Journal of Grid Computing (2023) 21:46
https://doi.org/10.1007/s10723-023-09679-6

RESEARCH

DAScheduler: Dependency-Aware Scheduling Algorithm for Containerized Dependent Jobs

Abdullah Alelyani · **Amitava Datta** · **Ghulam Mubashar Hassan**

Received: 26 March 2023 / Accepted: 9 July 2023 / Published online: 1 August 2023
© The Author(s) 2023

Abstract Containers have emerged recently as a cloud technology for improving and managing cloud resources. They improve resource sharing by allowing instances to run on top of the host's operating system. Container-based virtualization runs and manages hosted instances via the host kernel. Resource sharing can cause resource contention. In addition, dependent jobs, which may be deployed across multiple hosts, require frequent communication, resulting in a high volume of network traffic and network contention. The majority of existing research focuses on load balancing, with no consideration for the fact that network contention also plays a significant role in container performance. In this research, we propose a Dependency-aware Scheduling algorithm (DAScheduler) that deploys jobs into containers while accounting for both load balancing and job dependencies. The experimental results show that DAScheduler reduces network traffic by more than half and balances the loads. In comparison to one of the existing state-of-the-art techniques, DAScheduler improves overall cloud performance.

A. Alelyani (✉) · A. Datta · G.M. Hassan
Department of Computer Science and Software Engineering, The University of Western Australia, 35 Stirling Highway, Crawley 6009, WA, Australia
e-mail: abdullah.alelyani@research.uwa.edu.au

A. Datta
e-mail: amitava.datta@uwa.edu.au

G.M. Hassan
e-mail: ghulam.hassan@uwa.edu.au

Keywords Containers · Load balance · Scheduling · Network traffic · VMs

1 Introduction

Virtualization technology has been used to improve resource utilization. Today, sharing the physical machine (PM) resources across multiple tenants is a benefit of using virtualization. It allows each tenant to run its own software while sharing the physical infrastructure of the host, such as CPU and memory. Virtualization increases resource availability and decreases service costs, resulting in improved cloud computing performance. [1,2].

Cloud computing involves virtualization, where managing the resources of PMs plays a key role in improving its performance. Virtual machine monitoring (VMM) is the layer that manages virtual machines (VMs) in each PM. It is a resource management component that comes on top of the host's operating system, called *hypervisor*, which enables each VM to create its own operating system. As a result, VMM improves VM instance security.

One of the main objectives of VM is to improve the utilization and provision of resources of the PM. Due to advancements in VM technology, businesses including Google, Amazon, and Microsoft are able to share resources with the public users of the cloud [1,2]. Based on the needs of the clients, cloud resources are provisioned to run the client's job. The use of resources

varies over time. However, once the resources are allocated to the job, the performance of the job is limited by the allocated resources.

Recently, *container* was introduced as a lightweight technology that is less restrictive on the isolation of jobs. It allows the hosted jobs to share the host operating system as well as the resources [3]. Depending on the capacity of the resources, each physical machine can run more containers than virtual machines, as containers do not require their own operating system.

Microservices is a new technology that builds applications from small, loosely dependent services rather than as a single monolithic application [4]. Container technology is widely used in implementing microservices [5], allowing for scaling, isolation, and management of individual microservices, without affecting the entire system. Containers provide a standardized environment for running microservices, which is especially useful for complex systems like Netflix that consist of over 500 distributed microservices across different zones, including edge resources. According to [6], such infrastructures require a network policy to facilitate communication between microservices.

Scheduling containers into PMs is a critical problem that has attracted a lot of attention in the literature [7–9]. Proposing an efficient schedule is a challenging task. Many aspects are required to be considered while developing the schedule, such as allocating resources to containers, balancing the load, cloud performance, energy consumption, etc. However, load balancing and resource contention are the two most important factors influencing the performance of the container [10].

There are two main types of resource contentions: 1) local contention, which occurs when the resource, such as CPU, memory and I/O, are utilized intensively by hosted containers; and 2) network contention, which occurs when massive data is sent or received by containers at the same time across the network.

Load-balancing algorithms are widely used in data centres to enhance container performance and prevent service level agreement violations. However, such approaches often result in resource waste, including inefficient utilization of CPU and memory resources. Moreover, limiting container network traffic or prioritizing load balancing over traffic management may improve quality of service (QoS) but at the cost of increased latency. Therefore, achieving optimal QoS while improving resource efficiency and network latency remains an ongoing challenge in cloud computing research.

Local resource contention has negligible effects on container performance, whereas network contention affects latency and decreases throughput. To address issues related to network traffic and load balancing, we propose DAScheduler, a scheduling algorithm that considers container dependencies and balances the load to reduce traffic overhead. Our research makes the following contributions:

1. Evaluating the limitations of deploying dependent jobs on containers without considering the dependencies between them. In this research, a Round-Robin algorithm is employed to deploy jobs while considering the resource requirement constraint only.
2. Investigating the degradation of the performance caused by network contention.
3. Proposing and evaluating a novel scheduling algorithm for managing the network traffic, balancing the load, and improving the performance of containers.

The rest of the paper is organized as follows: Section 2 discusses the previous research on resource contention on containers. Section 3 defines the problem and its formulation. Section 4 discusses the proposed algorithm. Section 5 introduces the evaluation metrics. Section 6 illustrates the experiments used for evaluating the algorithm, and the analysis of the experiments. Finally, the conclusion is presented in the last section.

2 Related Work

In this section, existing state-of-the-art techniques related to container performance and resource management are discussed, which include resource contentions, resource utilization, and managing the quality of service. The aim is to shed light on research findings that have addressed network costs and performance in the cloud environment. Additionally, we identify the gaps in current research that need to be addressed.

A framework, known as *ConTuner*, was proposed in [7] to resolve the problem of resource contentions. The study aims to automate the procedure of setting the parameters of resource utilization. The proposed framework consists of two parts: 1) a configuration

pool that optimizes the utilization of resources for containers by tuning configuration parameters, and 2) a configuration optimizer that clusters the new jobs into a single historical group known as *new job group*. It then uses the optimized configuration pool to set the resource requirement parameters for this new job group. During the experiment, the framework was used to set the Docker parameters, such as the proportion of CPU, memory, and I/O. ConTuner ensures that all containers that share the same resources utilize the resources comprehensively and efficiently. The experimental results demonstrate that ConTuner can accurately predict resource contention and optimize resource usage for new containers running new arriving jobs. Overall, the framework improves the efficiency and accuracy of container resource utilization.

According to McDaniel et al., [8], managing the quality of service (QoS) and minimizing resource contention assists in achieving the desired performance. The study proposed an algorithm for tackling I/O contention at the cluster and node levels. In addition, an API was developed to give clients control over the I/O of the containers to set the utilization priority and the percentage of I/O utilization. Furthermore, it allows to monitor all containers at the cluster layer and guarantees the implementation of QoS. The results show that the two-tiered approach effectively enhances QoS by monitoring I/O usage in Docker containers. It also shows improvements in the performance of I/O-intensive applications.

A locality-aware scheduling algorithm was proposed to improve container performance by scheduling them on PMs that are close to their dependent containers [9]. The algorithm considers the distances between dependent containers and aims to minimize these distances by grouping containers in the same PMs, resulting in reducing network I/O contentions. However, local I/O contentions may increase due to grouping dependent containers on the same PMs. To balance the two types of contentions, a statistical model was proposed. Experimental results show that the proposed algorithm reduces network latency, leading to improved container performance. However, load balancing between PMs was not considered, which may cause local resource contentions. Overall, the scheduling algorithm improves container performance by considering the locality of data sources and reducing network latency.

In [11], the authors proposed a container traffic analyzer (COTA) to manage the network traffic between containers, which in turn enhances the time spent scaling resources. COTA proposed to enhance the network traffic by managing the following: 1) collects traffic data from host PMs, 3) reports violations to the manager, and 3)balances network traffic using scaler-provided tools. The study proposed an algorithm called Least Traffic Load Balancing (LTLB) to balance network traffic across containers. In addition, COTA has a resource prediction approach, which allows for enhancing the proactive auto-scaling efficiency, making it a promising approach for optimizing resource utilization in container environments.

A multi-objective algorithm was proposed in [12] to enhance container performance. The algorithm aimed to achieve multiple objectives, such as increasing the utilization of CPU and memory, and reducing network traffic. The proposed algorithm involved grouping similar jobs together in containers and finding the optimal physical machine (PM) to provision resources for new containers. The algorithm performs similarly to the Docker Swarm strategies. The results showed that clustering similar containers in a group improved the scheduling process. In addition, the experimental results show that the algorithm outperforms three well-known algorithms namely Spread, Binpack, and Random in terms of resource utilization and load balancing. However, the time complexity of the algorithm is significantly high compared to other algorithms. Finally, the proposed algorithm has the ability to consider multiple objectives while scheduling containers.

A multi-criteria approach that uses swarm techniques was proposed in [13]. The spread and bin packing algorithms were used, to select the PM that hosts containers. Three inputs were considered during the deployment of containers to PMs: 1) the number of containers in each PM; 2) the number of available CPUs; and 3) the amount of available memory. A multi-criteria decision-making algorithm, namely, order of prioritisation by similarity to ideal solution (TOPSIS) [13] was used, which selected the best PMs to run the containers. TOPSIS aims to optimize resource allocation and better load balancing. The experimental results indicate that this technique has the potential to outperform existing scheduling strategies in some scenarios.

Microservice architecture is a complex distributed system that heavily relies on communication between

microservices. Recently, researchers have focused on optimizing resource utilization and reducing network traffic for microservices in distributed systems. Three recent studies [14–16] share the common goal of improving microservice deployment and scheduling by taking into account different factors that affect resource utilization and network traffic. The Microservice-Oriented Topology-Aware Scheduling Framework (MOTAS) [14] partitions the microservices graph based on dependencies and eliminates nodes that violate resource balance provisions. In [15], the authors use machine learning models to optimize network utilization between nodes in the edge network and the cloud platform by predicting upcoming network usage. Lastly, [16] proposes a modification to Kubernetes to consider network requirements between nodes and application topology, taking into account run-time resource utilization. Overall, these studies contribute to the optimization of microservice architecture and provide insights into improving resource utilization and network traffic in distributed systems.

A comparison in Table 1 is presented which illustrates the features and differences between the techniques discussed in this section.

The cost of network traffic, including latency along with the impact of load balancing on container performance, has not been adequately addressed in the existing research. Furthermore, the topology of the data centre, which plays a significant role in scheduling dependent containers, has been lacking consideration in current studies. We aim to address these gaps by proposing an algorithm that minimizes network I/O contentions and balances load in modern data centres.

3 Problem Formulation

Definitions and assumptions are introduced before proceeding with the problem formulation.

1. **Job** refers to the workload that the user submits to the cloud for execution.
2. We refer to the small and isolated environment that runs the job as **container**. If two containers are allocated to two dependent jobs, the containers are dependent, and they need frequent communications either locally or over the network.
3. A data centre is a collection of heterogeneous PMs that communicate via a network. The data centre is also physically distributed across the world in various **zones**. Each zone has a limited number of PMs that can be expanded. We assume that zones are connected via the internet or a private network.

In addition to the terms listed above, there are other terms that are defined in Table 2.

There are three categories of jobs in terms of resource utilization: storage-intensive, I/O-intensive, and computation-intensive. We denote arriving jobs as a set of jobs J=$\{j_{1,\tau_1}, j_{2,\tau_2}, ...j_{n,\tau_i}\}$, where n represents different jobs and $\tau_k, 1 \leq k \leq i$ represents the arrival time of job k. We assume that each job $j \in J$ requires a certain amount of resources, which are known at the submission stage as part of a service level agreement (SLA). The resources that are required by a job are 1) CPU cores; 2) memory; and 3) I/O, and we denote them as the set R= $\{R_{CPU}, R_{mem}, R_{io}\}$. We also denote j_i^R as the requirements of resources R by job i. The cloud offers a set of cells denoted as C =$\{c_1, c_2, ...c_m\}$ that are accommodated by various PMs. Each cell $c_i \in C$ consists of the heterogeneous capacity of resources denoted as c^r=$\{c^{CPU}, c^{mem}, c^{io}\}$. We assume that each cell is a container. We assume that each container can host a limited number of jobs and that these jobs utilize the resources of the host up to a maximum level, as defined by an upper threshold (UTS). Specifically, the utilization of the resources by the container denoted as UR, should not exceed the UTS. The scheduling process ensures that the requirements of each job are met and that the maximum utilization of each container is below the UTS. The following equation expresses this constraint:

$$
\begin{aligned}
& c_m^r >= j_i^R : \ \forall \{t\} \ UR(c_m) <= UTS \\
subject\ to: & \\
& c_m^{CPU} >= j_i^{CPU} \\
& c_m^{mem} >= j_i^{mem} \\
& c_m^{io} >= j_i^{io}
\end{aligned}
\tag{1}
$$

Scheduling jobs to containers optimally is an NP-Complete problem with a high time complexity [17]. For instance, if we have to schedule s jobs into d containers, the time complexity can be as high as $O(s^d)$. Therefore, the best practice to minimize the complexity is to divide the containers into groups of zones Z. Containers should have similar resources across zones and are represented as Z^r=$\{Z^{CPU}, Z^{mem}, Z^{io}\}$.

Table 1 Comparison of techniques in the literature on container resource management and performance

Study	Problem Addressed	Proposed Solution	Parameters considered			
			Resource utilization contentions	CPU contention	Memory contention	I/O contention
Cai et al.(2019) [7]	Resource contention	ConTuner framework: automatic parameter setting for resource utilization	✓	✓	✓	✓
McDaniel et al. (2015) [8]	Quality of service (QoS) management	Algorithm for managing I/O contention				✓
Zhao et al.(2020) [9]	Container performance	Locality-aware scheduling algorithm	✓			✓
Kim et al. (2017) [11]	Resource utilization	Proactive scaling based on predicted network traffic	✓	✓	✓	✓
Liu et al. (2018) [12]	Container performance	Multi-objective algorithm for container scheduling	✓			✓
Menouer & Darmon (2019) [13]	Resource allocation	Multi-criteria approach for container scheduling	✓	✓	✓	
Li et al. (2023), Bao et al. (2023) and Marchese & Tomarchio (2022) [14–16]	Communication between Microservices	MOTAS, Machine learning Algorithm and Kubernetes plugin in and modifications	✓			✓

Table 2 Summary of definitions and notations

Name	abbreviation	definitions
Physical machine	PM	Physical computer system or a server that is composed of hardware components and operating system
Service level agreement	SLA	Contract between a service provider and a user that outlines the level of service that the provider guarantees to deliver.
Upper threshold	UTS	Value representing the maximum level of utilization of a particular resource.
UR	UR	Utilization of resources by a container
Current utilization of resources	U^r	Percentage of a resource that is currently being used.
Current utilization of CPU	U^{CPU}	Percentage of CPU that is currently being used.
Current utilization of memory	U^{mem}	Percentage of memory that is currently being used.
Current utilization of I/O	U^{io}	Percentage of I/O that is currently being used.

Furthermore, the zones are spread across different geographical areas. During scheduling, a job is deployed to the zone that meets its requirements of resources. For instance, if the zone meets the minimum resource requirements for running a job and the zone has an unoccupied container C_m, job j_n will be assigned to it. The constraint is explained by the following equation:

$$\begin{aligned} & Z_k - U^r >= j_i^R \\ subject\ to: & \\ & Z_k^{CPU} - U^{CPU} >= j_i^{CPU} \\ & Z_k^{mem} - U^{mem} >= j_i^{mem} \\ & Z_k^{io} - U^{io} >= j_i^{io} \end{aligned} \tag{2}$$

where U^r is the current utilization of the resources. In addition, U^{CPU}, U^{mem}, and U^{io} are the current utilization of CPU, memory and I/O respectively.

When a user submits one or more applications to the cloud for execution, each application is treated as an individual task. The job has at least one task, job j is defined as a collection of tasks, where $j = \{t_1, t_2, ... t_w\}$. However, there are two types of jobs: dependent and independent. In contrast to independent jobs, dependent jobs rely on other jobs or services to be performed. If dependent jobs are distributed across multiple containers that are located in different zones, they are required to use local or network I/O. The dependencies may not be known at the time of the arrival of a job. During the job scheduling process, only the CPU and memory requirements are known. It is also possible that dependent jobs will be distributed across multiple zones. As a result, the network will be heavily used, resulting in congestion, packet loss, and latency.

Congestion of network traffic also results in high communication latency among dependent jobs and system failure [18,19]. Redundant paths guarantee a reduction in the risk of the above issues. Therefore, we assume that the infrastructure of the network in each zone is multi-path. Furthermore, we assume that each zone is contained within PMs that have a similar capacity of containers as shown in Fig. 1. The term network traffic refers to the stream of packets travelling through a network. However, the latency that is caused by the congestion of the traffic is denoted as the cost of traffic.

Platforms such as Docker use a bridge network to isolate containers [20]. The bridge interface is called docker0 and it routes the packets between containers [21]. Edge computing, in contrast, employs containers to run microservices in resources close to the client. However, because of the limited resources in the node of the edge, containers could be distributed across multiple edges.

A platform such as Kubernetes is used to distribute containers into PMs. Also, it is used to map *pods* to available *nodes*. *A node* could be a PM that is located on either the edge or on the cloud [4]. In addition, Kubernetes supports inter-communication between microservices [6] which is the type of communication between microservices in different PMs.

In summary, our research focuses on the use of containers to run microservices. Therefore, a communication policy is needed for these microservices to interact with each other, whether through interfaces if they are located in the same PM or through the network if

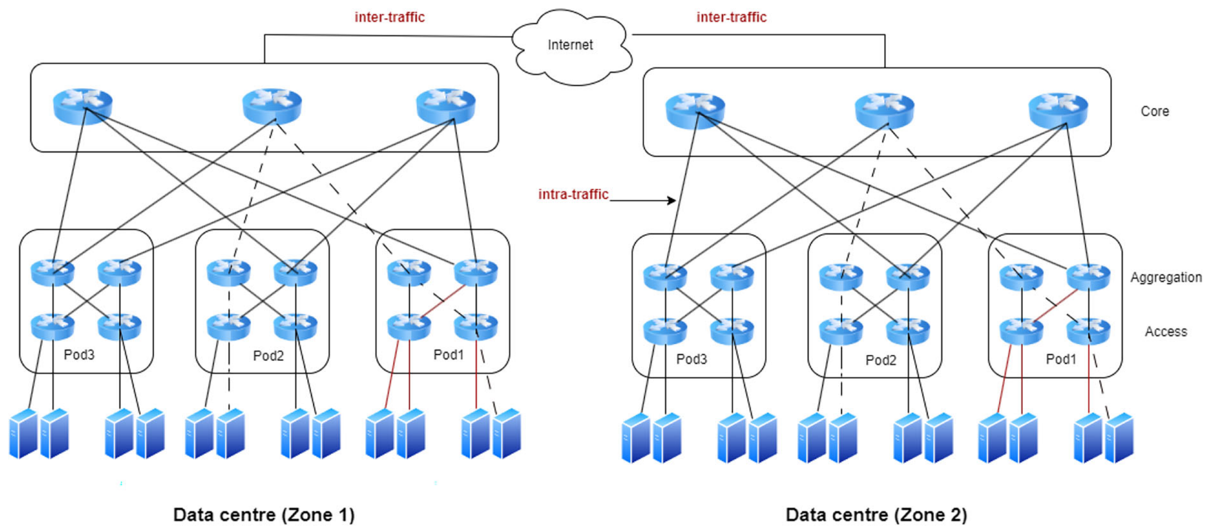

Fig. 1 Zones 1 and 2 represent distinct geographical regions within a single data centre. To handle traffic, a multi-path topology is proposed, in which the internet is used to connect zones within a single data centre

they are distributed across multiple zones. According to [22], network traffic in the data centre is classified as:

1. Inter-traffic: the traffic between the zones of the data centre that are located in different geographical locations.
2. Intra-traffic: the traffic between PMs within a single data centre.

A cloud network is represented as a graph of vertices and edges as $g = \{V, E\}$. The vertices represent the zones, while the edges represent the network. The cost of utilizing the network is calculated as 1) the number of edges in the path between dependent jobs; 2) the cost of each edge; and 3) the total time of using the path.

$$\text{Network cost(NC)} = \sum_{i=1}^{L} \sum_{j=1}^{L} D_{ij} * T_{ij} * M_{ij}$$
$$where: \tag{3}$$
$$D_{ij} = \begin{cases} 1 \; if \;\; App \; i \; and \; j \; dependent \\ 0 \; otherwise \end{cases}$$

where D denotes the dependency matrix between containers that run jobs. Furthermore, T_{ij} represents the cost of edges connecting dependent containers running jobs i and j. We assume that only the shortest paths between dependent containers are taken into account. M_{ij} is the number of paths between the container i and container j. Finally, the total number of containers is denoted by L.

It should be noted that we refer to containers as jobs to run. In addition, we refer to migrating jobs as the process of migrating the containers. The state of the containers (memory state) is migrated from the source host to the destination host during the migration process.

Additionally, we take into account the cost of migrating containers from one PM to another during the migration process. The size of the moved containers may make migration more expensive than sending or receiving packets through the network. However, during a particular time frame, the total cost of network traffic between dependent containers located in different zones could be higher than the cost of container migration.

4 The Algorithm Design and Policy

The first part of this section describes DAScheduler and its architecture. The policy of the algorithm is described in the second part of this section.

4.1 Proposed algorithm

Two factors are considered to have a significant impact on container performance: 1) workload balance, and 2)

the intensive use of the network between containers that are located in different zones. DAScheduler, presented in Fig. 2 consists of three main steps aiming to enhance the overall performance of the containers. The steps are 1) initial scheduling, 2) monitoring system, and 3) making a decision. These three steps aim to reduce the cost of using the network as well as balancing the load.

During the initial scheduling step, DAScheduler distributes newly arriving jobs into containers using a round-robin algorithm (RR). The initial scheduling expedites the process of accommodating incoming jobs.

In the second step, DAScheduler monitors container resource utilization, including network usage and container performance. This stage of our algorithm incorporates benchmarking tools such as Linpack [23], Y-Cruncher [24], NBENCH [25], STREAM [26], Netperf [27], and the Linux/Unix dd command [28–31]. The algorithm then calculates the total cost associated with network usage, load balancing, and container performance, which aids the decision-making process in the subsequent step.

The last step is to group dependent containers in the same zones. As mentioned in Section 3, there are two types of edges: 1) the edges between dependent containers in the same zone that carry intra-traffic; and 2) the edges between dependent containers in different zones that carry inter-traffic. In this study, the NC mentioned in (3) for the second type is considered costly, while the first is considered negligible. By treating this situation as a Max-Cut problem, we intend to partition the graph into subsets in order to minimize the inter-traffic cost between zones. The partitioning is then repeated until the minimum cost is reached.

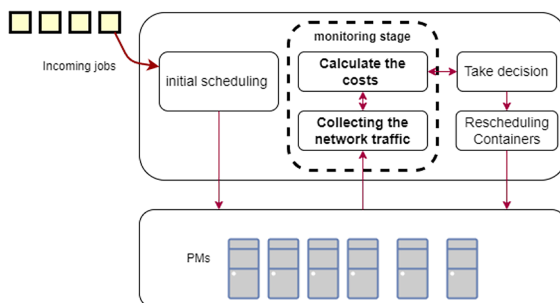

Fig. 2 DAScheduler contains three stages: initial scheduling, monitoring and decision-making

Finally, independent containers located in overloaded PMs are rescheduled to underloaded PMs. This step contributes to improving the load balance of the containers, which improves overall performance.

4.2 Scheduling Policy

Firstly, when new jobs arrive at containers, DAScheduler assigns them to a list L_a. The jobs are then sorted based on their arrival time. It is assumed that containers of similar capacities are assigned to the same zone. Secondly, the zones are assigned to lists $\{L_1, L_2, ...L_p\}$ based on their capacities. Then, DAScheduler sorts the lists of zones based on resource capacity. The RR algorithm is then used to distribute the jobs into zones. Throughout the rest of this study, we use the term "containers" to refer to the job placeholders that run and utilize resources. At this stage, only one constraint is considered which is to meet the resource requirements of each job. We believe that DAScheduler will distribute containers into zones equally. This step boosts the performance of the container in terms of response time, availability and flexibility. Nevertheless, we expect that the latency will be increased due to the intensive use of the network by dependent containers.

The next step focuses on optimizing network costs by grouping dependent containers within the same zone. It is a well-known "max-cut" problem. To tackle this problem, we utilize a modified local search algorithm (MLSA) that comprises the following steps:

1. Calculating the number of external dependencies, which are the dependencies between containers in different zones.
2. Assign dependent containers into list L_{app}.
3. Choose a container from L_{app}, then find its neighbours based on the highest network traffic cost and add them to L_{neg}.
4. Sort L_{neg} list by decreasing traffic cost, and then assign containers from L_{app} to the zone of the first neighbouring container on L_{neg}.
5. If both containers are in different zones, group them in the zone with the most available resources. Otherwise, select the next container from L_{neg}.
6. Steps 3, 4 and 5 are repeated until the last container in L_{app} is reached.

The decision to redeploy independent containers from overloaded zones to underloaded zones is

considered a step towards balancing the load of the containers. Algorithm 1 illustrates our algorithm policy and steps.

Algorithm 1 Proposed Algorithm.

1: $L_a \leftarrow j$ where $j \in J_i$;
2: $z \leftarrow c$ where $c \in C_c$;
3: $L_p \leftarrow Z_k$;
4: $SORT(L_a, t_{arr})$ in ascending order ; ▷ t_{arr}- job arrival time
5: $SORT(L_p, z_{capc})$ in ascending order; ▷ z_{capc}- zone capacity
6: RR(L_p ,L_a);
7: **for each** c_m **do**
8: $dep \leftarrow \sum_1^L D_l$;
9: $L_{neg} \leftarrow c_{m,d}$;▷ where $c_{m,d}$ is dependent container for c_m
10: Sort (L_{neg},dep) in decreasing order;
11: **while** L_{neg} != ∅ **do**
12: $d_c \leftarrow L_{neg}$;
13: **if** $Z_c \neq Z_d$ and $dep_c \neq 0$ **then**
14: Group c_m and d_c in one zone;
15: Remove d_c from L_{neg};
16: $dep \leftarrow dep - dep_c$;
17: **end if**
18: **end while**
19: **end for**
20: **for each** overloaded zone z_{overL} **do**
21: **for each** $c_n \notin L_{neg}$ and $c_n \in z_{overL}$ **do**
22: **if** z_{underL} **then**
23: $z_{underL} \leftarrow c_n$;
24: **end if**
25: **end for**
26: **end for**

5 Performance Evaluation

In this section, the metrics that are used to evaluate DAScheduler are discussed. These metrics measure load balancing and network traffic cost. Moreover, the objective function of our research is also introduced in this section.

5.1 Load Balancing Metric

Balancing the load of containers plays a key role in reducing the latency [32]. Therefore, managing and monitoring load balancing is considered by DAScheduler.

Zhao et al. used the adjusted coefficient of variation (CV) to measure the load balancing for each zone [9]. We aim to use the same method. However, as the number of running containers in each zone represents a different sample of distribution, we expect that CV would produce a different result each time the load balance changes, and will be zero when all zones have the same number of deployed containers. Therefore, we aim to minimize the CV to achieve the highest load balancing. The CV is given as:

$$\min(CV) = \frac{\sigma}{\mu}$$

$$where,$$

$$\sigma = \sqrt{\frac{1}{\|Z\|}\sum_{i=0}^{n}(A_{z,n} - \mu)^2} \quad (4)$$

$$\mu = \frac{\sum_{i=0}^{n} A_i}{\|Z\|}$$

In the given equation, $A_{z,n}$ represents the container number in a particular zone, where Z represents the zone and n represents the container number within that zone.

We observe that CV may give the same result for different samples of distributions and its worst load balance occurs when $CV_{worst} = 1/\sqrt{\|Z\|}$. Therefore, the load balance for each zone is divided by the worst case, and the proportion of the outcome will indicate the proportion of the load balancing for all zones, as shown in (5):

$$\text{LoadBalance(LB)} = \frac{100}{\sqrt{\|Z\|}}(1 - CV) \quad (5)$$

5.2 Network Traffic Cost Metric

High network traffic affects the latency and throughput. Therefore, improving network use should improve latency and network throughput. Guo et al. [33] claimed that for a single day, the maximum total latency in a data centre is found to be approximately 1397.63 milliseconds (ms). It was also mentioned that the maximum latency between containers located in different zones for the data centre was 1.34 ms, whereas 0.268 ms for containers within the zone. Furthermore, the latency in Amazon EC2 in Singapore and North California was 385 ms and 34 ms respectively [9,34].

Based on these observations, the values of 10, 100, and 1000 were chosen as reasonable approximations of network latency in data centres for measuring intra-traffic, inter-traffic, and container migration costs, respectively. While these values are not tied to specific

physical units of measurement, they can be interpreted as cost units or network traffic units for the purpose of clarity in the context of the network architecture being analyzed.

DAScheduler takes into account the type of edges in the graph and therefore, (3) becomes:

$$\text{Network cost(NC)} = \sum_{i=1}^{L}\sum_{j=1}^{L}(D_{ij} * C_{ty}) * M_{ij}$$

$$where: \tag{6}$$

$$D_{ij} = \begin{cases} 1 \; if \; App \; i \; and \; j \; dependent \\ 0 \; otherwise \end{cases}$$

where C_{ty} is the cost of using edge based on traffic type: inter, intra or migrating container traffic, resulting C_{ty} to be 10, 100, or 1000 respectively.

5.3 The Container Performance Metric

Load balance and network traffic are the two factors that influence container performance. A direct correlation between load balancing and cloud container performance is reported in [35]. Therefore, the best performance for cloud containers is obtained when all resources of the zones are utilized at the same level. In our research, we assume that the best preference is achieved when all zones have the same number of containers.

On the other hand, network traffic and container performance are inversely related. Therefore, it is ideal for all the containers to be either independent or dependent containers grouped in the same zones. To optimize performance, both load balancing and network traffic must be minimized.

$$\text{Performnce} = \frac{\alpha + \beta}{NC + CV} \tag{7}$$

where α is the factor of traffic and β is the factor of load balancing.

5.4 The proposed objective function

The proposed objective function has two parts: 1) to reduce the cost of network use, and 2) to balance the load. The network traffic can be minimized by grouping dependent containers into the same zones or minimizing the number of edges between zones, denoted as D. Similarly minimizing CV for all zones will result in maximizing load balancing. Thus, the objective function is given as:

$$\begin{aligned} &arg\ min \sum_{i}^{k} D_{i,k} + \sum_{1}^{z} CV \\ &subject\ to:\ Z(i) \neq Z(k) \end{aligned} \tag{8}$$

where Z(i) and Z(k) are the different zones where dependent containers i and k are located respectively.

6 Experimental Results

In this section, the experimental setup is explained, followed by the evaluation of the performance of our proposed algorithm.

6.1 Experimental Setup

DAScheduler was implemented by simulating the cloud container deployment system. The GO programming language was used to develop the simulation and was run on the Mac operating system with a hardware capacity of 2 GHz on an Intel Core i7 CPU and 4 GB of RAM.

The simulation is divided into four parts that handle scheduling jobs into containers. These parts are:

1. Initializing the scheduler, which configures the containers and zones.
2. Employing initial scheduling by applying the Round-Robin algorithm.
3. Monitoring process involves calculating the network utilization using a metric described in Section 5.2 and load balancing using a metric described in Section 5.1. This process is repeated over a period of time t to assess the efficacy of the algorithm. In addition, the metrics mentioned in Section 4.1 are used for monitoring other resource utilization, such as CPU and memory.
4. Improving the cloud system by migrating containers.

6.2 The Dataset and Experimental Results

DAScheduler was evaluated using two test cases with details of the datasets as shown in Table 3. In comparison to the first test case, the second test case has a significantly higher number of dependencies between jobs. The rest of this section is divided into subsections based on the two test cases.

6.2.1 First Test Case

When a stream of jobs arrived at the cloud, requesting to be deployed into containers, DAScheduler assigned them in a queue. Then the RR algorithm was used to deploy them into the zones. The jobs were equally divided into four zones based on their requirements. Thus, each zone hosted ten jobs. Table 4 shows the names of jobs and their hosted zone.

Dependencies between containers that hosted dependent jobs, were not considered at this stage. For instance, the dependency matrix in Fig. 3 shows that there was dependency between jobs A0 and A12 and the jobs were deployed in different zones, as shown in Table 4. However, load balancing was 100%, which means that all zones have the same number of running containers.

To model the real-world utilization of the network, we assume a cumulative growth of network traffic cost over time. This means that network utilization increases cumulatively as a result of dependent containers communicating. We consider a rate r to represent the growth of network traffic over time t. In our experiment, we capture and analyze the network traffic costs at the beginning and end of the experiment to observe the overall utilization of the network. The cumulative network traffic at time t is given by:

$$T(t) = T0 + rt \tag{9}$$

Table 3 Details of two test cases including the number of jobs, zones and dependencies between the jobs for each test case

Test Case Number	jobs	Zones	Dependencies
1	40	4	22
2	200	10	423

where $T0$ is the initial network traffic cost at the beginning of the experiment.

The initial total cost of network traffic usage by all containers was 3.41% (Table 5). However, due to the cumulative utilization of the network by dependent containers, the cost of network usage continued to increase. By the end of the experiment, the total cost of using the network reached 34.10%. Fig. 4 presents a gradual increase in the network traffic by the dependent containers during the experiment. Migrating containers to group them in the same zone reduced the intra-traffic cost. However, during the migration process, the network traffic was increased.

The total number of dependencies between the containers was initially 22 Table 5. DAScheduler migrates dependent containers into small groups in each iteration. Each group consists of dependent containers, which accounts for 20% of the total number of containers that need to be migrated.

DAScheduler gradually reduced the dependencies by migrating 4, followed by 4,4, and 2 containers to the zones of dependent containers in each iteration. As a result, DAScheduler reduced the total proportion of the cost of using the network from 34.10% to 12.01% (Fig. 5a, b). Additionally, DAScheduler reduced the risk of increasing the cost of using the network by migrating containers in small groups through the network.

To balance the load of all zones, DAScheduler migrated independent containers from overloaded zones into underloaded zones (Figs. 5c). The first migration of containers increased the proportion of load balancing to 93.87% while in the second migration, the algorithm migrated three containers from the overloaded zones to the underloaded zones, which improved the load balancing for all zones to 100%.

The results of our experiment, aimed at reducing dependencies between dependent containers, are presented in Fig. 6. Figure 6 shows the total number of migrated containers at each time instance, as well as the network traffic and load balancing percentages for all zones. The results indicate a significant reduction in network traffic after the migration of the last two containers. Specifically, DAScheduler was able to reduce network traffic by more than half, effectively lowering network contention. As a result of migrating dependent containers, the load balancing percentage was reduced to 94%, but DAScheduler was able to recover and balance the load completely.

Table 4 The initial distribution of jobs across the zones for the first test case

Zone Name	The job Scheduled into Zone								
zone0	A2	A3	A7	A8	A10	A11	A12	A13	A14	A16
zone1	A0	A1	A5	A17	A19	A20	A21	A25	A26	A27
zone2	A4	A23	A28	A29	A30	A31	A32	A33	A34	A35
zone3	A6	A9	A15	A18	A22	A24	A36	A37	A38	A39

Overall, our experimental results show that DAScheduler reduces the total number of dependencies significantly, which contributes to a reduction in the intra-traffic cost, and improvement in load balancing.

6.2.2 Second Test Case

In this test case, a large number of dependencies between containers were introduced. 200 containers were distributed across ten zones. In the first step, DAScheduler schedules the containers into 10 zones evenly by using the RR algorithm. This resulted in 33 dependencies between containers within the same zone and 281 dependencies between containers in different zones. Due to the dependencies between containers located in different zones, the proportion of the cost of using the network reached 37% as shown in Fig. 7.

DAScheduler migrates dependent containers into small groups in each iteration. Each group consists of dependent containers, which accounts for 20% of the total number of containers that need to be migrated. The total amount of network traffic generated by interdependence between containers decreased by 69.5%. A total of 278 dependent containers were grouped in the same zones. DAScheduler reduced the network traffic on the cost of load balancing which ended up at 86%.

The experimental results show a drop in the utilization of the network from 37.8 to 5 % as shown in Fig. 7. Some dependent containers were not grouped in the same zones and the network utilization increased over time slightly, therefore the network utilization increased to 26% of the total cost of using the network. Fig. 7 shows the experimental results for load balancing and network traffic costs using DAScheduler. We observed that load balancing decreased after grouping dependent containers, but DAScheduler addressed this by migrating independent containers from overloaded zones to underloaded zones. As a result, load balancing steadily improved, with a 7.24.% increase observed when 24 containers were migrated (from 86.01% to 93.25%). The load was further balanced by migrating 13 containers to underutilized zones.

Overall, DAScheduler monitors load balancing and network traffic data in real-time and reports it to the decision-making component. The collection of this data enables DAScheduler to improve decision-making during job scheduling. The experimental results in Fig. 8 show that DAScheduler reduces the network traffic significantly and maintains load balancing.

6.3 The Real-Time Scheduling Experiment

In our research, we also tested our algorithm in a real-time scenario to examine its ability to deploy containers as they arrive. During the test cases, containers that arrive with dependencies among distributed services or among themselves were considered. The number of containers for both test cases was set to 20. In the first test case, each container had at most one dependency, whereas in the second test case, each container had at least one dependency.

A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14 A15 A16 ....

A0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 ....
A1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ....
A2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 ....
A3 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ....
A4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ....

Fig. 3 Job Dependency Matrix shows the interaction between jobs where $D_{i,j} = 1$ means there is interaction between job $_i$ and job $_j$

Table 5 The preliminary scheduling results, which show the network traffic cost and job dependencies after they have been scheduled across zones

Description	Value
Number of Dependencies Between job in Different zones	15
Number of Dependencies Between job in the Same zones	7
Number of edges Between jobs in The Same Zone	82 edges
Number of edges Between jobs in different Zones	30 edges
Total proportion Traffic Used	3.410
Total Proportion traffic Used 10X	34.10
Coefficient Variation	0
Load balancing	100%

The first test case had 16 dependencies between containers that were scheduled in different zones, whereas the second test case had 302 dependencies between containers that were scheduled in different zones. Due to the high density of dependencies between containers, the traffic cost increased to 24.4% and to 27.44% for the first and second test cases, respectively (Figs. 9 and 10). The effects of DAScheduler on results are shown in both figures after the vertical black line. It is observed that in the first test case, the network traffic doubled due to a smaller number of new jobs being scheduled in a single PM, leading to higher network traffic. On the other hand, in the second test case, a high number of new jobs were scheduled across different PMs in

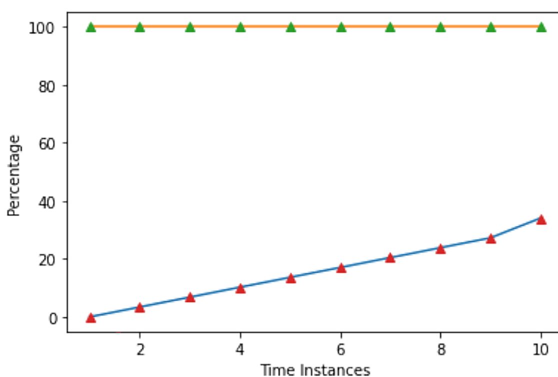

Fig. 4 The results of initial scheduling regarding the load balancing and the network traffic during the experiment. Additionally, the figure illustrates the cumulative growth of network traffic over time

various zones. As a result, some new jobs were scheduled close to their neighbors, which produced a slight increase in network traffic.

The experimental results in Figs. 9 and 10 illustrate that DAScheduler reduces the dependencies between containers in different zones from 16 to 4 and from 302 to 244 for the first and second test cases, respectively. Thus, our algorithm lowers the traffic cost by 42% and 28% for the first and second test cases, respectively. The experimental results further show that during migrating dependent containers, load balancing decreased to 88% and 90% in the first and second test cases, respectively. The last step of DAScheduler improves the load balancing to 89% and 91% for the first and second test cases, respectively.

6.4 Cloud Container Performance Results

In this section, we analyze the effectiveness of DAScheduler considering two factors: 1) the balance, and 2) network traffic. Figure 11 presents the experimental results which show that the performance is increased at time instances T2, T4, T5 and T6 and the algorithm reduces network traffic even though the load was balanced. In addition, whenever the network traffic increases then the performance decreases and vice versa.

6.4.1 Comparison with The State of The Art Algorithms

Zhao et al.'s [9] strategy is considered as the state-of-the-art study in this area of research which is used for comparison with our proposed algorithm.

In our study, we used the same evaluation metrics for measuring load balancing and network traffic and kept the experiment configuration the same to make a true comparison.

Zhao et al.'s [9] algorithm employs an objective function to manage the weight of load balance and network traffic while scheduling containers onto PMs. The scheduling strategy intends to configure α and β as traffic and load balancing, respectively.

During their experiments, Zhao et al. set up a variety of values for α and β, and demonstrated that α is not very sensitive when it becomes extremely large. This is expected as a limited number of zones and containers are considered. The algorithm proposes that

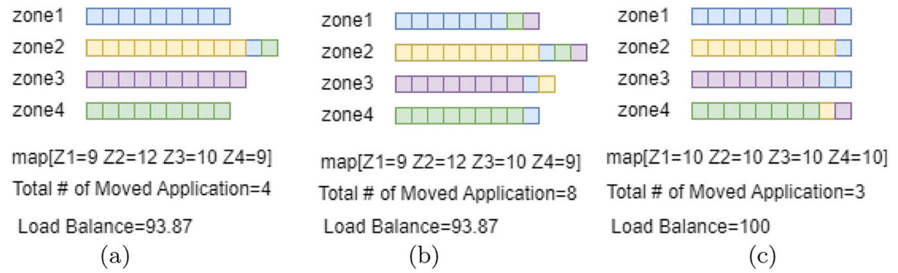

Fig. 5 Figures (a) and (b) show the process of grouping dependent containers, along with their percentage of grouping. Figure (c) depicts the final result after the algorithm achieved balanced load

when a zone reaches its full capacity, the algorithm will schedule the containers into the next zone. However, if there are considerably large resources in each zone; then most of the containers will be allocated to a small number of zones.

Despite the impressive reduction in network traffic of 89%, the load balancing technique used by Zhao et al. showed only a 50% improvement, suggesting that it may have been a bottleneck in the network and potentially limiting the overall performance. Furthermore, they reported that the best performance of their algorithm is achieved when $\beta = 0$, implying that load balance was not taken into account. In contrast, DAScheduler reduced network traffic by more than half and balanced the load to 90%.

The migration of containers between hosts is a fundamental operation in cloud computing that affects network traffic and resource allocation. The work by Zhao et al. has not adequately considered the potential side effects of container migration on network traffic. This omission leads to an incomplete assessment of the performance of their algorithm. In contrast, DAScheduler accounts for both container migration and the resulting network traffic overhead. Specifically, we introduce a strategy for migrating containers in small groups, which minimizes the impact of migration on network utilization and ultimately reduces the cost of intra-traffic. DAScheduler offers a more comprehensive solution for optimizing container migration and network traffic in cloud environments.

Lastly, Zhao et al. assumed in their experiments that all containers have the same capacity, which is not realistic. In DAScheduler, we considered that the jobs and containers are heterogeneous, which makes the simulation more accurate and realistic in terms of load balancing and network traffic. Moreover, Zhao et al. also did not consider the fact that new jobs can arrive at any time.

To sum up our comparison, we conclude that DAScheduler optimizes the performance of the containers by considering both load balancing and network traffic. Therefore, we believe that DAScheduler outperforms Zhao et al.'s [9] scheduling algorithm (Fig. 12).

7 Conclusion

In cloud computing, container technology has become a critical component for managing resources. It runs instances on top of the host operating system. However,

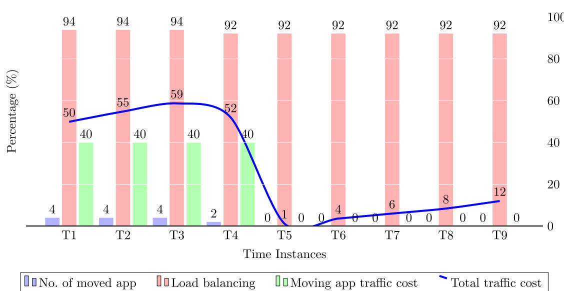

Fig. 6 The figure shows the results of grouping dependent containers for the first test case by illustrating the amount of network traffic, load balance, and the number of migrated containers

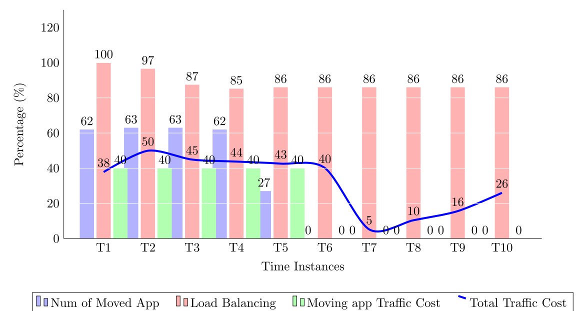

Fig. 7 The results of grouping dependent containers for the second test case. It illustrates the amount of network traffic, the load balance for all zones, and the number of migrated containers

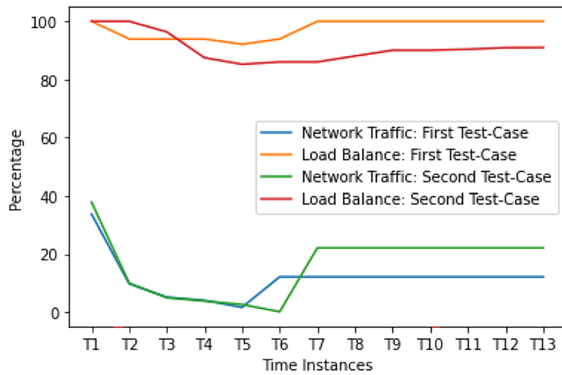

Fig. 8 The results of resource utilization reduction and load balancing during the experiment for both test cases

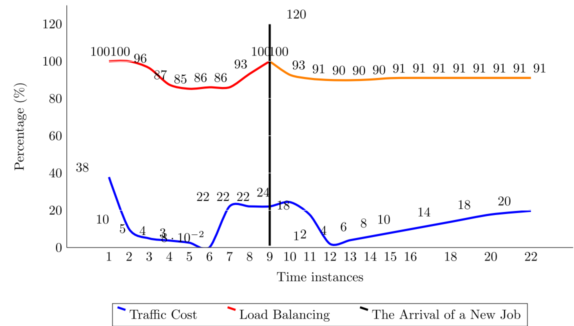

Fig. 10 Second real-time test case: the results of resource utilization and load balancing during real-time scheduling. The results begin after the vertical bar

resource contention can occur due to the sharing of resources, particularly with respect to network traffic. This can negatively impact latency when dependent jobs are deployed across different physical machines, leading to the need for frequent communication and further exacerbating network congestion.

Existing research primarily focuses on load balancing. In this research, we proposed a scheduling algorithm that considers both load balancing and job dependencies when deploying them into containers. The proposed algorithm reduces network contention by grouping dependent containers in the same zones. For this grouping, containers are migrated in steps to avoid a surge in network traffic. For each set of experiments, two test cases are considered. The first test case has fewer dependencies as compared to the second test case.

In the first test case of the first experiment, our proposed algorithm reduces container dependencies by 93%, while in the second test case, it reduces dependencies by 69.5%. These reductions in dependencies positively impact the performance of containers by decreasing network traffic by 64% and 50% for the two test cases, respectively. As compared to one of the state-of-the-art techniques, our algorithm achieved these reductions while maintaining load balance (Fig. 12).

During the simulation of real-time scheduling, we tested the ability of our algorithm to efficiently deploy containers as they arrived in the cloud. The results show that our algorithm reduced traffic contention by 42% and 28% in the first and second test cases, respectively.

Our study compared the performance of our proposed algorithm with state-of-the-art techniques, and the results demonstrate the superior ability and adaptability of our approach to deploying jobs into containers with optimal load balancing. Moreover, our algorithm effectively accommodates incoming jobs by dynamically placing them into containers upon arrival. The

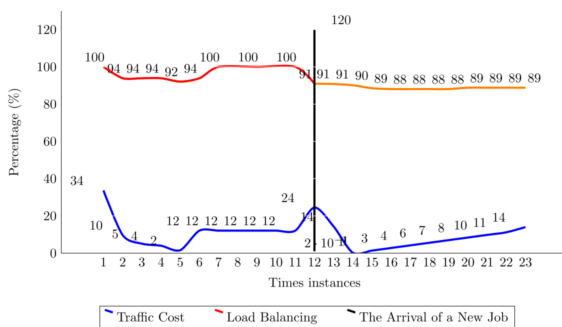

Fig. 9 First real-time test case: the results of resource utilization and load balancing during real-time scheduling. The results begin after the vertical bar

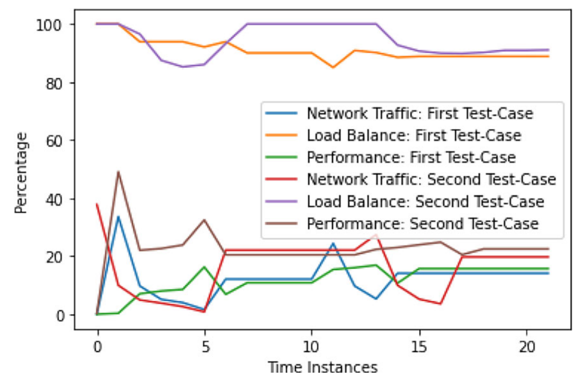

Fig. 11 The figure displays the system's performance, network traffic, and load balancing for both test cases

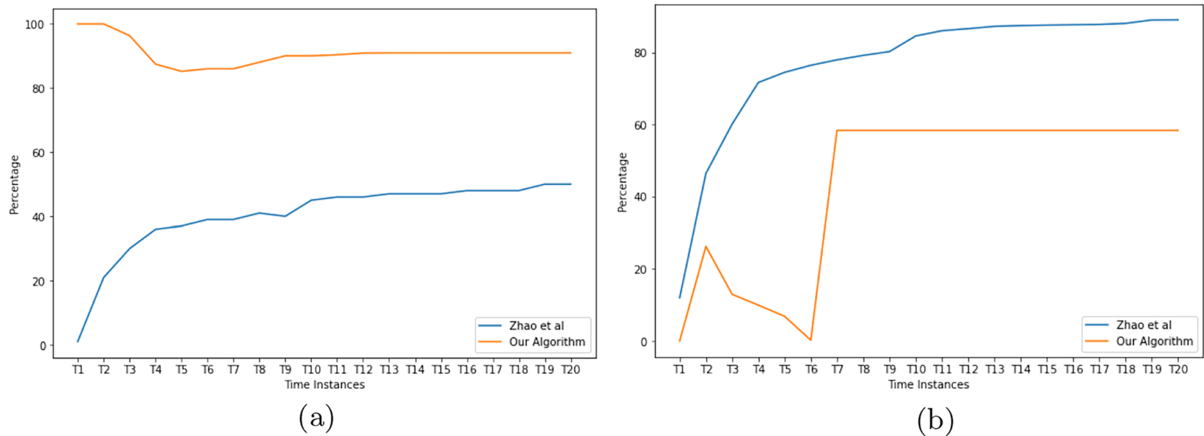

Fig. 12 Comparison between our and Zhao et al.'s [9] research. Figure (a) shows that our algorithm maintains the load balancing better than Zhao et al's. algorithm which achieves just 50% of the load balancing. However, Figure (b) shows the amount of the reduction in network traffic (RNTC) by the Zhao et al's. strategy is greater than our algorithm

most noteworthy feature of our algorithm is its ability to significantly reduce network contention while maintaining load balance, thereby resulting in improved performance for cloud computing environments. These findings highlight the potential impact of our approach on enhancing the efficiency and reliability of container-based systems.

8 Future Works

Two recommendations for future research can be made based on our findings. First, when designing scheduling algorithms for containers, it is crucial to consider all associated costs, such as execution, storage, and service costs. It is imperative to consider the impact of those costs on the overall system performance.

Secondly, we suggest utilizing a variety of performance metrics, such as the CiS2 proposed by [36] and [37], to evaluate their cloud-based algorithms. This will help to obtain a more comprehensive understanding of their proposed approach.

Acknowledgements The authors gratefully acknowledge the anonymous reviewers for their valuable suggestions

Author contributions AA conceptualized the study, conducted the literature review, formulated the research question, wrote the original draft, developed the methodology, and validated and analyzed the data. AD provided the initial idea, contributed to the theory, directed the methodology, supervised the project, and provided motivation. GMH contributed to the methodology, provide motivation, revised the first draft, and proofread the manuscript. All authors contributed to the writing of the manuscript, as well as the revision and proofreading of the final version

Funding Open Access funding enabled and organized by CAUL and its Member Institutions

Availability of data and materials The data is synthetic and will be available upon request

Declarations

Conflicts of interest The authors declare no conflict of interest

Ethical approval Ethics approval was not required

Consent to participate The authors provided consent to participate

Consent for publication The authors provided consent for publication

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Duan, Q.: In 2011 IEEE World Congress on Services (IEEE), pp. 548–555, (2011)
2. Xiang, J. Chen, L.: In Proceedings of the 2nd International Conference on Cryptography, Security and Privacy, pp. 159–164, (2018)
3. Singh, S. Singh, N.: In 2016 2nd international conference on applied and theoretical computing and communication technology (iCATccT) (IEEE), pp. 804–807, (2016)
4. Kayal, P.: In 2020 IEEE 6th World Forum on Internet of Things (WF-IoT) (IEEE), pp. 1–6 (2020)
5. Wan, X., Guan, X., Wang, T., Bai, G., Choi, B.Y.: Application deployment using microservice and docker containers: Framework and optimization. J Netw Comput Appl **119**, 97–109 (2018)
6. Budigiri, G. Baumann, C. Mühlberg, J.T. Truyen, E. Joosen, W.:In 2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit) (IEEE), pp. 407–412, (2021)
7. Cai, L., Qi, Y., Wei, W., Li, J.: Improving resource usages of containers through autotuning container resource parameters. IEEE Access **7**, 10853–108541 (2019)
8. McDaniel, S., Herbein, S., Taufer, M.: In 2015 IEEE International Conference on Cluster Computing (IEEE), pp. 490–491 (2015)
9. Zhao, D., Mohamed, M., Ludwig, H.: Locality aware scheduling for containers in cloud computing. IEEE Trans Cloud Comput **8**(2), 635–646 (2020)
10. da Silva Pinheiro, T.F., Pereira, P., Silva, B., Maciel, B.: A performance modeling framework for microservices-based cloud infrastructures. The Journal of Supercomputing pp. 1–42 (2022)
11. Kim, W.Y., Lee, J.S., Huh, E.N.:In Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication (Association for Computing Machinery, New York, NY, USA), IMCOM-17 (2017). https://doi.org/10.1145/3022227.3022243
12. Liu, B., Li, P., Lin, W., Shu, N., Li, Y., Chang, V.: A new container scheduling algorithm based on multi-objective optimization. Soft Comput **22**(23), 7741–7752 (2018)
13. Menouer, T., Darmon, P.:In 2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP) (IEEE), pp. 101–107 (2019)
14. Li, X., Zhou, J., Wei, X., Li, D., Qian, Z., Wu, J., Qin, X. Lu, S.:Topology-aware scheduling framework for microservice applications in cloud. IEEE Transactions on Parallel and Distributed Systems (2023)
15. Bao, B., Yang, H., Yao, Q., Guan, L., Zhang, J., Cheriet, M.:Resource allocation with edge cloud collaborative traffic prediction in integrated radio and optical networks. IEEE Access (2023)
16. Marchese, A., Tomarchio, O.: In: CLOSER, pp. 190–198, (2022)
17. Lu, W., Li, B., Wu, B.: In: IEEE 23rd International Conference on Computer Supported Cooperative Work in Design (CSCWD), pp. 380–385 (2019). https://doi.org/10.1109/CSCWD.2019.8791871
18. Nugroho, Y.N., Andika, F., Sari, R.F.: In: IEEE Conference on Application, Information and Network Security (AINS) (2019), pp. 89–93 (2019)
19. Duan, J., Guo, Z., Yang, Y.: In: IEEE conference on computer communications (INFO COM) (IEEE, 2015), pp. 136–144 (2015)
20. Zhang, J., Zhou, X., Ge, T., Wang, X., Hwang, T.: Joint task scheduling and containerizing for efficient edge computing. IEEE Trans Parallel Distrib Syst **32**(8), 2086–2100 (2021)
21. Shah, J., Dubaria, D.: In: IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC) (IEEE, 2019), pp. 0184–0189
22. Zafar, S., Bashir, A., Chaudhry, S.A.: On implementation of dctcp on three-tier and fat-tree data center network topologies. SpringerPlus 5(1), 1–18 (2016)
23. Petitet, A.C.A., Whaley, R.C., Dongarr, J.: A portable implementation of the high-performance linpack benchmark for distributed-memory computers (2018). https://netlib.org/benchmark/hpl/
24. Yee, A.J.:A multi-threaded pi-program (2022). http://www.numberworld.org/y-cruncher/
25. Performance testing and benchmarking for .net: Nbench (1990). https://nbench.io/
26. McCalpin, J.D.: Memory bandwidth: Stream benchmark performance results (n.d.). https://www.cs.virginia.edu/stream/
27. Netperf homepage (n.d.). https://hewlettpackard.github.io/netperf/
28. Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: In 2015 IEEE international symposium on performance analysis of systems and software (ISPASS) (IEEE), pp. 171–172 (2015)
29. Morabito, R., Kjällman, J., Komu, M.: In 2015 IEEE International Conference on cloud engineering (IEEE), pp. 386–393, (2015)
30. Xie, X.L., Wang, P., Wang, Q.: In 2017 13th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD) (IEEE), pp. 2137–2141 (2017)
31. Kozhirbayev, Z., Sinnott, R.O.: A performance comparison of container-based technologies for the cloud. Futur Gener Comput Syst **68**, 175–182 (2017)
32. Hellemans, T., Van Houdt, B.: Improved load balancing in large scale systems using attained service time reporting. IEEE/ACM Trans Networking **30**(1), 341–353 (2022). https://doi.org/10.1109/TNET.2021.3110186
33. Guo, C., Yuan, L., Xiang, D., Dang, Y., Huang, R., Maltz, D., Liu, Z., Wang, V., Pang, B., Chen, H., et al.: In Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, pp. 139–152, (2015)
34. Zhou, R., Li, Z., Wu, C.: Scheduling frameworks for cloud container services. IEEE/ACM Trans Networking **26**(1), 436–450 (2018). https://doi.org/10.1109/TNET.2017.2781200

35. Guo, Y., Yao, W.: In NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium, pp. 1–6 (2018). https://doi.org/10.1109/NOMS.2018.8406285
36. Juiz, C., Bermejo, B.: Thecis 2: a new metric for performance and energy trade-off in consolidated servers. Clust Comput **23**(4), 2769–2788 (2020)
37. Bermejo, B., Juiz, C.: A general method for evaluating the overhead when consolidating servers: performance degradation in virtual machines and containers. The Journal of Supercomputing **78**(9), 11345–11372 (2022)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.