$ and $m^3_{r_3}$ are scheduled on rack k_2. That is, one server on each of racks k_1 and k_2 is assigned to the tenant-grained request r_1. One server on each of racks k_1 and k_2 is assigned to r_2. Two servers on rack k_1 and one server on rack k_2 are assigned to r_3.

3.3 Discussion

We give the following discussion to enhance the applicability and practicality of our proposed algorithm.

Dealing with Small-scale Tenants: For small-scale tenants, tenant isolation requirement may reduce resource utilization. To deal with the waste of resources caused by small-scale tenants, we use the following steps to achieve the trade-off between resource utilization and tenant isolation. Specifically, in the first step, we calculate the resource utilization of each small-scale tenant. Let u_t be the number of servers allocated to the tenant-grained request of tenant t. Let u'_t be the minimum number of servers ideally consumed by tenant t, which can be a decimal. The ideal minimum number of servers required by a tenant is obtained by calculation. The scheduler evaluates the resource demand of each tenant based on service type. u'_t is calculated through dividing the total resource requirements by the resource amount owned by a server. The resource utilization of tenant t can be formulated as u'_t/u_t. In the second step, we pick out those tenants whose resource utilization is below a threshold, then aggregate the individual requests from these tenants into a new tenant-grained request. It should be noted that there is a trade-off between resource utilization and tenant isolation in this step. The more tenants' individual requests are aggregated, the higher the resource utilization, but the lower the tenant isolation. The threshold for aggregating the individual requests from small-scale tenants is obtained by experience. System administrators can set the threshold according to experience based on historical data such that the resource utilization is improved while ensuring proper tenant isolation. At the last step, we can perform tenant-grained request scheduling algorithms proposed in Sections 3 and 4 to allocate resources for the aggregated requests. In a request scheduling system, there is a queue for requests to be scheduled [29]. Under the online scheduling scenario, the aggregated objects are currently queued requests submitted by tenants.

Combining With Fine-Grained Request Scheduling. We should note that the fine-grained request scheduling algorithm can be used as a complementary scheme of our tenant-grained request scheduling algorithm. Specifically, we first use our proposed algorithm for scheduling tenants' traffic. Under this case, some servers may be under-utilization. To improve the resource utilization on some servers, we can schedule some requests individually in a fine-grained manner, e.g., scheduling an individual request from one server to another. Section 5.2.4 conduct a set of simulations for the combination of tenant-grained request scheduling combined and fine-grained request scheduling. It shows that, combined with fine-grained request scheduling, tenant-grained request scheduling can achieve similar system throughput performance to using only fine-grained request scheduling.

4 Online Algorithm Description

4.1 Problem Definition

For each arrival tenant-grained request, the scheduler will construct a set of scheduling schemes. Now we use an example to illustrate the scheduling scheme. Assume that there are two racks $\{k_1, k_2\}$ and a tenant-grained request $r \in R$ which needs two servers. One of the scheduling schemes is that two servers on rack k_1 are allocated for request r, which is represented as $[(k_1, 2)]$. Similarly, we can list the other two schemes as $[(k_1, 1), (k_2, 1)]$ and $[(k_2, 2)]$. Note that only when the type of server on the rack matches the service type required by the request, the server on this rack will be assigned to the request. Let D_r be the scheduling scheme set of request $r \in R$. If all potential schemes are explored, the size of D_r may be very large. We will present an algorithm called SCH in Section 4.3 to construct a scheduling scheme set with low cost, and give its performance analysis.

In the online scenario, we focus on maximizing the total reward, such as system throughput and the number of processed requests. The reward of request $r \in R$ is denoted as $w(r)$. $I(r,d,k)$ denotes the number of servers assigned to the request r on the rack k according to the scheme $d \in D_r$. Since the resources (e.g., bandwidth capacity) are limited, not all requests will be served in the cloud. Thus, we use the total reward as the performance metric of the online algorithm. We describe the online TRS problem as follows:

$$\max \sum_{r \in R} \sum_{d \in D_r} x^d_r \cdot w(r)$$

$$S.t. \begin{cases} \sum_{d \in D_r} x^d_r \le 1, & \forall r \in R \\ \sum_{r \in R} \sum_{d \in D_r} x^d_r I(r,d,k) \le C_k, & \forall k \in \mathcal{K}_s, s \in S \\ \sum_{r \in R} \sum_{d \in D_r} x^d_r I(r,d,k) \frac{b_s(r)}{p_s(r)} \le B_k, & \forall k \in \mathcal{K}_s, s \in S \\ \sum_{r \in R} \sum_{d \in D_r} x^d_r I(r,d,k) \frac{e_s(r)}{p_s(r)} \le E_k, & \forall k \in \mathcal{K}_s, s \in S \\ x^d_r \in \{0,1\}, & \forall r \in R, d \in D_r \end{cases} \tag{5}$$

TABLE 3
Key Notations for the PDRA Algorihtm

Parameters	Description
D_r	the scheduling scheme set of request $r \in R$
$w(r)$	the reward of request $r \in R$
$I(r, d, k)$	the number of servers assigned to the request r on the rack k according to the scheme $d \in D_r$
x_r^d	whether request $r \in R$ chooses the scheduling scheme $d \in D_r$ or not
P_d	the profit of scheduling scheme $d \in D_r$
d^*	the scheduling scheme with the maximum profit, i.e., $d^* = \arg\max_{d \in D_r} P_d$
$\mathcal{K}_{r,s}$	the candidate rack set with service type $s \in S$ for the request $r \in R$
$q_{k,s}$	the resource consumed by a unit request for service $s \in S$ on rack $k \in \mathcal{K}_s$

Note that x_r^d means that whether the tenant-grained request $r \in R$ chooses the scheduling scheme d or not. The first set of inequalities denotes that a tenant-grained request will choose at most one scheduling scheme. The second set of inequalities means that the number of servers that will be allocated on a rack cannot exceed the rack's capacity. The third and fourth sets of inequalities denote the bandwidth and power constraints on racks, respectively. Our objective is to maximize the total reward of tenant-grained requests with resource constraints.

Algorithm 2. PDRA: Online Primal-Dual Algorithm

```
 1: Step 1: Algorithm Initialization
 2: Set constant $F^*$ by Eq. (7)
 3: Set constant $\mathcal{N}$ with $3 \cdot |\mathcal{K}|$, and $\epsilon \in (0, 1)$
 4: Initialize the dual variables:
 5: $\alpha_r = 0$, $\beta_k = 0$, $\delta_k = 0$, $\theta_k = 0, \forall r, k$
 6: Step 2: Determining a Scheduling Scheme for the
    Tenant-grained Requests
 7: for each arrival of request $r \in R$ do
 8:    Use the SCH algorithm to compute scheduling scheme
       set $D_r$
 9:    Calculate the price of each scheduling scheme in $D_r$
       according to Eq. (8)
10:    Set $P = \max_{d \in D_r} P_d$
11:    if $P <= 0$ then
12:       Reject the request $r$
13:    else
14:       $d^* = \arg\max_{d \in D_r} P_d$
15:       Schedule request $r$ according to $d^*$
16:       Update $\alpha_r$ as $P$
17:       Update $\beta_k$, $\delta_k$ and $\theta_k$ by Eqs. (9), (10) and (11)
```

4.2 Algorithm Description

We present an online primal-dual request scheduling algorithm called PDRA to solve Eq. (5). For ease of reference, we summarize the main notions used in the PDRA algorithm in Table 3. We first construct the dual problem for the linear relaxation of Eq. (5). For the optimization objective and each constraint in the TRS problem defined in Eq. (5), there is a dual variable [41]. Specifically, the dual variable α_r is for the objective, $\beta_k, \delta_k, \theta_k$ are for the three sets of constraints, respectively. Note that all these dual variables are non-negative. We describe the dual problem as follows:

$$\min \sum_{r \in R} \alpha_r + \sum_{k \in \mathcal{K}_s} \sum_{s \in S} (C_k \cdot \beta_k + B_k \cdot \delta_k + E_k \cdot \theta_k)$$

$$S.t. \begin{cases} \alpha_r \geq w(r) - \sum_{k \in \mathcal{K}_s} \sum_{s \in S} I(r, d, k) \beta_k \\ \quad - \sum_{k \in \mathcal{K}_s} \sum_{s \in S} I(r, d, k) \frac{b_s(r)}{p_s(r)} \delta_k \\ \quad - \sum_{k \in \mathcal{K}_s} \sum_{s \in S} I(r, d, k) \frac{e_s(r)}{p_s(r)} \theta_k, \quad \forall r \in R, d \in D_r \\ \alpha_r \geq 0, \beta_k \geq 0, \delta_k \geq 0, \theta_k \geq 0, \quad \forall r \in R, k \in \mathcal{K} \end{cases} \tag{6}$$

The PDRA algorithm first initializes the dual variables and corresponding constants F^*, $\mathcal{N}$ and ϵ. According to the first set of inequalities in Eq. (6), we define constant F^* in Eq. (7) as the maximum usage of each resource over all possible schemes of all requests. The constant $\mathcal{N}$ represents the number of inequalities from the second set to the fourth set in Eq. (5). That is, $\mathcal{N} = 3 \cdot |\mathcal{K}|$. The constant $\epsilon \in [0, 1]$ denotes a trade-off between resource violation and reward.

$$F^* = \max_{r,d,s} \left\{ \max_k \frac{I(r, d, k)}{w(r)}, \max_k \frac{I(r, d, k) \cdot e_s(r)}{p_s(r) \cdot w(r)}, \max_k \frac{I(r, d, k) \cdot b_s(r)}{p_s(r) \cdot w(r)} \right\} \tag{7}$$

The second step of PDRA is to choose a scheduling scheme with the maximum profit for each request. When request $r \in R$ comes, the PDRA algorithm derives a scheduling scheme set D_r by using the SCH algorithm, which will be introduced in Section 4.3. Then, the PDRA algorithm calculates the profit of each scheme $d \in D_r$. The profit means the utility over all types of resources for the request $r \in R$. We denote the profit of scheme $d \in D_r$ as follows:

$$P_d = w(r) - \sum_{k \in \mathcal{K}_s} \sum_{s \in S} I(r, d, k) \left[\frac{b_s(r)}{p_s(r)} \delta_k + \beta_k + \frac{e_s(r)}{p_s(r)} \theta_k \right]. \tag{8}$$

Then we determine the maximum profit of all possible scheduling schemes for request $r \in R$, that is, $P = \max_{d \in D_r} P_d$. If P is negative, it does not bring any profit by scheduling this request. So, we reject this request. Otherwise, we will choose an efficient scheduling scheme for request $r \in R$. In a request scheduling system, there is a queue for those requests waiting to be scheduled [29]. The rejected requests can re-enter the queue and wait for the next scheduling opportunity when there are enough resources. We use d^* to represent the scheduling scheme with the maximum profit P_d, that is, $d^* = \arg\max_{d \in D_r} P_d$. Next, we allocate resources for the tenant-grained request according to scheme d^*. Then, we update the dual variables α_r as the maximum profit. Meanwhile, since the available resources of racks are used by request r, the corresponding dual variables will increase accordingly. To avoid the confusion, we use β_k and β_k' to denote the dual variables before and after scheduling request r. The dual variable update of server resource is as follows:

$$\beta_k' = \beta_k \left[1 + \frac{I(r, d^*, k)}{C_k} \right] + \epsilon \cdot \frac{I(r, d^*, k)}{\mathcal{N} \cdot C_k \cdot F^*}, \quad \forall k \in \mathcal{K} \tag{9}$$

For the bandwidth resource and power resource on rack k, the dual variable updates are as follows:

$$\delta_k' = \delta_k\left[1+\frac{I(r,d^*,k)\cdot b_s(r)}{B_k\cdot p_s(r)}\right] + \epsilon\cdot\frac{I(r,d^*,k)\cdot b_s(r)}{\mathcal{N}\cdot B_k\cdot F^*\cdot p_s(r)} \tag{10}$$

$$\theta_k' = \theta_k\left[1+\frac{I(r,d^*,k)\cdot e_s(r)}{E_k\cdot p_s(r)}\right] + \epsilon\cdot\frac{I(r,d^*,k)\cdot e_s(r)}{\mathcal{N}\cdot E_k\cdot F^*\cdot p_s(r)}. \tag{11}$$

In Eqs. (10) and (11), δ_k' and θ_k' denote the values of δ_k and θ_k after the resources are occupied by request r, respectively.

Algorithm 3. SCH: Constructing Scheduling Scheme Set for Request $r \in R$

```
1: Step 1: Computing Candidate Rack Set
2: for each type service $s \in S$ do
3:     Initialize the candidate rack set $\mathcal{K}_{r,s}$ to the empty set
4:     for each rack $k \in \mathcal{K}_s$ do
5:         Compute the resource consumption denoted as $q_{k,s}$ on
           rack $k$ for the unit request $m_{r,s}$
6:         Update $\varphi$ as the maximum resource consumption of
           rack $k \in \mathcal{K}_{r,s}$
7:         if $|\mathcal{K}_{r,s}| < p_s(r)$ then
8:             Put rack $k$ into $\mathcal{K}_{r,s}$
9:         else if $q_{k,s} < \varphi$ then
10:            Put rack $k$ into $\mathcal{K}_{r,s}$ and remove the rack with
               maximum resource consumption
11: Step 2: Computing Scheduling Scheme Set
12: Devise scheduling scheme set $D_r$ based on candidate rack
    set and Theorem 6
```

The scheduler has the ability to detect whether a request has been finished [29]. Once a request ends, the resources occupied by this request will be released, and the dual variables of the released resources will be decreased accordingly. The reduction of dual variables is based on Eqs. (9), (10), and (11). We take the server resource as an example. To avoid the confusion, we use β_k and β_k'' to denote the dual variables before and after server resource are released, respectively. Similar to Eq. (9), we have

$$\beta_k = \beta_k''\left[1+\frac{I(r,d^*,k)}{C_k}\right] + \epsilon\cdot\frac{I(r,d^*,k)}{\mathcal{N}\cdot C_k\cdot F^*}, \quad \forall k\in\mathcal{K} \tag{12}$$

Then, we transform Eq. (12) into

$$\beta_k'' = \left[\beta_k - \epsilon\cdot\frac{I(r,d^*,k)}{\mathcal{N}\cdot C_k\cdot F^*}\right]\Big/\left[1+\frac{I(r,d^*,k)}{C_k}\right] \quad ,\forall k\in\mathcal{K}. \tag{13}$$

Similarly, let δ_k and δ_k'' be the dual variables before and after the bandwidth resource are released, respectively. θ_k and θ_k'' be the dual variables before and after the power resource are released, respectively. Then, the update of dual variables δ_k and θ_k are as follows.

$$\delta_k'' = \left[\delta_k - \epsilon\cdot\frac{I(r,d^*,k)\cdot b_s(r)}{\mathcal{N}\cdot B_k\cdot F^*\cdot p_s(r)}\right]\Big/\left[1+\frac{I(r,d^*,k)\cdot b_s(r)}{B_k\cdot p_s(r)}\right] \tag{14}$$

$$\theta_k'' = \left[\theta_k - \epsilon\cdot\frac{I(r,d^*,k)\cdot e_s(r)}{\mathcal{N}\cdot E_k\cdot F^*\cdot p_s(r)}\right]\Big/\left[1+\frac{I(r,d^*,k)\cdot e_s(r)}{E_k\cdot p_s(r)}\right]. \tag{15}$$

After a tenant-grained request quits, the dual variables can be updated according to Eqs. (13), (14), and (15).

4.3 Construction of Scheduling Scheme Set

In this section, we introduce how to construct the scheduling scheme set for a tenant-grained request with low cost. To this end, we design the SCH algorithm in Algorithm 3 to derive a set of scheduling schemes. The basic idea of SCH is to first get a rack set, which is called candidate rack set, for each type of service $s \in S$ required by request $r \in R$. The racks in candidate rack set have the minimum resource consumption. Then SCH builds a set of scheduling schemes on the basis of candidate racks. We denote the candidate rack set with service type $s \in S$ for the request $r \in R$ as $\mathcal{K}_{r,s}$, whose size is $p_s(r)$. Note that the resource consumption includes the consumption of server, bandwidth and power. To evaluate the resource consumption of rack $k \in \mathcal{K}_s$ for $r \in R$, we use a unit request of the tenant-grained request, which requires a single server with bandwidth of $\frac{e_s(r)}{p_s(r)}$ and power of $\frac{b_s(r)}{p_s(r)}$. For ease of expression, we denote a unit request of the tenant-grained request $r \in R$ with service type $s \in S$ as $m_{r,s}$. The resource consumption of $m_{r,s}$ on rack k is denoted as:

$$q_{k,s} = \frac{e_s(r)}{p_s(r)}\cdot\theta_k + \beta_k + \frac{b_s(r)}{p_s(r)}\cdot\delta_k$$

The optimal scheduling scheme must satisfy the following two theorems, which can be used to greatly reduce the size of the scheduling scheme set D_r.

Theorem 5. *For any service $s \in S$ required by request $r \in R$, racks in the optimal scheduling scheme must belong to $\mathcal{K}_{r,s}$.*

Proof. We prove this theorem by contradiction. Let d_r' be the scheduling scheme with the maximum profit. We use $\mathcal{K}_{d',s}$ to denote the rack set with service type $s \in S$ in scheme d'. For a rack $k \in \mathcal{K}_{d',s}$, the number of servers allocated for the request r is denoted as t_k'. Let $\mathcal{K}^1_{d',s}$ be the rack set which is not included in $\mathcal{K}_{r,s}$, and is a subset of $\mathcal{K}_{d',s}$. We use $q_{k,s}'$ to denote the resource consumption of unit request $m_{r,s}$ on rack $k \in \mathcal{K}_{d',s}$. The profit of scheme d_r' is denoted as P_d'.

We construct another scheduling scheme denoted as d_r. The difference between d_r and d_r' is that we use the rack in $\mathcal{K}_{r,s}$ to replace the rack in $\mathcal{K}^1_{d',s}$. We denote this new rack set as $\mathcal{K}^2_{d,s}$. The profit of scheme d_r is denoted as $P_{d,r}$. Then, we can get the difference between $P_{d,r}$ and $P_{d,r}'$:

$$P_{d,r} - P_{d,r}' = \sum_{k\in\mathcal{K}^1_{d',s}} q_{k,s}'\cdot t_k' - \sum_{k\in\mathcal{K}^2_{d,s}} q_{k,s}\cdot t_k' \geq 0.$$

The last inequality is satisfied because the resource consumption $q_{k,s}$ of the rack in $\mathcal{K}^2_{d,s}$ is smaller than that in $\mathcal{K}^1_{d',s}$, that is, $q_{k,s} \leq q_{k,s}'$. This means that scheme d_r has a higher profit than that of d_r', which contradicts the assumption. □

Let $\mathcal{K}^*_{r,s}$ be the rack set with service type $s \in S$ in the optimal scheduling scheme d^*. Assume that there are two racks k_i and k_j in $\mathcal{K}^*_{r,s}$. We denote the number of servers allocated for request $r \in R$ on rack k_i and k_j as t_{k_i} and t_{k_j}. For the optimal scheme d^*, we have the following theorem:

Theorem 6. *In the optimal scheduling scheme, for any service type $s \in S$, if the resource consumption on rack k_i is greater than or equal to that on rack k_j, that is, $q_{k_i,s} \geq q_{k_j,s}$, the number* *of servers allocated on rack k_i for request r is smaller than or equal to that on rack k_j, that is, $t_{k_i} \le t_{k_j}$.*

Proof. We prove this theorem by contradiction. Let d''_r be the scheduling scheme of request r with the maximum profit. For any service type $s \in S$, the rack set of d''_r is denoted as $\mathcal{K}''_{r,s}$. Assume that there are two racks k''_i and k''_j in $\mathcal{K}''_{r,s}$. The resource consumption of each rack is denoted as $q''_{k_i,s}$ and $q''_{k_j,s}$, which satisfies $q''_{k_i,s} \ge q''_{k_j,s}$. We denote the number of servers allocated for request r on rack k''_i and k''_j as $t_{k''_i}$ and $t_{k''_j}$, which satisfies $t_{k''_i} \ge t_{k''_j}$. The profit of scheme d''_r is denoted as $P''_{d,r}$.

We construct another scheduling scheme denoted as d_r. These two schemes have the same rack set. The number of servers allocated for the request r on rack k''_i is t''_{k_j} and the number of servers allocated for the request r on rack k''_j is t''_{k_i}. The profit of scheme d_r is denoted as $P_{d,r}$. We have

$$\begin{aligned} P_{d,r} - P''_{d,r} &= q''_{k_i,s} \cdot t''_{k_i} + q''_{k_j,s} \cdot t''_{k_j} - q''_{k_i,s} \cdot t''_{k_j} - q''_{k_j,s} \cdot t''_{k_i} \\ &= (q''_{k_i,s} - q''_{k_j,s}) \cdot (t''_{k_i} - t''_{k_j}) \ge 0. \end{aligned}$$

The last inequality is satisfied because of $q''_{k_i,s} \ge q''_{k_j,s}$ and $t''_{k_i} \ge t''_{k_j}$. This means that the profit of d_r is greater than that of d''_r, which contradicts the assumption. □

Since we can only construct schemes satisfying Theorems 5 and 6, the size of the scheduling scheme set will be greatly reduced. Now, we analyze the time complexity of SCH. The first step of SCH takes $O(|\mathcal{N}|)$ to traverse all racks to get the candidate rack set for the tenant-grained request. In order to reduce time consumption, we can pre-compute the scheduling scheme set for candidate rack sets of different sizes, which makes the second step of SCH takes constant time. Thus, the time complexity of SCH is $O(|\mathcal{N}|)$. Since the rack in the candidate rack set has the least resource consumption, the scheduling scheme in D_r has a higher profit, that is, the value of P_d is larger. To further reduce the time consumed by PDRA, we can specify the size of D_r, e.g., 20.

4.4 Performance Analysis of PDRA

In this section, we analyze the performance of PDRA. We first give the following lemma.

Lemma 7. *For all dual variables, the update rules in the online algorithm are dual feasible.*

Proof. For dual variables β_k, δ_k and θ_k, their values are nonnegative, as they increase from zero during the update. For a request r, if it is rejected, we know that P_d is negative for $\forall d \in P_d$. If it is accepted, we set α_r with the maximum value of P_d. Moreover, further updates can only make the right hand of Eq. (8) to be smaller, thus preserving the feasibility of the constraints. □

Lemma 8. *The objective value of Eq. (6) will be increased no more than $1+\epsilon$ whenever a request r is accepted.*

Proof. According the the update rules of the dual variables, once a request r has been accepted and resource are allocated with the scheme d^*. Let k_s be a rack, on which the servers can provide service $s \in S$. We denote the incremental objective value the dual problem as Δ.

$$\begin{aligned} \Delta = w(r) &+ \sum_{k_s \in d^*} \sum_{s \in S} \epsilon \cdot \frac{I(r,d,k)}{\mathcal{N} \cdot F^*} \\ &+ \sum_{k_s \in d^*} \sum_{s \in S} \epsilon \cdot \frac{I(r,d,k) \cdot e_s(r)/p_s(r)}{\mathcal{N} \cdot F^*} \\ &+ \sum_{k_s \in d^*} \sum_{s \in S} \epsilon \cdot \frac{I(r,d,k) \cdot b_s(r)/p_s(r)}{\mathcal{N} \cdot F^*} \\ &\le w(r) + \frac{\epsilon \cdot \mathcal{N} \cdot F^*}{\mathcal{N} \cdot F^*} \cdot w(r) = (1+\epsilon) \cdot w(r) \end{aligned}$$

□

For each request $r \in R$ and each rack $k \in \mathcal{K}$, we use $L(k,r)$ and $\beta(k,r)$ to denote the number of servers which have been occupied on the rack k (after the request has been processed) and the value of β_k, respectively. Similarly, we define the bandwidth load on rack k and the value of δ_k as $G(k,r)$ and $\delta(k,r)$, respectively. We also define $H(k,r)$ as the total power consumption on rack k. Let $\beta(k,r)$ and $\delta(k,r)$ be β_k and δ_k, respectively.

Lemma 9. *For request $r \in R$ and each rack k, we have*

$$\begin{cases} \beta(k,r) \ge \epsilon \cdot \frac{exp[L(k,r)/C_k]-1}{\mathcal{N} \cdot F^*} \\ \delta(k,r) \ge \epsilon \cdot \frac{exp[G(k,r)/F_k]-1}{\mathcal{N} \cdot F^*} \\ \theta(k,r) \ge \epsilon \cdot \frac{exp[H(k,r)/E_k]-1}{\mathcal{N} \cdot F^*} \end{cases}. \tag{16}$$

Proof. We prove the first set of inequalities in Eq. (9) by induction of aggregated request r_i. In the initial stage of the algorithm, we set $\beta(k,r_0) = L(k,r_0) = 0$. Thus, the inequalities hold. We consider the situation in which a request r_i arrives. If request r_i is rejected, the inequality is still satisfied. If it is accepted, we have

$$\begin{aligned} L(k,r_i) &= L(k,r_{i-1}) + I(r_i,d^*,k) \\ \beta(rk,r_i) &= \beta(k,r_{i-1}) \left[1 + \frac{I(r,d^*,k)}{C(k)}\right] \\ &\quad + \epsilon \cdot \frac{I(r,d^*,k)}{\mathcal{N} \cdot C(k) \cdot F^*} \end{aligned}$$

By induction hypothesis, it follows

$$\begin{aligned} &\beta(k,r_i) \\ &\ge \epsilon \cdot \frac{exp[L(k,r_{i-1})/C(k)]-1}{\mathcal{N} \cdot F^*} \left[1 + \frac{I(r,d^*,k)}{C(k)}\right] \\ &\quad + \epsilon \cdot \frac{I(r,d^*,k)}{\mathcal{N} \cdot C(k) \cdot F^*} \\ &= \epsilon \cdot \frac{1}{\mathcal{N} \cdot F^*} \left\{ exp\left[\frac{L(k,r_{i-1})}{C(k)}\right] \left[1 + \frac{I(r,d^*,k)}{C(k)}\right] - 1 \right\} \\ &\approx \epsilon \cdot \frac{1}{\mathcal{N} \cdot F^*} \left\{ exp\left[\frac{L(k,r_{i-1})}{C(k)}\right] exp\left[\frac{I(r,d^*,k)}{C(k)}\right] - 1 \right\} \\ &= \epsilon \cdot \frac{exp[L(k,r)/C(k)]-1}{\mathcal{N} \cdot F^*}. \end{aligned}$$

The proof of the second and third sets of inequalities in Eq. (9) is similar with the above analysis. Thus, the proof of the following two sets of inequalities is omitted here. □

We give the definition of competitive ratio for an online algorithm according to [42].

Definition 2. *If an online algorithm achieves at least $\zeta \cdot OPT$, where OPT is the optimal result for TRS, and the server resource constraint, the bandwidth capacity constraint and the power capacity constraint are violated by a multiplicative factor η, we call that the online algorithm is $[\zeta, \eta]$ competitive.*

The parameter ζ measures how much we lose by restricting access to the request information in the online case, and parameter η measures the extent to which the algorithm may overload the system by violating certain resource constraints.

Theorem 10. *For any specified $\epsilon \in (0,1)$, the online algorithm is $[(1-\epsilon), O(\log 3 \cdot n + \log(1/\epsilon))]$ competitive.*

Proof. Whenever request $r \in R$ is accepted, the objective value of Eq. (5) will be increased with $w(r)$. According to Lemma 8, the objective value of the dual problem increases less than $(1+\epsilon) \cdot w(r)$. Therefore, the overall objective value of Eq. (5) is at least $(1-\epsilon)$ times as that of Eq. (6). Thus, we know that the reward of the PDRA algorithm is at least $(1-\epsilon) \cdot OPT$, where OPT is the optimal result for the PDRA algorithm.

According to Eq. (9), it follows that $\beta_k \leq 1 + F^*$ in any iteration of the PDRA algorithm. Combining with the above analysis and Lemma 9, we have

$$\begin{aligned}\frac{L(k,r)}{C_k} &\leq \log\left[\frac{\beta_k \cdot \mathcal{N} \cdot F^*}{\epsilon} + 1\right] \\ &\leq \log\left[\frac{(1+F^*) \cdot \mathcal{N} \cdot F^*}{\epsilon} + 1\right] \\ &= O(\log 3 \cdot n + \log(1/\epsilon)).\end{aligned}$$

This result shows that at the end of the online algorithm, the violation level on the rack capacity constraint is upper bounded by $O(\log 3 \cdot n + \log(1/\epsilon))$. Therefore, the proposed PDRA algorithm can achieve the competitive ratio of $[(1-\epsilon), O(\log 3 \cdot n + \log(1/\epsilon))]$. □

Under online scenario, the PDRA algorithm can achieve the approximation factor of $1-\epsilon$ for the objective of total reward, while violating the resource constraints by a factor of $O(\log 3 \cdot n + \log(1/\epsilon))$ at most. Combining with Definition 2, we conclude that PDRA is $[(1-\epsilon), O(\log 3 \cdot n + \log(1/\epsilon))]$ competitive. The constant $\epsilon \in [0,1]$ denotes a trade-off between resource violation and reward. Now, we give a practical example to illustrate the approximation performance of PDRA. We assume that $\epsilon = 0.1$. Consider a large-scale network with $n = 1000$ racks, so that $\log 3 \cdot n + \log(1/\epsilon) \approx 4.4$. It means that the approximation factor of the optimization objective is 0.9, while violating resource constraints by a factor of 4.4.

Algorithm Running Example: The settings of the example are the same as those of NTRS's example (see the end of Section 3.2). Assume that the reward of r_1, r_2 and r_3 are $w(r_1) = 2$, $w(r_2) = 2$ and $w(r_3) = 3$, respectively. Following lines 1-5, we initialize the parameters of PDRA. We calculate that $F^* = 2.5$, $\mathcal{N} = 6$, and set $\epsilon = 0.1$, $\alpha_r = 0$, $\beta_k = 0$, $\delta_k = 0$ and $\theta_k = 0$. We suppose that requests r_1, r_2 and r_3 are sequentially submitted to the system in the online scenario. Thus, we first determine the scheduling scheme for r_1. Following line 8 in Algorithm 2, we know that the candidate rack set $\mathcal{K}_{r_1,s} = \{k_1, k_2\}$. According to SCH described in Algorithm 3, we compute the scheduling scheme set D_{r_1}, which includes three scheduling schemes. The first one is that two servers in rack k_1 are assigned to r_1. The second one is that one server in each of racks k_1 and k_2 are assigned to r_1. The third one is that two servers in rack k_2 are assigned to r_1. These three schemes are represented by $d^1_{r_1}$, $d^2_{r_1}$ and $d^3_{r_1}$, respectively. Following lines 9-10 in Algorithm 2, we calculate the price of each scheduling scheme. Since β_k, δ_k and θ_k are all initialized to 0 at the beginning, the prices of $d^1_{r_1}$, $d^2_{r_1}$ and $d^3_{r_1}$ are all equal to 2. We assume that scheduling scheme $d^2_{r_1}$ is selected. Thus, we assign one server on each of racks k_1 and k_2 to r_1. Similar to the steps above, one server on each of racks k_1 and k_2 is assigned to r_2. Two servers on rack k_1 and one server on rack k_2 are assigned to r_3.

5 Performance Evaluation

In this section, we evaluate the performance of our proposed algorithms through both small-scale testbed implementation and large-scale simulations. Section 5.1 first introduces the performance metrics and methodology. Then, Section 5.2 presents the simulation results. Last, Section 5.3 gives the experiment results. The conclusion of experiment results agrees with that from the simulation results.

5.1 Performance Metrics and Methodology

For the offline scenario, we choose two metrics. The first one is *rack load factor (RLF)*. During the system running, we measure the bandwidth load $l(k)$ of the ToR switch on each rack k, and the rack load factor is defined as: $RLF = \max\{l(k)/B(k), k \in \mathcal{K}\}$, where $B(k)$ is the bandwith capacity of the ToR switch associated with rack k. The second one is *the number of tenants served by a server (NTS)*. The smaller RLF and NTS means better bandwidth load balancing among ToR switches on racks and tenant isolation, respectively. To evaluate how well the proposed algorithm performs, we compare it with the other three benchmarks. The first one is the optimal result, denoted as OPT-LP, which can be derived by optimally solving the linear programming based on Eq. (1). The second one is the fine-grained request scheduling algorithm (FRS), modified from [10]. FRS sorts all individual requests in descending order according to bandwidth requirements, then chooses a server with the maximum bottleneck resource. The last one is a greedy load balancing algorithm, denoted as Greedy, which first sorts all tenant-grained requests in descending order according to bandwidth requirements. Then it chooses a server with the maximum available bottleneck resource for each tenant-grained request.

For the online scenario, we use the following four performance metrics. The first metric is *scheduling time* which is the time for the scheduler to make scheduling decisions. The second metric is *scheduling message consumption*, which is the bandwidth overhead for the scheduler to send scheduling decision messages to servers. These two metrics evaluate the degree of system scalability and the scheduling overhead. The third metric is NTS, which is used to estimate the performance of tenant isolation. Since throughput is important for clouds, we use the system throughput as the reward under the online scenario. Thus, the last metric is

TABLE 4
Simulation Parameter Settings

Parameters	Description
the number of servers in a rack	[25,30]
the bandwidth capacity of a rack	[15Gbps, 20Gbps]
the power capacity of a rack	[15kW, 20kW]
scheduling message cost per request	1KB
data set	Google Cluster

the *system throughput*. To evaluate the performance of our online algorithm, we choose two benchmarks for performance comparison. The first one is the online fine-grained request scheduling algorithm [11], denoted as ONFR. Once an individual request arrives, ONFR first chooses the rack with the maximum available bottleneck resource, then schedules this individual request. The second one is based on the online bin packing algorithm that schedules requests at the granularity of tenant, which is denoted as ONPA and modified from [9]. ONPA traverses racks in sequence until it finds a rack which can handle the incoming request.

5.2 Simulation Evaluation

In this section, we introduce the simulation settings, including the rack set settings and generation methods of individual requests and tenant-grained requests.

5.2.1 Simulation Settings

1) Rack Set: According to our statistics from Google Clusters [43], we set the number of servers that a rack can hold with a random number in [25, 30]. To emulate the heterogeneous environment (e.g., the bandwidth and power capacity of different racks would be different), the bandwidth capacity of each rack is drawn uniformly between 15Gbps and 20Gbps, and the power capacity is generated randomly from 15kW to 20kW.

2) Request Generation: Similar to [10], [11], we use the data traces of Google Clusters [43] to generate the individual request set. Two data sets denoted as (a) and (b) in Google Clusters are adopted in this simulation. The difference between these two data sets is that the individual requests in Data Set (a) require more computation resources than that in Data Set (b). Note that, these data traces only contain the information of CPU cost of individual requests. To be more practical, we randomly generate other required information for our simulations. Specifically, for the individual request set, the bandwidth and power requirement of individual requests are generated through multiplying the CPU demand with a random value in (0.1,1), respectively.

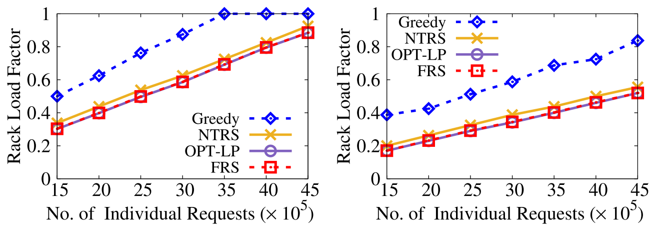


Fig. 2. Rack Load Factor versus No. of Individual Requests *Left plot*: Dataset (a); *right plot*: Dataset (b).

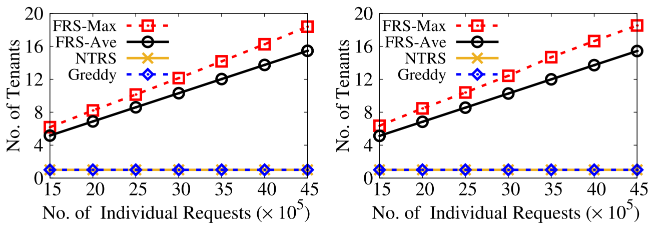


Fig. 3. No. of Tenants Served by a Server versus No. of Individual Request *Left plot*: Dataset (a); *right plot*: Dataset (b).

Moreover, we assume that each tenant-grained request consists of 500 individual requests, and the CPU requirement of each tenant-grained request is the sum of the CPU requirement of these individual requests. With the similar way, we can get the bandwidth and power requirements of each tenant-grained request. According to [29], we set the schedule message consumption for the scheduler to schedule one request to the specified server and 1KB, respectively. For ease of reference, we summarize the main parameters used in simulation in Table 4.

3) Simulation Scenarios: The simulations are performed under two scenarios, that is the offline scenario and online scenario. To evaluate the performance of our proposed algorithms, we test the performance of OPT-LP, FRS, NTRS and Greedy under the offline scenario. Under the online scenario, we compare PDRA with ONFR and ONPA. We evaluate our algorithm using the computer equipped with a core i7-8700k processor and 32GB of RAM.

5.2.2 Simulation Results Under the Offline Scenario

The first set of simulations investigates the rack load factor and NTS by changing the number of individual requests, given 1×10^4 racks. The results are shown in Figs. 2 and 3. In Fig. 2, as the number of individual requests increases, the rack load factor increases for all algorithms. However, the increasing rate of NTRS is much slower than that of Greedy. Since it is easier to achieve bandwidth load balancing for resource allocation by fine-grained request scheduling, the rack load factor of FRS is very close to that of OPT-LP. Our proposed NTRS algorithm can achieve the load balancing performance with a very small gap with OPT-LP and FRS. Specifically, when there are 35×10^5 individual requests in the dataset (a), the gap between NTRS and OPT-LP/FRS is within 3% while decreasing the rack load factor by about 28% compared with Greedy. In Fig. 3, we can see that the average and the maximum number of tenants served by a server increase for FRS, while that of NTRS remains at one, which means that NTRS can achieve better tenant isolation. The reason is that NTRS allocates resources at the granularity of tenant-grained requests with tenant isolation, while others can not.

In the second set of simulations, we observe the rack load factor by changing the number of racks under 30×10^5 individual requests. The results are shown in Fig. 4. As the number of racks increases, the rack load factor decreases for all four algorithms, since there are more resources. However, the rack load factor of NTRS is always smaller than that of Greedy and very close to that of OPT-LP. Specifically, in dataset (a), when the number of racks is 1.3×10^4, NTRS

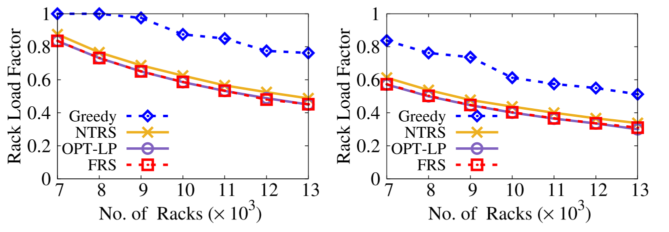


Fig. 4. Rack Load Factor versus No. of Racks *Left plot*: Dataset (a); *right plot*: Dataset (b).

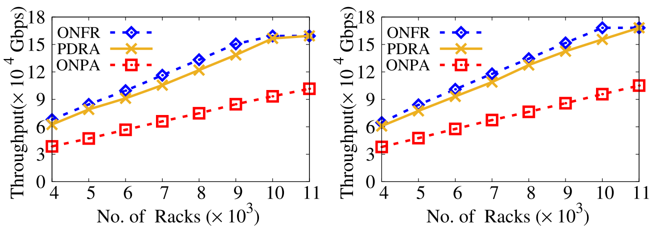


Fig. 7. Throughput versus No. of Racks *Left plot*: Dataset (a); *right plot*: Dataset (b).

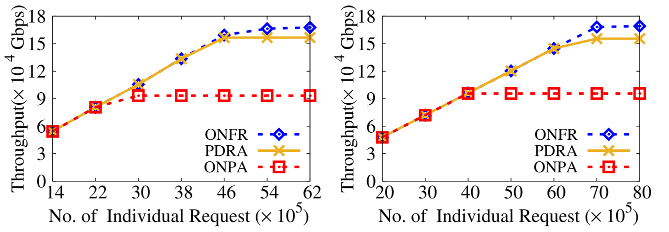


Fig. 5. Throughput versus No. of Individual Requests *Left plot*: Dataset (a); *right plot*: Dataset (b).

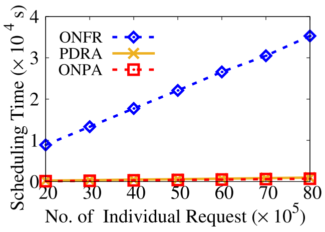


Fig. 8. Scheduling Time versus No. of Individual Requests.

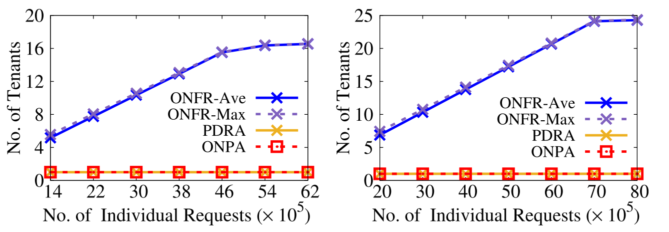


Fig. 6. No. of Tenants Served by a Server versus No. of Individual Requests *Left plot*: Dataset (a); *right plot*: Dataset (b).

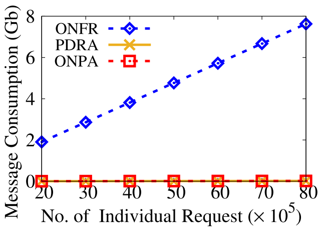


Fig. 9. Scheduling Message Consumption versus No. of Individual Requests.

can decrease the rack load factor by 27.5% compared with Greedy, while the gap between NTRS and OPT-LP is 3%. This shows that NTRS can effectively reduce the rack load factor.

5.2.3 *Simulation Results Under the Online Scenario*

The third set of simulations evaluates the throughput and NTS by changing the number of individual requests given 1×10^4 racks under the online scenario. The results are shown in Figs. 5 and 6. Fig. 5 shows that with the number of individual requests increases, the throughput increases for all three algorithms. The throughput of our proposed algorithm is higher than that of ONPA and slightly lower than that of ONFR. Specifically, by the right plot of Fig. 5, in the case of 50×10^5 individual requests, PDRA can achieve higher throughput about of 40% than that of ONPA while close to that of ONFR. Since fine-grained request scheduling can use fragmented resources, the performance of ONFR is slightly better than PDRA. In Fig. 6, we can observe that the PDRA algorithm can achieve better tenant isolation under the online scenario. With the number of individual requests increases, the average and the maximum number of tenants served by a server increase for the fine-grained request scheduling, while that of our proposed algorithm remains at one.

In the fourth set of simulations, we change the number of racks to observe the throughput. Since the individual requests in dataset (a) require more resources than in dataset (b), we generate 45×10^5 individual requests for dataset (a) and 70×10^5 individual requests for dataset (b). The results are shown in Fig. 7. The throughput of all three algorithms increases with the increasing number of racks. The throughput of PDRA is close to that of ONFR and much higher than that of ONPA. For example, in the dataset (a), when the number of racks is 1×10^4, The PDRA algorithm can improve the system throughput by about 41.3% compared with ONPA and the throughput gap between PDRA and ONFR is within 2%.

The fifth set of simulations shown in Figs. 8 and 9 investigates the scheduling time and message consumption by changing the number of individual requests given 1×10^4 racks. As the number of individual requests increases, the scheduling time and message consumption of the benchmarks are increasing rapidly, while these scheduling overhead of the PDRA algorithm grows slowly. For example, given 8×10^6 individual requests, the scheduling time and message consumption of the FRS algorithm, which is a fine-grained request scheduling method, are $500\times$ more than that of our algorithms. Though the scheduling time and message consumption of the ONPA algorithm, which schedules requests at the tenant granularity, are as low as that of the PDRA algorithm, our algorithm can achieve a higher throughput about of 40% as shown in Fig. 5.

From these simulation results, we can draw the following conclusions. First, compared with the fine-grained request scheduling methods, our proposed algorithms can achieve similar network performance (i.e., bandwidth load factor in

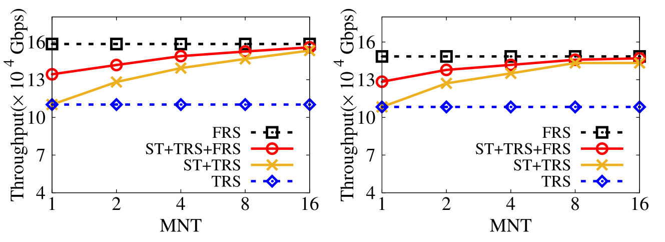


Fig. 10. Throughput versus MNT *Left plot*: Dataset (a); *right plot*: Dataset (b).

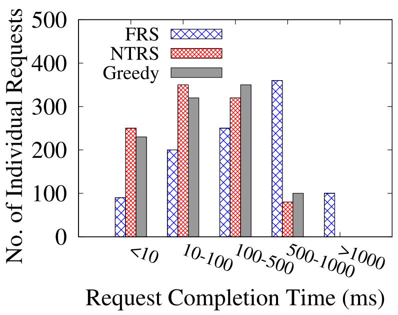


Fig. 11. No. of Individaul Reqeusts versus Request Completion Time under the Offline Scenario.

the offline scenario and throughput in the online scenario) while reducing schedule overhead more than 90% and achieving better tenant isolation. Second, compared with the other tenant-grained request scheduling benchmarks, our proposed algorithms can achieve much better network performance, e.g., our algorithms can reduce the rack load factor by about 28% under the offline scenario, and increase throughput by more than 40% in the online scenario.

5.2.4 *Dealing With Small-Scale Tenants and Combining Tenant-Grained Request Scheduling With Fine-Grained Request Scheduling*

We design a set of simulation to illustrate the resource utilization improvement by the combination of fine-grained request scheduling and tenant-grained request scheduling. Since the simulation results in Figs. 5 and 7 show that the tenant-grained request scheduling has limited impact on resource utilization of large-scale tenants, the following simulations are for small-scale tenants. We compare the performance of four scheduling methods. The first one is the proposed tenant-grained request scheduling algorithm (TRS) in this paper. The second one is denoted as ST+TRS, which aggregates individual requests from multiple small-scale tenants first, and then performs tenant-grained request scheduling. ST+TRS can avoid the resource waste problem of small-scale tenants. The third one is denoted as ST+TRS+FRS, which first performs ST+TRS to schedule tenant-grained requests, and then performs fine-grained request scheduling to schedule individual requests from one underutilized server to another. ST+TRS+FRS combines tenant-grained request scheduling and fine-grained request scheduling, which can further improve resource utilization. The last one is fine-grained request scheduling (FRS), which performs the best for resource utilization. Since the system throughput reflects resource utilization, we choose system throughput as the metric for performance comparison.

In the simulation, we change the maximum number of tenants a server can serve (MNT) and observe its impact on system throughput. This set of simulation can evaluate the trade-off between resource utilization and tenant isolation. The simulation results are shown in Fig. 10. We use the data traces (a) and (b) of Google Clusters [43] to generate 60×10^5 individual requests. Since FRS does not consider tenant isolation and TRS always assigns a server to only one tenant, their throughput does not change with MNT. The system throughput of FR+TRS and ST+TRS+FRS grows with MNT, since the tenant isolation constraint is weakened. When MNT is greater than 8, the throughput gap between ST+TRS+FRS and FRS is within 5%, which means that the combination of fine-grained request scheduling method and tenant-grained request scheduling method can further improve resource utilization. Thus, we can use ST+TRS +FRS to improve resource utilization for small-scale tenants.

5.3 Testbed Evaluation

5.3.1 *Testbed Settings*

To better evaluate the performance of tenant-grained request scheduling, we implement the proposed algorithms on a real testbed, which contains seven servers running Ubuntu 18.04. Among them, six servers are used to start virtual machines (VMs) for processing tenants' requests, and each server is equipped with a 22-core Intel Xeon 6152 processor and 128GB memory. Another server is equipped with a 10-core Intel i9-10900 processor and 64 GB of RAM, and is used as a scheduler that is responsible for allocating VMs to requests through scheduling algorithms. It should be noted that the computing resources allocated for tenant-grained requests in the testbed are VMs rather than servers due to the limited number of servers in the testbed. We set 20 tenants in our experiment, and the individual requests generated by each tenant come from Google Cluster [43]. We use the vnStat tool [44] to monitor and collect scheduling message consumption from the scheduler. To verify the tenant isolation performance, we randomly select a tenant as a malicious one, which can launch denial of service (DoS) attacks [45] with the hping tool [46]. Once other tenants' applications are attacked, their request completion time will be longer. Since every tenant expects that their requests can be processed as soon as possible, the nearly last request completion time of each tenant can be a meaningful metric to evaluate tenants' QoS. Thus, we measure the completion time of all requests and 99% request completion time of each tenant for tenant isolation performance comparison. Similar to simulation, the testbed are conducted under both offline and online scenarios.

5.3.2 *Testbed Results Under the Offline Scenario*

Figs. 11 and 12 show the tenant isolation performance given 1000 individual requests under the offline scenario. Fig. 11 depicts the request completion time when the malicious tenant launches a DoS attack. From the experimental results, we observe that the request completion time of FRS (based on fine-grained request scheduling) is much longer than that of NTRS and Greedy (based on tenant-grained request scheduling). Specifically, the percentages of requests with completion time more than $500ms$ through NTRS, Greedy and FRS are 8%, 10% and 46%, respectively. Fig. 12 displays

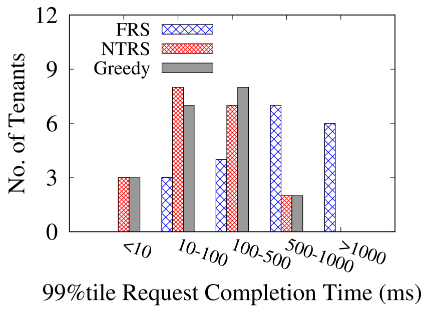


Fig. 12. No. of Tenants versus 99%tile Request Completion Time under the Offline Scenario.

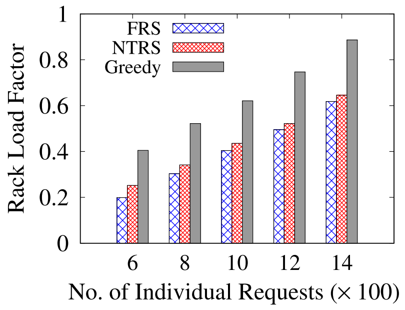


Fig. 13. Rack Load Factor versus No. of Individual Requests under the Offline Scenario.

the 99%tile request completion time of each tenant. The results show that the numbers of tenants with 99%tile request completion time more than $500ms$ through NTRS, Greedy and FRS are 2, 2 and 13, respectively. This illustrates that tenant isolation achieved by tenant-grained request scheduling under the offline scenario can effectively reduce the impact of malicious tenant attacks on other normal tenants.

Fig. 13 exhibits the rack load factor performance by changing the number of individual requests. As the number of individual requests increases, the rack load factor increases for all algorithms and the increasing rate of NTRS is much slower than that of Greedy. Specifically, when there are 1000 individual requests, the rack load factors of NTRS, FRS and Greedy are 0.43, 0.41 and 0.66, respectively. With the help of fine-grained request scheduling, the rack load factor performance of FRS is slightly better than that of NTRS, and the gap between NTRS and FRS is within 5%. Moreover, NTRS can decrease the rack load factor by about 34.8% compared with Greedy.

From the experiment results under the offline scenario, we can draw the following conclusion. First, from Figs. 11 and 12, the offline tenant-grained request scheduling through NTRS can achieve tenant isolation, so the request completion time performance of NTRS is better than that of FRS. Specifically, NTRS can reduce the number of tenants whose 99%tile request completion time exceeds $500ms$ by 55%. Second, Fig. 13 shows that the NTRS algorithm can still achieve similar rack load factor performance to FRS, and reduce rack load factor by 34% compared with Greedy while ensuring tenant isolation. In conclusion, the offline tenant-grained request scheduling algorithm can achieve rack load factor performance similar to fine-grained request scheduling with tenant isolation guaranteeing.

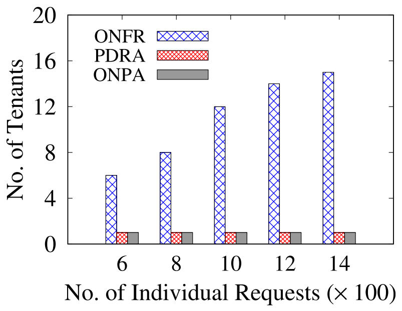


Fig. 14. No. of Tenants Served by a VM versus No. of Individual Requests under the Online Scenario.

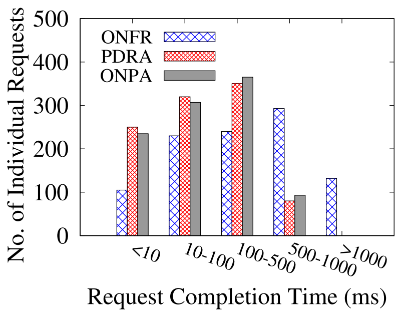


Fig. 15. No. of Individual Reqeusts versus Request Completion Time under the Online Scenario.

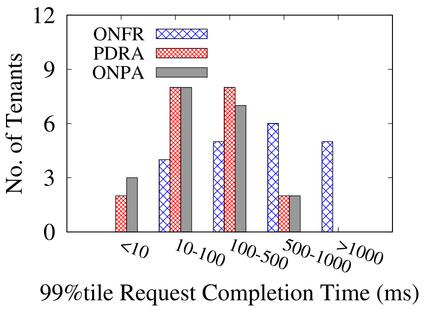


Fig. 16. No. of Tenants versus 99%tile Request Completion Time under the Online Scenario.

5.3.3 Testbed Results Under the Online Scenario

The experiment results under the online scenario are shown in Figs. 14, 15, 16, 17, 18, and 19. Figs. 14, 15, and 16 show the tenant isolation performance. Specifically, Fig. 14 depicts that the number of tenants served by a VM through ONFR (based on fine-grained request scheduling) grows when the number of individual requests increases. On the contrary, PDRA and ONPA (based on tenant-grained request scheduling) can ensure that each VM only serves one tenant, thereby guaranteeing tenant isolation. Fig. 15 exhibits the request completion time results given a malicious tenant. The experiment results show the request completion time of ONFR is much longer than that of PDRA and ONPA. Specifically, the completion time of more than 90% requests through the proposed PDRA algorithm is less than $500ms$, while around 42.5% of requests through ONFR are completed in more than $500ms$. Fig. 16 displays the 99%tile request completion time of tenants. The experiment results show that the numbers of tenants with 99%tile request completion time more

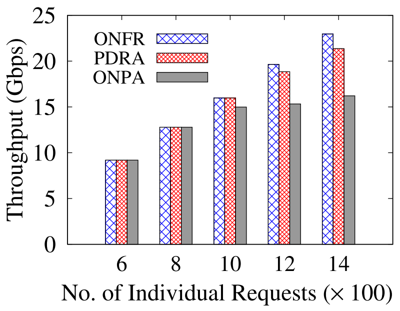


Fig. 17. Throughput versus No. of individual requests under online scenario.

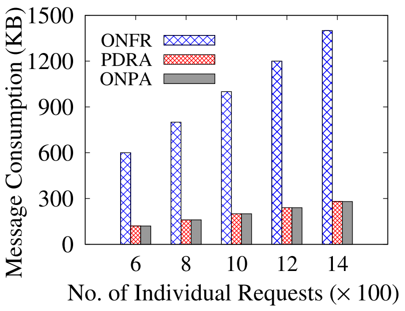


Fig. 19. Scheduling Message Consumption versus No. of Individual Requests under the Online Scenario.

than $500ms$ through PDRA, ONPA and ONFR are 2, 2 and 11, respectively. These results confirm that tenant isolation achieved by tenant-grained request scheduling under the online scenario can effectively reduce the impact of malicious tenants on other normal tenants.

Figs. 17, 18, and 19 give the online scheduling performance results, including throughput, scheduling time and message consumption. Specifically, Fig. 17 displays the throughput performance by changing the number of individual requests. The results show that as the number of individual requests increases, the throughput increases for all benchmarks. The throughput by the proposed PDRA algorithm is close to that of ONFR and higher than that of ONPA. For example, when there are 1400 individual requests, the throughput achieved by ONFR, PDRA and ONPA are 22.96Gbps, 21.37Gbps and 16.21Gbps, respectively. It means that PDRA can increase the throughput by about 30% compared with ONPA. Figs. 18 and 19 exhibit the scheduling time and message consumption by changing the number of individual requests. The results show that when the number of individual requests increases, the scheduling time and message consumption of the fine-grained request scheduling algorithm (i.e., ONFR) grow rapidly and those of tenant-grained request scheduling algorithms (i.e., PDRA and ONPA) increase slowly. For example, given 1400 individual requests, the scheduling time and message consumption of ONFR are $0.844s$ and 1427KB, respectively. Compared with ONFR, the proposed PDRA algorithm can reduce the scheduling time by 70% and the scheduling message consumption by 80.4%. It verifies that the tenant-grained request scheduling method under the online scenario can greatly reduce the scheduling time and message consumption, thereby reducing scheduling overhead and enhancing system scalability.

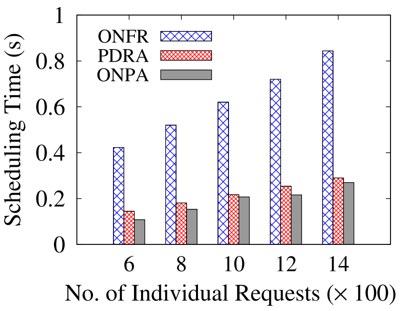


Fig. 18. Scheduling Time Consumption versus No. of Individual Requests under the Online Scenario.

From the experiment results under the online scenario, we can draw the following conclusion. First, from Figs. 14, 15, and 16, the online tenant-grained request scheduling through the proposed PDRA algorithm can achieve tenant isolation, so the request completion time performance of PDRA is better than that of ONFR. For example, it can reduce the number of tenants whose request completion time exceeds $500ms$ by 45% compared with ONFR. Second, Fig. 17 shows that PDRA can improve throughput by 30% compared with ONPA and still maintain a similar throughput performance to ONFR while ensuring tenant isolation. Third, from Figs. 18 and 19, PDRA can reduce the scheduling time and message consumption by 70% and 80.4% compared with ONFR, respectively.

6 Related Works

To provide high-quality services, cloud providers expect to meet requests' requirements (e.g., computing resources, completion time *etc*.) as much as possible through scheduling requests, while ensuring tenants' security. In this section, we summarize the state-of-the-art request scheduling methods and security guarantee solutions in the cloud.

Request scheduling has been extensively studied in recent years to maximize cloud providers' revenue and resource utilization. The comprehensive survey of request scheduling can be found in [47], [48], [49], [50]. Almost all the previous methods are based on fine-grained request scheduling. Mann et al. [10] study the request scheduling problem by jointly optimizing the mapping virtual machines (VMs) to physical machines and the mapping application components to VMs, and they also take into account colocation constraints, hardware affinity relations, sizing aspects and license costs. Atmant et al. [16] design a semi-online request scheduling framework based on a bin packing approach. This approach gathers information on incoming requests during a short time window before deciding on their assignments, and uses a dynamic and real-time allocation algorithm to make scheduling decisions. Wardat et al. [17] try to fulfil requests' requirements and maximize the revenue while reducing the total operational cost through servers consolidation. Li et al. [22] propose WIHAUL, a network-wide airtime resource allocation and scheduling mechanism that works with TDM-based medium access protocols (including 3GPP 5G NR and IEEE 802.11ad), which explicitly guarantees inter-flow max-min fairness in mm-wave backhauls. POCLib [51] presents a set of new solutions that enable efficient random access on hierarchically compressed data, significantly reducing large bandwidth consumption during request scheduling. It should be noted that the tenant-grained request scheduling method reduces bandwidth consumption by reducing the number of scheduling messages, while the compression-based direct processing technique like POCLib reduces bandwidth consumption by compressing data. Although these two methods have different mechanisms to reduce bandwidth consumption, they can be applied in the system at the same time.

However, the above request scheduling methods can not guarantee tenants' security. Malicious tenants can launch attacks such as DoS to paralyze servers and pose a great threat to other normal tenants. To prevent malicious tenants, a series of detection mechanisms have been proposed. Seawall [52] deploys a traffic analysis module on the hypervisor for detecting the UDP traffic or abnormally behaving TCP stack from the malicious tenants. Once malicious traffic is detected, the security module may limit the malicious traffic speed or shut down the malicious VM. Privateeye [53] detects the malicious VM based on the 10-minute flow pattern changes, but it still cannot achieve 100% malicious detection. Ho et al. [54] present a new approach for detecting credential spearphishing attacks in enterprise settings. Their method uses features derived from an analysis of fundamental characteristics of spearphishing attacks, combined with a new non-parametric anomaly scoring technique for ranking alerts.

Although the above malicious tenant detection methods can ensure tenants' security to a certain extent, the damage may have been caused before malicious tenants are successfully detected. Thus, even if they can detect malicious tenants, they cannot avoid the influence of malicious tenants on other normal tenants. Different from these methods, this paper only assigns a server to an enterprise tenant while scheduling requests to achieve tenant isolation, so as to avoid malicious tenants from attacking other normal tenants on the same server, thereby enhancing cloud reliability. Moreover, compared with previous request scheduling methods based on fine-grained request scheduling, this paper proposes the tenant-grained request scheduling method to greatly reduce scheduling overhead.

7 Conclusion

In this paper, we study the tenant-grained request scheduling to achieve high scalability, while considering tenant isolation and resource constraints. We design two efficient algorithms for the offline and online scenarios and formally analyze their approximation performance. Extensive simulation and experiment results show that our algorithms can greatly reduce scheduling overhead and achieve tenant isolation, while maintaining similar performance (i.e., rack load factor and throughput) to the fine-grained scheduling method.

References

[1] A. Fox et al., "Above the clouds: A berkeley view of cloud computing," Dept. Electrical Eng. Comput. Sciences, Univ. California, Berkeley, CA, Tech Rep. UCB/EECS, 2009.

[2] D. Catteddu, "Cloud computing: Benefits, risks and recommendations for information security," in *Proc. Iberic Web Application Secur. Conf.*, 2009, pp. 17–17.

[3] A. Khajeh-Hosseini, I. Sommerville, and I. Sriram, "Research challenges for enterprise cloud computing," 2010, *arXiv:1001.3257*.

[4] M. Armbrust et al., "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[5] T. Dillon, C. Wu, and E. Chang, "Cloud computing: Issues and challenges," in *Proc. 24th IEEE Int. Conf. Adv. Informat. Netw. Appl.*, 2010, pp. 27–33.

[6] "Cloud services global market opportunities and strategies to 2022," 2019. Accessed: Aug. 28, 2022. [Online]. Available: https://www.businesswire.com/news/home/20190411005524/en/528.4-Billion-Cloud-Services-Market-Global-Opportunities

[7] C.-L. Hsu and J. C.-C. Lin, "Factors affecting the adoption of cloud services in enterprises," *Informat. Syst. E-Bus. Manage.*, vol. 14, no. 4, pp. 791–822, 2016.

[8] E. Truyen, D. Van Landuyt, V. Reniers, A. Rafique, B. Lagaisse, and W. Joosen, "Towards a container-based architecture for multi-tenant SaaS applications," in *Proc. 15th Int. Workshop Adaptive Reflective Middleware*, 2016, pp. 1–6.

[9] C. Li and X. Tang, "On fault-tolerant bin packing for online resource allocation," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 4, pp. 817–829, Apr. 2019.

[10] Z. Á. Mann, "Resource optimization across the cloud stack," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 1, pp. 169–182, Jan. 2017.

[11] Y. Bao, Y. Peng, C. Wu, and Z. Li, "Online job scheduling in distributed machine learning clusters," in *Proc. IEEE Conf. Comput. Commun.*, 2018, pp. 495–503.

[12] W. Song, Z. Xiao, Q. Chen, and H. Luo, "Adaptive resource provisioning for the cloud using online bin packing," *IEEE Trans. Comput.*, vol. 63, no. 11, pp. 2647–2660, Nov. 2014.

[13] H. Xu and B. Li, "Joint request mapping and response routing for geo-distributed cloud services," in *Proc. IEEE INFOCOM*, 2013, pp. 854–862.

[14] R. Burra, C. Singh, and J. Kuri, "Service scheduling for bernoulli requests and quadratic cost," in *Proc. IEEE Conf. Comput. Commun.*, 2019, pp. 2584–2592.

[15] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015.

[16] V. Armant, M. De Cauwer, K. N. Brown, and B. O'Sullivan, "Semi-online task assignment policies for workload consolidation in cloud computing systems," *Future Gener. Comput. Syst.*, vol. 82, pp. 89–103, 2018.

[17] M. Wardat, M. Al-Ayyoub, Y. Jararweh, and A. A. Khreishah, "Cloud data centers revenue maximization using server consolidation: Modeling and evaluation," in *Proc. IEEE Conf. Comput. Commun. Workshops*, 2018, pp. 172–177.

[18] S. Mazumdar and M. Pranzo, "Power efficient server consolidation for cloud data center," *Future Gener. Comput. Syst.*, vol. 70, pp. 4–16, 2017.

[19] J. Mate, K. Daudjee, and S. Kamali, "Robust multi-tenant server consolidation in the cloud for data analytics workloads," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 2111–2118.

[20] M. Korupolu and R. Rajaraman, "Robust and probabilistic failure-aware placement," *ACM Trans. Parallel Comput.*, vol. 5, no. 1, pp. 1–30, 2018.

[21] L. Chen, S. Liu, B. Li, and B. Li, "Scheduling jobs across geo-distributed datacenters with max-min fairness," *IEEE Trans. Netw. Sci. Eng.*, vol. 6, no. 3, pp. 488–500, Jul.–Sep. 2019.

[22] R. Li and P. Patras, "Max-min fair resource allocation in millimetre-wave backhauls," *IEEE Trans. Mobile Comput.*, vol. 19, no. 8, pp. 1879–1895, Aug. 2020.

[23] M. Shafiee and J. Ghaderi, "On max-min fairness of completion times for multi-task job scheduling," in *Proc. IFIP Netw. Conf.*, 2020, pp. 100–108.

[24] 2020. [Online]. Available: https://www.alibabacloud.com/

[25] C. Delimitrou and C. Kozyrakis, "Bolt: I know what you did last summer... in the cloud," *ACM SIGARCH Comput. Architecture News*, vol. 45, no. 1, pp. 599–613, 2017.

[26] L. Csikor, C. Rothenberg, D. P. Pezaros, S. Schmid, L. Toka, and G. Rétvári, "Policy injection: A cloud dataplane dos attack," in *Proc. ACM SIGCOMM Conf. Posters Demos*, 2018, pp. 147–149.

[27] A. O. F. Atya, Z. Qian, S. V. Krishnamurthy, T. La Porta, P. McDaniel, and L. M. Marvel, "Catch me if you can: A closer look at malicious co-residency on the cloud," *IEEE/ACM Trans. Netw.*, vol. 27, no. 2, pp. 560–576, 2019.

[28] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of large-scale multi-tenant {GPU} clusters for {DNN} training workloads," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 947–960.

[29] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with borg," in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, pp. 1–17.

[30] N. Bronson, T. Lento, and J. L. Wiener, "Open data challenges at facebook," in *Proc. IEEE 31st Int. Conf. Data Eng.*, 2015, pp. 1516–1519.

[31] H. Sun, P. Stolf, and J.-M. Pierson, "Spatio-temporal thermal-aware scheduling for homogeneous high-performance computing datacenters," *Future Gener. Comput. Syst.*, vol. 71, pp. 157–170, 2017.

[32] G. Zhao, H. Xu, Y. Zhao, C. Qiao, and L. Huang, "Offloading tasks with dependency and service caching in mobile edge computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 11, pp. 2777–2792, Nov. 2021.

[33] R. V. Lopes and D. Menascé, "A taxonomy of job scheduling on distributed computing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 12, pp. 3412–3428, Dec. 2016.

[34] M. Hirono, T. Sato, J. Matsumoto, S. Okamoto, and N. Yamanaka, "HOLST: Architecture design of energy-efficient data center network based on ultra high-speed optical switch," in *Proc. IEEE Int. Symp. Local Metrop. Area Netw.*, 2017, pp. 1–6.

[35] D. Kim, Y. Zhu, C. Kim, J. Lee, and S. Seshan, "Generic external memory for switch data planes," in *Proc. 17th ACM Workshop Hot Top. Netw.*, 2018, pp. 1–7.

[36] L. Phan, B. Hu, and C.-X. Lin, "An evaluation of turbulence and tile models at server rack level for data centers," *Building Environ.*, vol. 155, pp. 421–435, 2019.

[37] J. Hartmanis, "Computers and intractability: A guide to the theory of NP-completeness," *Siam Rev.*, vol. 24, no. 1, 1982, Art. no. 90.

[38] K. Makarychev and Y. Makarychev, "Nonuniform graph partitioning with unrelated weights," in *International Colloquium Automata Languages Programming*. Berlin, Germany: Springer, 2014, pp. 812–822.

[39] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz, "On the effect of forwarding table size on SDN network utilization," in *Proc. IEEE Conf. Comput. Commun.*, 2014, pp. 1734–1742.

[40] M. Mastrolilli and G. Stamoulis, "Bi-criteria and approximation algorithms for restricted matchings," *Theor. Comput. Sci.*, vol. 540, pp. 115–132, 2014.

[41] J. B. Orlin, "Duality in linear programming," [Online]. Available: http://web.mit.edu/15.053/www/AMP-Chapter-04.pdf

[42] H. Wang, H. Xu, H. Huang, M. Chen, and S. Chen, "Robust task offloading in dynamic edge computing," *IEEE Trans. Mobile Comput.*, to be published, doi: 10.1109/TMC.2021.3068748.

[43] "Google cluster-data," 2019. [Online]. Available: http://github.com/google/cluster-data

[44] "vnStat," 2021. [Online]. Available: http://osrg.github.io/ryu/

[45] I. Vaccari, M. Aiello, and E. Cambiaso, "Slowite, a novel denial of service attack affecting MQTT," *Sensors*, vol. 20, no. 10, 2020, Art. no. 2932.

[46] "hping," 2021. [Online]. Available: http://hping.org/

[47] P. Hosseinioun, M. Kheirabadi, S. R. Kamel Tabbakh, and R. Ghaemi, "Atask scheduling approaches in fog computing: A survey," *Trans. Emerg. Telecommun. Technol.*, vol. 33, 2020, Art. no. e3792.

[48] M. Kumar, S. C. Sharma, A. Goel, and S. P. Singh, "A comprehensive survey for scheduling techniques in cloud computing," *J. Netw. Comput. Appl.*, vol. 143, pp. 1–33, 2019.

[49] B. Wang, C. Wang, Y. Song, J. Cao, X. Cui, and L. Zhang, "A survey and taxonomy on workload scheduling and resource provisioning in hybrid clouds," *Cluster Comput.*, vol. 23, no. 4, pp. 2809–2834, 2020.

[50] B. Saha, "Green computing: Current research trends," *Int. J. Comput. Sci. Eng.*, vol. 6, no. 3, pp. 467–469, 2018.

[51] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and X. Du, "POCLib: A high-performance framework for enabling near orthogonal processing on compression," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 2, pp. 459–475, Feb. 2022.

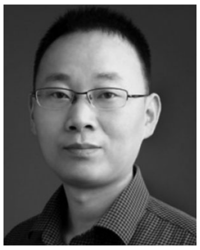
[52] A. Shieh, S. Kandula, A. G. Greenberg, and C. Kim, "Seawall: Performance isolation for cloud datacenter networks," in *Proc. 2nd USENIX Conf. Hot Top. Cloud Comput.*, 2010, pp. 1–7.

[53] B. Arzani et al., "PrivateEye: Scalable and privacy-preserving compromise detection in the cloud," in *Proc. 17th USENIX Symp. Netw. Syst. Des. Implementation*, 2020, pp. 797–815.

[54] G. Ho, A. Sharma, M. Javed, V. Paxson, and D. Wagner, "Detecting credential spearphishing in enterprise settings," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 469–485.

Huaqing Tu (Student Member, IEEE) is currently working toward the PhD degree in computer science with the University of Science and Technology of China. Her main research interests include software-defined networks, network function virtualizatioin, and cloud computing.

Gongming Zhao (Member, IEEE) received the PhD degree in computer software and theory from the University of Science and Technology of China, in 2020. He is currently an associate professor with the University of Science and Technology of China. His current research interests include software-defined networks and cloud computing.



Hongli Xu (Member, IEEE) received the BS degree in computer science and the PhD degree in computer software and theory from the University of Science and Technology of China, China, in 2002 and 2007, respectively. He is currently a professor with the School of Computer Science and Technology, University of Science and Technology of China (USTC). He has published more than 100 articles in famous journals and conferences, including *IEEE/ACM Transactions on Networking*, *IEEE Transactions on Mobile Computing*, *IEEE Transactions on Parallel and Distributed Systems*, International Conference on Computer Communications (INFOCOM), and International Conference on Network Protocols (ICNP). He has held more than 30 patents. His research interests include software defined networks, edge computing, and the Internet of Things. He was awarded the Outstanding Youth Science Foundation of NSFC in 2018. He has won the best paper award or the best paper candidate in several famous conferences.

Xianjin Fang received the PhD degree in computer application technology from Anhui University, in 2010. He is currently a professor and a MS Supervisor with the Anhui University of Science and Technology. His research interests include information security and data mining.