Concurrent container scheduling on heterogeneous clusters with multi-resource constraints

Hu, Y.; Zhou, H.; de Laat, C.; Zhao, Z.

DOI
10.1016/j.future.2019.08.025

Publication date
2020

Document Version
Final published version

Published in
Future Generation Computer Systems

Future Generation Computer Systems 102 (2020) 562–573

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Concurrent container scheduling on heterogeneous clusters with multi-resource constraints

Yang Hu [a,b], Huan Zhou [a], Cees de Laat [a], Zhiming Zhao [a,*]

[a] Institute for Informatics, University of Amsterdam, The Netherlands
[b] School of Computer Science, National University of Defense Technology, China

ARTICLE INFO

Article history:
Received 10 April 2019
Received in revised form 22 July 2019
Accepted 27 August 2019
Available online 4 September 2019

Keywords:
Container
Concurrent scheduling
Heterogeneous cluster
Multi-resource constraints

ABSTRACT

By effectively virtualizing operating systems and encapsulating necessary runtime contexts of software components and services, container technologies can significantly improve portability and efficiency for distributed application deployment. It flexibly extends virtual machine based cloud (Infrastructure-as-a-Service) as a much lighter virtual environment (container cluster) for agile application management. However, existing container management systems are not capable of handling concurrent requests efficiently, particularly for the underlying clusters with heterogeneous machines and the requested containers with multi-resource demands. In this paper, we propose an Enhanced Container Scheduler (ECSched) for efficiently scheduling concurrent container requests on heterogeneous clusters with multi-resource constraints. We formulate the container scheduling problem as a minimum cost flow problem (MCFP), and represent the container requirements using a specific graph data structure (flow network). ECSched affords flexibility in constructing the flow network based on a batch of concurrent requests, and performs the MCFP algorithm to schedule the concurrent requests in an online manner. We evaluate ECSched in different testbed clusters, and measure the scheduling overhead with large-scale simulations. The experimental results show that ECSched outperforms state-of-the-art container schedulers in container performance and resource efficiency, and only introduces a small and acceptable scheduling overhead in large-scale clusters.

1. Introduction

Container technologies effectively virtualize the operating system and are becoming increasingly popular in cloud computing. By encapsulating runtime contexts of software components and services, containers significantly improve portability and efficiency for cloud application deployment. Major cloud providers have provided container-based cloud services to cater for this popularity, such as Amazon Elastic Container Service [1] and Azure Container Service [2]. Meanwhile, container orchestration platforms, such as Docker Swarm [3], Mesosphere Marathon [4], and Google Kubernetes [5], are emerging to provide container-based infrastructure for automating deployment, scaling, and management of containers on underlying clusters.

Typically, Infrastructure as a Service (IaaS) offered by the cloud providers (e.g., Amazon EC2 [6], Microsoft Azure [7]) relies on the underlying Virtual Machines (VMs). Compared with VM-based infrastructure, container-based infrastructure has some new features.

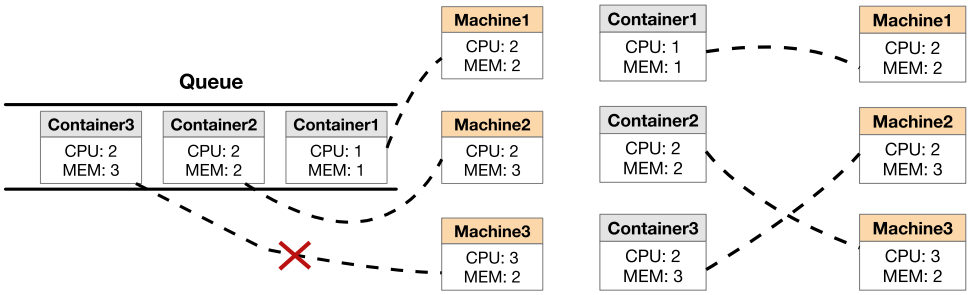
- It can be deployed on both physical and virtual machines, and the highly diverse configuration of VMs makes the clustered machines more heterogeneous.
- It can provide fine-grained resource allocation due to the operating-system-level virtualization techniques of containers, which is much more flexible than predefined VM types in VM-based infrastructure.
- It can support users specifying affinities among containers (e.g., Affinity in Kubernetes) for a distributed application, which facilitates container orchestration over the cluster.

With these new features, container-based infrastructure imposes emerging and stringent requirements on container scheduling in order to provide performance guarantee for deployed applications.

1. The resources demanded by a container are often a combination of multiple resources (CPU, memory, network, etc.), which have to be satisfied by the underlying container cluster; it becomes extremely challenging when the nodes have diverse capacity and capability.
2. Containers of a distributed application often have strong affinity with other containers (due to frequent data communication) or specific machines (due to data locality).

* Corresponding author.
E-mail addresses: y.hu@uva.nl (Y. Hu), h.zhou@uva.nl (H. Zhou), delaat@uva.nl (C.d. Laat), z.zhao@uva.nl (Z. Zhao).

https://doi.org/10.1016/j.future.2019.08.025



(a) Container-by-container Scheduling (b) Concurrent Scheduling

Fig. 1. An example of container-by-container scheduling and concurrent scheduling for three concurrent requests.

Placing containers on the appropriate node can significantly reduce the latency of container communication or decrease the volume of data transferred. Hence, the affinity has to be taken into account when scheduling containers.

3. The high scheduling overhead in large clusters may hurt the performance of applications with high-quality constraints [8–11], e.g., real-time analytics [12,13]. Moreover, the scheduling algorithms are frequently invoked during the application execution, in particular when scaling out or recovering from failure, which often have critical time constraints. Thus, the scheduling overhead should be small so that the scheduler is able to scale to large clusters.

Meanwhile, with the adoption of cloud services and the scale of applications increase, modern cloud platforms have to deal with a large number of concurrent requests at the same time. By analyzing the Google cluster trace [14], the scheduler needs to make hundreds of task placement decisions per second in peak hours. When considering the above requirements, it inevitably introduces new challenges to the container schedulers. In recent years, container management and scheduling have attracted quite a lot of research attentions. A queue-based scheduler is widely used in the orchestration platforms, such as Marathon [15], Swarm [3] and Kubernetes [16]. All deployment requests first enter a queue; the scheduler fetches requests from the queue and processes one container (one pod in Kubernetes) at a time. Regarding the scheduling algorithms to the queue-based schedulers, variants of heuristic packing algorithms, such as Best-Fit Decreasing (BFD) and First-Fit Decreasing (FFD) [17,18], are often adopted to achieve practical solutions.

Container-by-container scheduling has the advantage for making parallel decisions on distributed deployment [12,19], but it also has crucial limitations for achieving high-quality placements of the entire workloads (concurrent tasks), due to its lack of global view on the waiting containers. For instance, the scheduler makes a decision early for a requested container, which would restricts its choices for the waiting containers. Fig. 1 shows an example of container-by-container scheduling and concurrent scheduling for three concurrent requests, where the resource demands of containers and the available resources of machines are depicted. For the container-by-container scheduling, if we apply a simple scheduling algorithm (i.e., First come, first served), Container3 cannot be scheduled at this moment. It is because Machine3 does not have enough resources to run Container3. For the concurrent scheduling, as the scheduler has a global view of the entire workloads, it could schedule all the containers to the machines. In the optimization of service placement or VM placement, the problem of scheduling a batch of concurrent requests can be usually formulated as an integer programming problem [20,21] or a mixed integer programming problem [22]. However, those are NP-hard [23].

In this paper, we propose an Enhanced Container Scheduler (ECSched) for efficiently scheduling concurrent container requests with multi-resource constraints on heterogeneous clusters. We formulate the container scheduling problem as a minimum-cost flow problem (MCFP) and represent the container requirements using a specific graph data structure (flow network). In the flow network, we propose a novel approach to encode the multi-resource demands and affinity requirements of requested containers. By analyzing the properties of different classical MCFP algorithms, we choose an appropriate variant of successive shortest path algorithm implemented in ECSched. In our implementation, ECSched first constructs the flow network based on a batch of concurrent requests, and then performs the MCFP algorithm to schedule the concurrent requests at the same time. To evaluate the scheduling quality, we compare the container performance and the resource efficiency of ECSched and state-of-the-art container schedulers in different testbed clusters. To understand the scheduling overhead, we measure the algorithm runtime of ECSched and state-of-the-art container schedulers by performing large-scale simulations.

we summarize our contributions as follows:

- We analyze the characteristics of existing container schedulers and identify the specific challenges in handling concurrent container requests with multi-resource constraints on heterogeneous clusters.
- We model the concurrent container scheduling problem as a minimum cost flow problem (MCFP) and propose several novel methods to handle the requirements of container requests in the flow network. We implement a prototype scheduler ECSched with an appropriate MCFP algorithm.
- We show that ECSched outperforms state-of-the-art container schedulers in container performance and resource efficiency, and only introduces a small and acceptable scheduling overhead in large-scale clusters.

The rest of the paper is organized as follows. In Section 2, we formulate the container scheduling problem, and analyze the deployment requirements of requested containers. In Section 3, we discuss the minimum cost flow problem. In Section 4, we present the design and implementation of our approach ECSched. In Section 5, we compare the performance of ECSched against that of state-of-the-art container schedulers through extensive experiments. In Section 6, we discuss the related work. In Section 7, we summarize our results and present future work.

2. Problem formulation

In this section, we first present the formulation of the containers scheduling problem. Then, we analyze different deployment requirements of container requests. The notation used in the work is presented in Table 1.

Table 1
Notation and description.

Notation	Description
$\mathcal{M}$	Set of heterogeneous machines in the cluster: $\mathcal{M} = \{m_1, m_2, \ldots, m_M\}$
M	Number of the machines: $M = \lvert\mathcal{M}\rvert$
$\mathcal{R}$	Set of resource types: $\mathcal{R} = \{r_1, r_2, \ldots, r_R\}$
R	Number of the resource types: $R = \lvert\mathcal{R}\rvert$
V_i	Vector of available resources on machine m_i: $V_i = (v_i^1, v_i^2, \ldots, v_i^R)$
v_i^j	Amount of resource r_j available on machine m_i
$\mathcal{C}$	Set of concurrent container requests at one moment: $\mathcal{C} = \{c_1, c_2, \ldots, c_N\}$
N	Number of the concurrent requests: $N = \lvert\mathcal{C}\rvert$
D_i	Vector of resource demands of container c_i: $D_i = (d_i^1, d_i^2, \ldots, d_i^R)$
d_i^j	Amount of resource r_j that container c_i demands
$\mathbb{A}$	Matrix of container affinity: $\mathbb{A} = [a_{ij}]_{N\times N}$, where $a_{ij} = 1$ if container c_i has an affinity with container c_j, otherwise $a_{ij} = 0$
$\mathbb{B}$	Matrix of machine affinity: $\mathbb{B} = [b_{ij}]_{N\times M}$, where $b_{ij} = 1$ if container c_i has an affinity with machine m_j, otherwise $b_{ij} = 0$
$\mathbb{X}$	A placement solution: $\mathbb{X} = [x_{ij}]_{N\times M}$, where $x_{ij} = 1$ if container c_i is to be placed on machine m_j, otherwise $x_{ij} = 0$

2.1. Model description

In container-based infrastructure, the cluster is typically composed of a set of networked heterogeneous machines $\mathcal{M} = \{m_1, m_2, ..., m_M\}$, where $M = |\mathcal{M}|$ is the number of machines. We consider R types of resources $\mathcal{R} = \{r_1, r_2, \ldots, r_R\}$ (e.g., CPU, memory, or network bandwidth) in each machine. For machine m_i, let $V_i = (v_i^1, v_i^2, \ldots, v_i^R)$ be the vector of its resource capacities where the element v_i^j denotes the total amount of resource r_j available on machine m_i.

We consider the container requests continuously arrive over time. At one moment, we assume there is a set of concurrent requests $\mathcal{C} = \{c_1, c_2, \ldots, c_N\}$ that are to be deployed on the cluster, and $N = |\mathcal{C}|$ is the number of requests. For container c_i, let $D_i = (d_i^1, d_i^2, \ldots, d_i^R)$ be the vector of its resource demands, where the element d_i^j denotes the amount of resource r_j that the container c_i demands. To affinity specification, let 0–1 matrix $\mathbb{A} = [a_{ij}]_{N\times N}$ denote the container affinity. If $a_{ij} = 1$, it means that the container c_i has an affinity with container c_j. Let 0–1 matrix $\mathbb{B} = [b_{ij}]_{N\times M}$ denote the machine affinity. If $b_{ij} = 1$, it means that the container c_i has an affinity with machine m_j.

Next, we model a placement solution of the scheduler. Note that a placement solution means a mapping of containers to machines on the cluster in this work. Let 0–1 matrix $\mathbb{X} = [x_{ij}]_{N\times M}$ denote a solution, where x_{ij} is 1 if container c_i is to be deployed on machine m_j.

2.2. Deployment requirements

By analyzing the features of container-based infrastructure, a placement solution is desired to satisfy the following requirements.

2.2.1. Multi-resource guarantee

Providing multi-resource guarantee for each container on the heterogeneous cluster is the primary requirement to the scheduler. Container-based infrastructure, which has the advantages and benefits of container techniques inherently, can allocate resources in a more fine-grained way than VM-based infrastructure; it facilitates the flexibility of resource allocation for applications. Given the constraints of Service Level Agreements (SLAs) with users, different types of resource demands should be at least guaranteed with a placement solution so that SLAs are not violated. Thus, the resource demands of the containers in one machine should not exceed its capacity.

$$\sum_{c_i\in\mathbb{C}} x_{ij}\cdot d_i^k \le v_j^k \quad \forall m_j\in\mathcal{M},\ \forall r_k\in\mathcal{R} \tag{1}$$

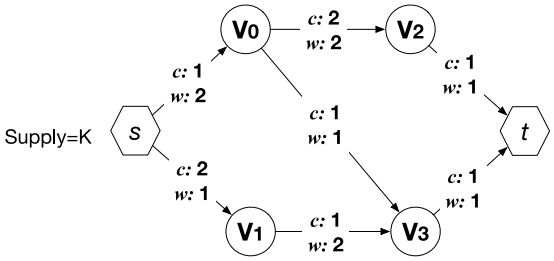


Fig. 2. An example of a flow network.

2.2.2. Affinity awareness

In container-based infrastructure, users can specify the affinity of containers in a deployment request, which represents the demands of data communication or the location of data input. As distributed applications transfer data frequently, especially data-intensive applications, the network condition would directly affect the overall performance. Considering the influence of the network, the scheduler should be aware of the affinity requirements so that it can make effective use of this information to adjust container placements. The intuitive and effective solution is to co-locate the containers which have affinity to others on the same machine,

$$\sum_{m_k\in\mathcal{M}} x_{ik}\cdot x_{jk} \ge a_{ij} \quad \forall c_i, \forall c_j \in \mathcal{C} \tag{2}$$

and place the container on the affinity machine.

$$x_{ik} \ge b_{ik} \quad \forall c_i\in\mathcal{C}, \forall m_k\in\mathcal{M} \tag{3}$$

Accordingly, the challenge for a scheduler is how to efficiently schedule the concurrent requests while satisfying all the deployment requirements of requested containers.

3. Minimum cost flow problem

As existing queue-based schedulers process one container at a time, the other waiting requests cannot be considered in the decision-making phase. Consequently, it is hard for a scheduler to achieve high-quality placements, since it makes a separate decision for each container. In this work, we choose a graph-based approach to model the container scheduling problem as minimum cost flow problem (MCFP) [24], which can perform the container scheduling of concurrent requests at the same time.

The minimum cost flow problem is an optimization and decision problem to find the minimum-cost way of sending a certain amount of flow through a flow network. A flow network is a directed graph $G = (V, E)$ with a source node $s \in V$ and a sink node $t \in V$, where each edge $e_{u,v} \in E$ has a capacity $c_{u,v} > 0$ and a unit transportation cost $w_{u,v}$. Fig. 2 shows an example of a flow network.

In the flow network, the cost of sending a flow of $f_{u,v} \ge 0$ units along the edge $e_{u,v}$ is $f_{u,v}\cdot w_{u,v}$. The problem requires K units of flow (source node with a supply of K units) to be sent from source

s to sink t, and the goal is to minimize the total cost of the flow over all edges:

$$\text{Minimize} \quad \sum_{e_{u,v} \in E} f_{u,v} \cdot w_{u,v} \tag{4}$$

$$\text{subject to:} \quad f_{u,v} \leq c_{u,v} \tag{5}$$

$$\sum_{x \in V} f_{x,u} = \sum_{x \in V} f_{u,x} \ (u \neq s, t) \tag{6}$$

$$\sum_{x \in V} f_{s,x} = \sum_{x \in V} f_{x,t} = K \tag{7}$$

Eq. (5) guarantees that the amount of the flow that goes through an edge cannot exceed its capacity. Eq. (6) guarantees that the amount of the flow that goes into a node is equal to the amount of the flow that comes out of the node, except source node and sink node. Eq. (7) guarantees that both the amount of the flow that comes out of source node and the amount of the flow that goes into sink node are equal to K. Eq. (4) expresses the goal of the minimum cost flow problem.

In this paper, we strive to schedule a batch of concurrent container requests efficiently at a moment. To simplify the problem, for each pair of a container request and a machine, we choose to use a scalar value to indicate the fitness level between them. The objective of our work is thus to find a placement solution (mapping between the concurrent requests and the machines) that maximizes the total values at each moment. As MCFP is a well-studied problem in the past years [24], there are many effective and efficient algorithms [25,26] that have been proposed to solve the MCFP in a polynomial time. If we can convert the container scheduling problem to the MCFP with a flow network, we could also solve the container scheduling problem in a polynomial time. This is because a solution of MCFP can be extracted as a mapping between the nodes in the flow network. By setting the corresponding capacity and transportation cost on the edges in the flow network, MCFP algorithms can find the optimal placement solution (mapping) that maximizes the total values for a batch of concurrent container requests. Consequently, the question is how to convert the container scheduling problem to the MCFP, and what MCFP algorithm is used to solve the problem.

4. ECSched approach

In this section, we describe how to construct the specific graph data structure (flow network) of MCFP to solve the container scheduling problem, what MCFP algorithms to use, and how to build ECSched scheduler.

4.1. Flow network structure

To map the container scheduling problem to the MCFP, we formulate the problem using a specific structure of flow networks. Fig. 3 shows a case flow network of tackling the container scheduling problem, but we only annotate the capacity on the edges in the figure. The flow network corresponds to an instantaneous status of the container cluster, while encoding a set of requested containers and clustered machines. The overall structure of the flow network can be described as follows.

- **Source Node:** The source node s on the left hand with a supply of K units of flow, which represents how many containers can be handled at a time in our context. The maximum supply is the total number of requested containers in the scheduler ($K = N$).
- **Container Node:** Each requested container in the flow network is represented as a node C_i and has an edge from source node s with a capacity of 1 unit.

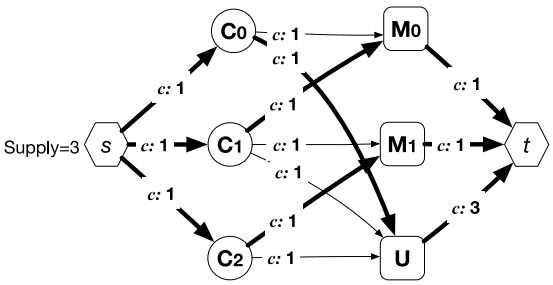


Fig. 3. A case flow network of tackling the container scheduling problem.

- **Machine Node:** Each clustered machine in the flow network is represented as a node M_i and has an edge from a container node with a capacity of 1 unit if the machine is eligible to place the container.
- **Unscheduled Node:** Inspired by previous works [27,28], we add a new node in the flow network, which called unscheduled node U. All container nodes have an outgoing edge to the node U with a capacity of 1 unit.
- **Sink Node:** The sink node t on the right hand is the place to drain off the flow. All machine nodes have an edge to the sink node with a capacity of 1 unit, and the unscheduled node has an edge to the sink node with a capacity of N units.

MCFP algorithms would optimally route the flow from the source to the sink without exceeding the capacity constraints on any edge. A path of one MCFP solution first gets to a container node from the source node, and then reaches the sink node through a machine node or the unscheduled node. Thus, if a path goes through a machine node, it corresponds to an assignment for the container. Otherwise, if a path goes through an unscheduled node, it does not schedule the container at this moment.

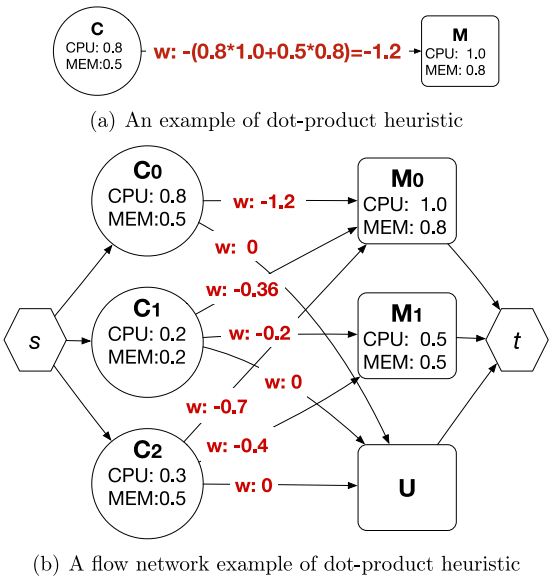
For instance, all bold edges can be one possible solution returned by MCFP algorithms as shown in Fig. 3. The solution corresponds to a placement solution: Container0 is not scheduled at this moment; Container1 is assigned to Machine0; Container2 is assigned to Machine1. Accordingly, the scheduler can successively perform MCFP algorithms to continuously schedule containers.

4.2. Encoding deployment requirements

As the goal of the MCFP problem is to minimize the total cost of the flow over all edges, flexible assignment of the costs on the edges can make the MCFP algorithms return a placement solution which we desire for container scheduling. Considering two deployment requirements as described in the previous section, we propose following methods to encode them in the flow network.

4.2.1. Multi-resource guarantee

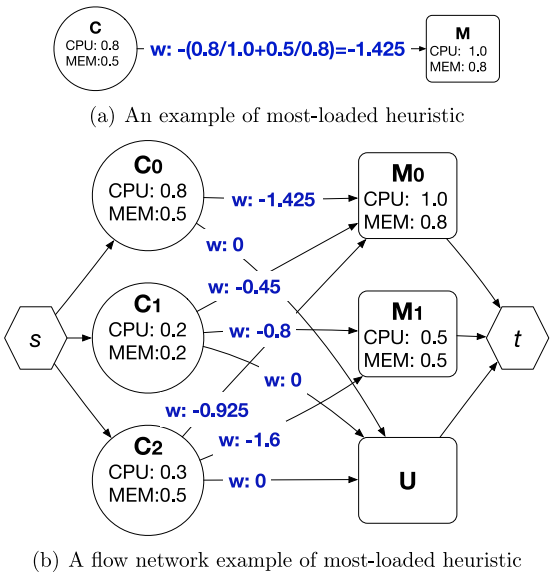
Providing multi-resource guarantee for each requested container is the primary objective of the container scheduling. In order to make the values of different resources comparable to each other and easy to handle, we first normalize the resource number to be the fraction of the maximum capacity in the cluster independently. For instance, there are two requested containers with resource demands: (CPU: 1 core, MEM: 2 GB) and (CPU: 2 cores, MEM: 1 GB), and there are two machines in the cluster with resource capacities: (CPU: 4 cores, MEM: 2 GB) and (CPU: 2 cores, MEM: 4 GB). After normalization, the vector of container resource demands becomes (CPU: 0.25, MEM: 0.5) and (CPU: 0.5, MEM: 0.25); the vector of machine resource capacities becomes (CPU: 1.0, MEM: 0.5) and (CPU: 0.5, MEM: 1.0), since the maximum



(a) An example of dot-product heuristic

(b) A flow network example of dot-product heuristic

Fig. 4. An example for encoding the multi-resource requirements based on dot-product heuristic.



(a) An example of most-loaded heuristic

(b) A flow network example of most-loaded heuristic

Fig. 5. An example for encoding the multi-resource requirements based on most-loaded heuristic.

number of CPU cores is 4 and the maximum capacity of memory is 4 GB in the cluster.

Next, the scheduler would construct the flow network as mentioned earlier. To construct the flow network, the scheduler checks whether the machines in the cluster have sufficient resources to place the requested containers. If a machine is eligible for a container, it adds an edge from the container node to the machine node with a capacity of 1 unit. The key challenge here is how to assign the costs on the edges to distinguish the quality (fitness level) of different mappings between containers and machines. In this work, we introduce two strategies which are inspired by vector bin packing algorithms: dot-product heuristic [29] and most-loaded heuristic.

- **Dot-product:** In this heuristic, we prioritize different placements based on the dot product. Here, the dot product between the demand vector of container c_i and the capacity vector of machine m_j is defined as $dp_{ij} = \sum_{r_k \in \mathcal{R}} d_i^k v_j^k$. Fig. 4(a) shows an example. The dot product between the container and the machine in the figure can be calculated as: $0.8 \times 1.0 + 0.5 \times 0.8 = 1.2$. The idea of this heuristic is that it takes into account not only the resource demands of containers but also how well its demands align with the resource capacities of machines; the dot product implies the degree of the alignment. Thus, for this heuristic, the higher dp_{ij} is, the better the placement is. In MCFP, the cost on the edge is inversely related, which is a flow is better if the cost of the flow is lower. Therefore, the cost on the edge between the container node and the machine node is assigned to $-dp_{ij}$ in the flow network. For the edge from container node to unscheduled node, the cost is assigned to 0 which is the highest. A flow network example is shown in Fig. 4(b).
- **Most-loaded:** In this heuristic, we prioritize different placements based on the load situations of the machines. The container tends to be placed on the most loaded machine in the cluster. In this cost model, it is also based on a scalar value which is defined as $ml_{ij} = \sum_{r_k \in \mathcal{R}} \frac{d_i^k}{v_j^k}$, which implies the mapping quality between the container c_i and the machine m_j. Fig. 5(a) shows an example. The value between the container and the machine in the figure can be calculated as: $\frac{0.8}{1.0} + \frac{0.5}{0.8} = 1.425$. Thus, the higher ml_{ij} is, the more loaded the machine is in this heuristic. Similar to dot-product heuristic, the cost on the edge is assigned to $-ml_{ij}$. For the edge from container node to unscheduled node, the cost is also assigned to 0. A flow network example is shown in Fig. 5(b).

4.2.2. Affinity awareness

When submitting a deployment request, users can specify the affinities among the containers. It represents the demands of data communication or the location of data input. An appropriate placement of containers can lead to lower network latency and better network utilization. Thus, the location of containers is crucial for the overall performance. In the flow network, it is flexible to handle container affinity (co-located on the same machine) and machine affinity (located on a specific machine) by dynamically adjusting the edges in the flow network.

- **Machine affinity:** Considering the location of input data or container image, users would specify the machine affinity to indicate the preferred machine. Placing the container on the specified machine can reduce network transmission time significantly. Thus, we adjust the flow network to only connect the container with the preferred machine, which can limit placement options of the requested container. Fig. 6 shows an example with machine affinity, where container c_1 has a machine affinity to machine m_1. In the example, container c_1 has only one edge to machine m_1 but no edge to machine m_0 that is also eligible for container c_1. Accordingly, container c_1 can only be scheduled to machine m_1.
- **Container affinity:** Considering the network latency within the containers, users would specify the container affinity among certain containers. Placing these containers on the same machine can reduce network latency significantly. In the flow network, we add a new node, called aggregator node A_i, to merge affinity containers. Fig. 7 shows an example with container affinity, where container c_0 and container c_1 have an affinity. In the example, we add an aggregator node A_0 in the flow network. Both container c_0 and container c_1 have an edge to aggregator node A_0. Hence, the scheduler would treat these two container nodes as one node to perform scheduling.

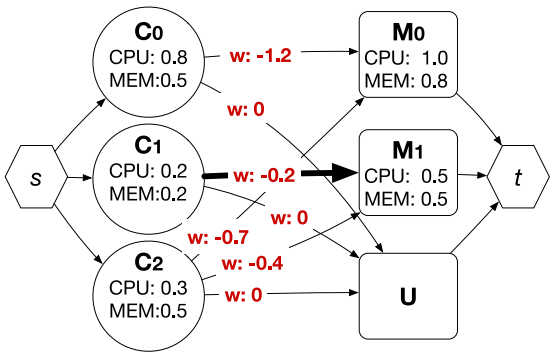


Fig. 6. A flow network for encoding the machine affinity requirements based on dot-product heuristic.

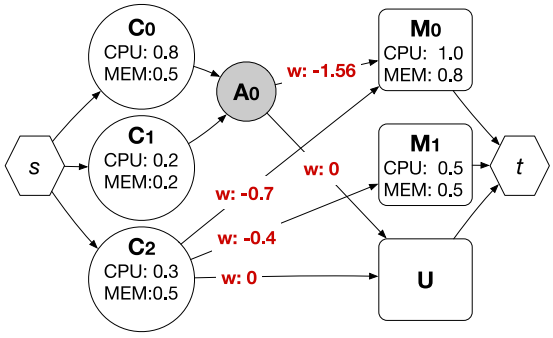


Fig. 7. A flow network for encoding the container affinity requirements based on dot-product heuristic.

4.3. MCFP algorithms

After constructing the flow network, the scheduler will perform MCFP algorithms to find the optimal routing solution with respect to the costs we have assigned. In this section, we discuss two kinds of classical MCFP algorithms: **cycle canceling algorithm** and **successive shortest path algorithm**. We then explain the MCFP algorithm that we implemented in ECSched.

The simplest MCFP algorithm is cycle canceling algorithm [30]. This algorithm maintains a feasible solution meeting Eqs. (5)–(7) and at every iteration attempts to improve its optimality. The algorithm first establishes a feasible flow in the flow network by solving a maximum flow problem [24]. Then it iteratively finds negative cost-directed cycles in the residual network and augments flows on these cycles. The residual network is defined as follows. In this flow network, each edge $e_{u,v} \in E$ with a capacity $c_{u,v} > 0$ and a unit transportation cost $w_{u,v}$ is replaced by two edges: $e_{u,v}$ and $e_{v,u}$. Then edge $e_{u,v}$ has a residual capacity $r_{u,v} = c_{u,v} - f_{u,v}$ and a unit transportation cost $w_{u,v}$, while edge $e_{v,u}$ has a residual capacity $r_{v,u} = f_{u,v}$ and a unit transportation cost $-w_{u,v}$. All constraints of MCFP can also be applied in the residual network. The cycle canceling algorithm terminates when the residual network contains no negative cost-directed cycle.

The cycle canceling algorithm maintains feasibility of the solution at every step and attempts to achieve optimality. In contrast, the successive shortest path algorithm [31] maintains optimality of the solution at every step and strives to attain feasibility. At each step, the algorithm sends flow from the source node s to the sink node t along the shortest path in the residual network. The algorithm terminates when the current solution meets Eqs. (5)–(7) of MCFP.

Based on these two classical MCFP algorithms, many optimization methods and scaling algorithms have been proposed [25,26,32–34]. Known worst-case complexity bounds on the MCFP are $O(N^2C\log(N))$ [34] for the successive shortest path based algorithm, and $O(N^2M\log(NW))$ [25] for the cycle canceling based algorithm. N is the number of nodes; M is the number of edges; C is the number of the largest edge capacity; W is the number of the largest edge cost. In our container scheduling problem, it is the case as $M > N > C > W$. Thus, the successive shortest path based algorithm would perform better due to the complexities ($C\log(N) < M\log(NW)$). On the other hand, the cost on the edges is not integer in the flow network as we defined in above sections; it can be easier to solve through the successive shortest path based algorithm. Therefore, we choose to implement a variant of successive shortest path algorithm in our ECSched.

4.4. Implemention

The Implemention of ECSched is based on a heartbeat mechanism. On a heartbeat, ECSched first fetches a set of container requests to construct a flow network, and then performs the MCFP algorithm to place the requested containers. The details are explained as follows.

4.4.1. Constructing flow network

First, ECSched would fetch a certain number of concurrent container requests from the queue system. In our implementation, users can customize the maximum number of requests that ECSsched can fetch. Considering the tradeoff between scheduling quality and scheduling overhead, selecting a proper number is crucial for the overall performance. We discuss the tradeoff in Section 5.5. Then, ECSched constructs the flow network based on these concurrent container requests. In addition, we discuss the time complexity of the flow network construction. We assume the number of container requests that are fetched from the queue system is n. The number of machines in the cluster is M as defined in the model description. To construct the flow network, we first group the container requests according to the requirements of container affinity, whose time complexity is $O(n^2)$. Then, we build the flow network and add the edges according to the strategies we described earlier, whose time complexity is $O(nM)$. Hence, the overall time complexity of the flow network construction is $O(n^2 + nM)$.

4.4.2. Placing requested containers

Next, ECSched would perform the MCFP algorithm (a variant of successive shortest path algorithm) to find an optimal flow solution over the flow network. Then, ECSched extract the container placements out of the optimal flow solution. According to the placements, ECSched places the scheduled containers on the corresponding machines and puts the unscheduled containers back to the queue system for the next scheduling.

5. Evaluation

We implement ECSched with a container manager and a variant of MCFP algorithm in Python. In this section, we evaluate our ECSched in testbed clusters of ExoGENI [35] experimental environment to compare the scheduling quality with state-of-the-art container schedulers. In order to understand the scheduling overhead of ECSched, we measure the algorithm runtime by performing large-scale simulations.

5.1. Experimental setup

Cluster. We create two different container clusters with 30 virtual machines (VM) in ExoGENI [35] which is a multi-domain Infrastructure-as-a-Service testbed. For the first cluster, we use 30 homogeneous VMs of "XOLarge" type (2-core CPU, 6 GB of memory) in the testbed. Considering the heterogeneity, we choose three types of VM configurations for the second cluster. The cluster is composed of 10 VMs of "XOMedium" type (1-core CPU, 3 GB of memory), 10 VMs of "XOLarge" type (2-core CPU, 6 GB of memory) and 10 VMs of "XOXLarge" type (4-core CPU, 12 GB of memory). After normalization, the capacity vectors of the machines in the homogeneous cluster are all: (CPU: 0.5, MEM: 0.5); the capacity vectors of the machines in the heterogeneous cluster are: (CPU: 0.25, MEM: 0.25), (CPU: 0.5, MEM: 0.5), and (CPU: 1, MEM: 1) respectively.

Workloads. To test our prototype, we yield container requests based on the Google cluster trace [14], which provides data from a 12,500-machine cluster over a month-long period. As we choose to spend 6 h at each experiment, we analyzed the trace of the first 6 h. There are around 100,000 tasks completed within the first 6 h, and the average duration of the tasks is around 740 s. Considering the scale of our testbed cluster, we randomly sample 2500 tasks (2.5%) from them at each experiment. The generator yields container requests according to following aspects from the trace: task submission times, task durations and task resource requirements. The resource requirements have been normalized in the trace. Additionally, we add the requirements of container affinity and machine affinity with a probability according to the task constraints in the trace [14] (ensure that affinity containers can be packed into a machine).

ECSched. We implement two strategies in our scheduler. ECSched-dp is based on dot-product heuristic; ECSched-ml is based on most-loaded heuristic. For the heartbeat mechanism, the scheduling interval is set to be 100 ms in the scheduler. Unless otherwise specified, the maximum number of container requests that ECSched can fetch from the queue system on a heartbeat is 100.

Baselines. We compare ECSched to the state-of-the-art scheduling algorithms implemented in two container orchestration systems: Google Kubernetes [16] and Docker Swarm [3]. Under multi-resource requirements, the default scheduler of Kubernetes tends to distribute pods (smallest deployable units in Kubernetes) evenly across the cluster to balance the overall resource usage, while the scheduler of Swarm tends to place containers on the most loaded machines to improve resource utilization over the cluster. Both are queue-based schedulers, which process one unit at a time.

Metrics. The primary metric to quantify container performance is the improvement in the average container completion time. We define the *Factor of Improvement* as follows:

$$\text{Factor of Improvement} = \frac{\text{Duration of a Baseline}}{\text{Duration of ECSched}} \tag{8}$$

Factor of Improvement greater than 1 means ECSched performs better, and vice versa.

Additionally, we use *Average Resource Utilization* over the cluster to measure resource efficiency during experiments.

Finally, to evaluate scheduling overhead, we compute *Algorithm Runtime* of schedulers in different scenarios.

5.2. Comparison of container performance

We first compare the container performance with baseline schemes to handle 2500 container requests on both clusters. Fig. 8 shows the results on the cluster with homogeneous machines. Overall, ECSched speeds up 83% to 86% of the containers and only slows down 8% to 12% of the containers. The reason is that ECSched schedules a batch of requests at the same time to find a more compact placement for the overall performance, which may hurt a small part of the requests due to the contention. We observe that ECSched-dp slightly outperforms ECSched-ml in the evaluation, as dot-product heuristic is that it takes into account not only the resource demands of containers but also how well its demands align with the resource capacities of machines. Compared to the scheduler of Kubernetes, ECSched-dp speeds up containers by 1.32 × at the median and 1.68 × at the 80th percentile. Compared to the scheduler of Swarm, ECSched-dp speeds up containers by 1.23 × at the median and 1.57 × at the 80th percentile.

Fig. 9 shows the results on the cluster with heterogeneous machines. In this cluster, ECSched speeds up 79% to 81% of the containers and slows down 10% to 15% of the containers. Similar to the homogeneous cluster, ECSched-dp performs better than ECSched-ml. Compared to the scheduler of Kubernetes, ECSched-dp speeds up containers by 1.20 × at the median and 1.50 × at the 80th percentile. Compared to the scheduler of Swarm, ECSched-dp speeds up containers by 1.28 × at the median and 1.63 × at the 80th percentile.

Fig. 10 shows the Cumulative Distribution Functions (CDFs) of container completion times for different schedulers. Relative to the baselines, the performance of containers is improved in both clusters. ECSched lowers the average container completion time by up to 1.3 × throughout the experiments. The improvements accrue from the increase in the number of simultaneously running containers on the cluster (less waiting time in the queue), as ECSched takes a batch of concurrent requests into consideration to make placement decisions.

Table 2
Average resource utilization on the cluster with homogeneous machines.

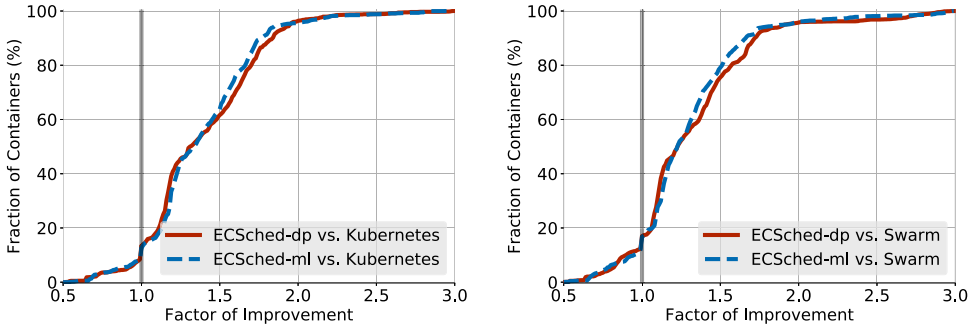
Scheduler	CPU utilization	Memory utilization
ECSched-dp	**76.57**%	**67.03**%
ECSched-ml	76.10%	66.61%
Kubernetes	71.21%	62.33%
Swarm	71.88%	62.92%

Table 3
Average resource utilization on the cluster with heterogeneous machines.

Scheduler	CPU utilization	Memory utilization
ECSched-dp	**79.81**%	**69.86**%
ECSched-ml	79.22%	69.34%
Kubernetes	75.53%	66.11%
Swarm	75.18%	65.80%

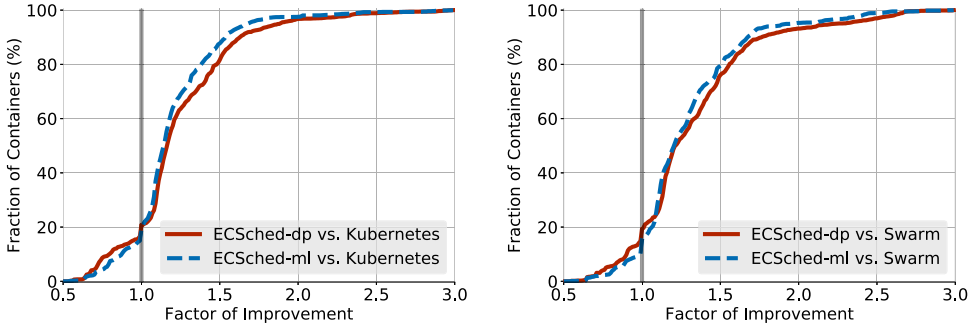
5.3. Comparison of resource efficiency

Next, we compare the average resource utilization over the cluster to evaluate the resource efficiency of different schedulers. During the experiments, we monitor the resource utilization across the cluster in every second. The resource utilization here is the ratio of the utilized resources to the total resources in the cluster. Tables 2 and 3 show the average resource utilization throughout the entire experiment. We observe that ECSched sustains higher resource utilization than the baselines. Consistent with the container performance, ECSched-dp achieves the highest average resource utilization. For the cluster with homogeneous machines, ECSched-dp increases resource utilization by 4.1% to 5.3% compared to the two baselines. For the cluster with heterogeneous machines, ECSched-dp increases resource utilization by 3.7% to 4.6% compared to the two baselines. In order to better



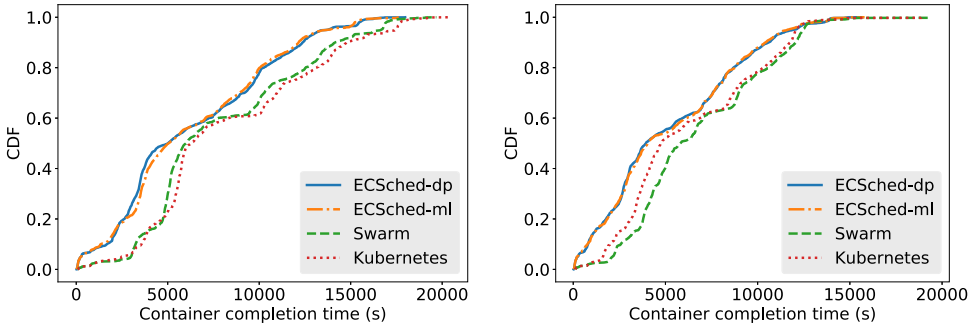
(a) Comparing with Kubernetes (b) Comparing with Swarm

Fig. 8. CDFs of *Factor of Improvement* on the cluster with homogeneous machines.



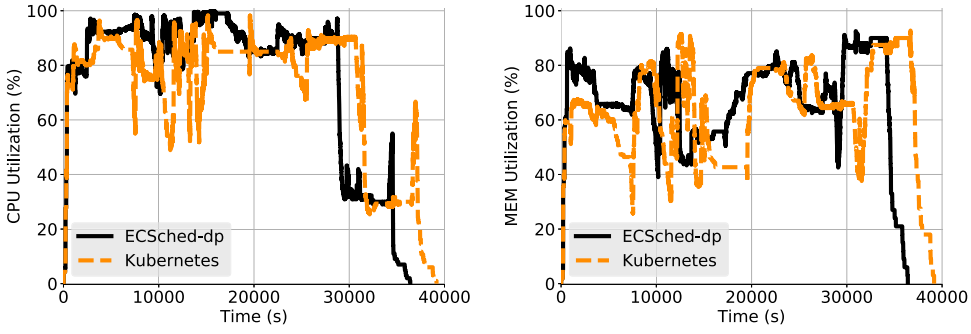
(a) Comparing with Kubernetes (b) Comparing with Swarm

Fig. 9. CDFs of *Factor of Improvement* on the cluster with heterogeneous machines.



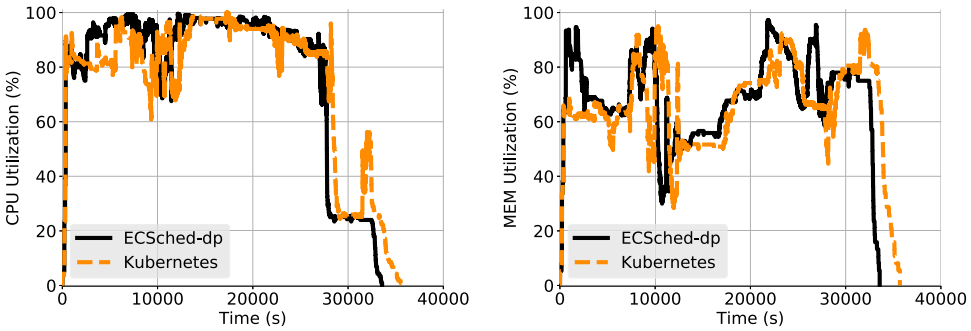
(a) Homogeneous Cluster (b) Heterogeneous Cluster

Fig. 10. CDFs of container completion times for different schedulers.



(a) CPU utilization (b) Memory utilization

Fig. 11. Comparing the resource utilization on the cluster with homogeneous machines.



(a) CPU utilization (b) Memory utilization

Fig. 12. Comparing the resource utilization on the cluster with heterogeneous machines.

understand the resource efficiency, we choose to plot the exact resource utilization of ECSched-dp and Kubernetes during the experiment.

Figs. 11 and 12 show the details of ECSched-dp and Kubernetes for both clusters. We observe that the requests we yielded are more CPU intensive. The CPU resources are highly competitive, and the utilization of CPU remains high during the experiments. Nevertheless, ECSched-dp still achieves higher resource utilization than Kubernetes in the peak hours. These improvements are because ECSched leverages the MCFP algorithm to find a better placement solution for the concurrent container requests, which can lead to less resource fragmentation of machines. Overall, it demonstrates that the ECSched outperforms existing container schedulers in terms of resource efficiency on different cluster.

5.4. Impact of concurrent scheduling

Compared to state-of-the-art schedulers, scheduling a set of concurrent requests at the same time is an innovative advantage of ECSched. As described earlier, ECSched would fetch a certain number of container requests from the queue system to construct a flow network for scheduling. For the above experiments, we set the maximum number that ECSched can fetch to 100. Thus, ECSched can schedule up to 100 container requests at a time. In this section, we configure the ECSched with the maximum fetch number of 1, 10, 50 and 100 to evaluate the container performance in different configurations. In order to understand how much influence does it have on the container completion time, we compare the configuration with the number of 10, 50, 100 to the configuration with the number of 1 in this experiment.

Fig. 13 shows the results of *Factor of Improvement* on the cluster with heterogeneous machines. We observe that along with the increase of maximum fetch number, the improvement of the container performance also increases. The impact to these two heuristics is quite similar in the experiment. Compared to the configuration with maximum fetch number of 1, ECSched lowers the average container completion time by 1.08 $\times$ with the number of 10, 1.21 $\times$ with the number of 50, and 1.30 $\times$ with the number of 100. Consequently, it demonstrates that ECSched can efficiently handle concurrent container requests, and significantly benefits from concurrent scheduling.

5.5. Overhead evaluation

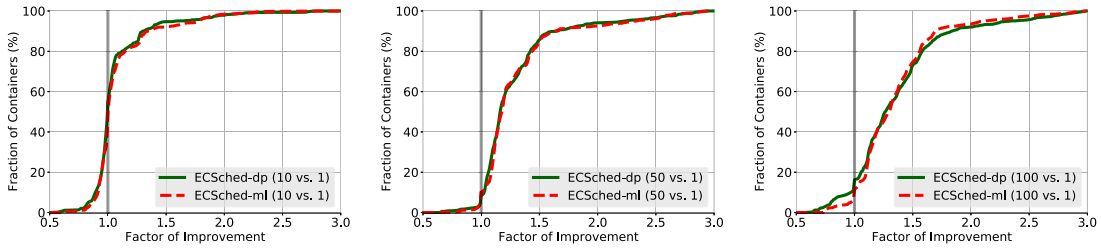
As we model the scheduling problem as a MCFP, the scheduling algorithm in our scheduler is more complex than existing schedulers. It would cause the overhead of ECSched to be higher than the others. To estimate the scheduling overhead, we perform large-scale simulations to measure the algorithm runtime of different schedulers. We consider two cluster sizes in the simulation: 1000-machine cluster and 5000-machine cluster (largest cluster which Kubernetes can support currently). In order to make the simulated cluster more heterogeneous, the configuration of each machine is chosen uniformly at random from 4 types of VM instances of ExoGENI experimental environment. As hundreds of requests need to be processed per second in peak hours according to the Google cluster trace [14], we choose to submit 100, 200 and 300 concurrent container requests to the scheduler for testing at the same time. Accordingly, we configure the ECSched with maximum fetch number of 100, 200 and 300. In order to fairly compare the algorithm runtime, we also implement the scheduling algorithm of Kubernetes and Swarm in Python, which is the same with ECSched. We conduct this experiment on a dedicated server with Intel Xeon E5-2630 2.4 GHz CPU and 64 GB memory.

Fig. 14 shows the results of the average algorithm runtime which we repeated one hundred times. We observe that the algorithm runtime of ECSched is highest while Swarm is lowest. The algorithm of Swarm is a simple greedy search to place requested containers on the most loaded machines. Compared to Swarm, the algorithm of Kubernetes is a bit complex, which has multiple predicated policies and priorities policies to filter and score machines. Obviously, our algorithm is the most complicated one, and has higher overhead. Nevertheless, ECSched can respond in sub-second time when the number of concurrent requests is less than 100. When processing 300 containers concurrently, the ECSched responds in about 1.8 s for 1000-machine cluster and about 3.4 s for 5000-machine cluster. Actually, compared to the average duration (740 s in our experiments) of the containers in the cluster [14], this overhead is relatively small and acceptable. Considering the container performance we discussed in previous section, there thus is a tradeoff between the quality and the overhead when scheduling containers. Users can dynamically adjust the maximum fetch number of ECSched to seek a best tradeoff for their workloads. Overall, we believe that ECSched is effective and usable in practice.

6. Related work

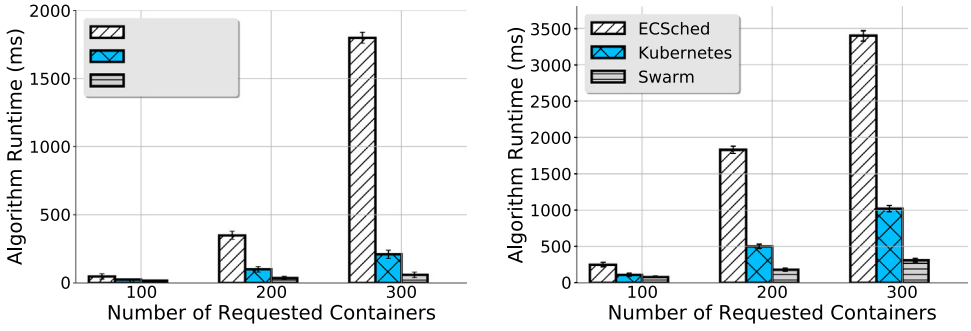
The problem investigated in this chapter – container scheduling on heterogeneous clusters with multi-resource constraints – is related to a variety of research topics as follows.

Bin packing The problem of VM placement or consolidation which is similar to our problem is often formulated as vector bin packing problem, and various heuristics have been proposed for this problem [17,36–39]. Stillwell et al. [40] studied the resource allocation problem in shared hosting platforms for static workloads with machines which provide multiple types of resources. They proposed several kinds of vector bin packing algorithms and



(a) 10 vs. 1 (b) 50 vs. 1 (c) 100 vs. 1

Fig. 13. Comparing the container performance in the configurations with maximum fetch number of 1, 10, 50 and 100.



(a) 1000-machine cluster (b) 5000-machine cluster

Fig. 14. Comparing algorithm runtime with large-scale simulations.

evaluated them over a wide range of simulations. They concluded that the first fit decreasing (FFD) heuristic that reasons on the sum of the resource demands of the tasks are the most effective. Furthermore, Panigrahy et al. [29] systematically studied variants of the FFD algorithm that have been proposed for VM placement problems, and presented a different generalization of the classical FFD heuristic. In their empirical evaluations, it showed that the Dot-Product heuristic often outperforms FFD-based heuristics. These studies focus on the packing problem with identical bins (i.e., machines), and consider each request independently. Different from them, we tackle the problem of scheduling concurrent requests on heterogeneous cluster and consider the requirements of container affinity and machine affinity at the same time.

Metaheuristics In recent years, many metaheuristic techniques have become prevalent for the approximate solution of multi-objective optimization problems [41–44]. Mi et al. [45] proposed a genetic algorithm based approach, namely GABA, to adaptively self-reconfigure the virtual machines on large-scale clusters which is composed of heterogeneous machines. Xu et al. [41] presented a modified genetic algorithm to find global optimal solutions of virtual machine placement problem. Their approach leverages a fuzzy-logic based evaluation for incorporating different objectives. Gao et al. [42] proposed a multi-objective ant colony system algorithm to find a set of Pareto solutions for the virtual machine placement problem. However, these approaches often take minutes or even hours, particularly for large-scale clusters, to generate a placement solution, which would face difficulties for a online response. In contrast, we formulate the scheduling problem as a minimum cost flow problem, which can be solved in a polynomial time.

Schedulers Many cluster schedulers have been proposed for different purposes [46–49]. Sparrow [12] and Tarcil [19] are distributed schedulers developed for clusters that achieve a high throughput for short tasks. Quincy [28], a fair cluster scheduler, models the fair scheduling problem as a minimum cost flow problem to schedule jobs into slots. In their flow network, the edge capacities and weights encode the demands of starvation-freedom, data locality, and fairness. And then, they used a standard solver to compute the optimal solution based on a cost model. In contrast, we focus on the concurrent container requests with multi-resource demands and implement an appropriate MCFP algorithm for our problem. Firmament [27], a centralized scheduler that can scale to over ten thousand machines at sub-second placement latency via a min-cost max-flow (MCMF) optimization. They proposed some problem-specific optimizations for MCMF algorithms and can achieve low latency by solving the problem incrementally. Xiao et al. [50] employed a minimum cost flow network model on the layered video streaming to achieve high throughput. Cho et al. [51] leveraged the flow network model to solve the multiprocessor real-time task scheduling problems. However, they all cannot handle the requests with multi-resource demands. Different from them, ECSched shows that how to encode multi-resource constraints and affinity requirements in the minimum cost flow problem.

7. Conclusion

In this paper, we have presented ECSched, an efficient solution for handling concurrent container requests on heterogeneous clusters with multi-resource constraints. ECSched is a graph-based scheduler, which can leverage the minimum-cost flow model to effectively process concurrent container requests. In the testbed experiments, we demonstrate that ECSched can achieve better scheduling quality than state-of-the-art container schedulers, which can lower the average container completion time by up to 1.3 × and noticeably improve resource utilization. The large-scale simulations show that there is relatively small overhead of ECSched, but it is acceptable in practice. In the future work, we will investigate problem-specific optimizations

to improve our implementation, and consider container dependencies and resource dynamics in the scheduler to adopt more sophisticated situations.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research has received funding from the European Union's Horizon 2020 research and innovation program under grant agreements No. 654182 (ENVRIPLUS project), No. 676247 (VRE4EIC project), No. 643963 (SWITCH project), No. 824068 (ENVRIFAIR project) and No. 825134 (ARTICONF project). The research is also funded by Chinese Scholarship Council.

References

[1] Amazon elastic container service, https://aws.amazon.com/ecs/.
[2] Azure container service, https://azure.microsoft.com/product-categories/containers/.
[3] Docker swarm, https://docs.docker.com/engine/swarm/.
[4] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A.D. Joseph, R.H. Katz, S. Shenker, I. Stoica, Mesos: A platform for fine-grained resource sharing in the data center, 11 (2011) (2011) 22.
[5] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, J. Wilkes, Borg, omega, and kubernetes, Commun. ACM 59 (5) (2016) 50–57.
[6] Amazon web services, https://aws.amazon.com/.
[7] Microsoft azure, https://azure.microsoft.com/.
[8] Y. Hu, J. Wang, H. Zhou, P. Martin, A. Taal, C. de Laat, Z. Zhao, Deadline-aware deployment for time critical applications in clouds, in: European Conference on Parallel Processing, Springer, 2017, pp. 345–357.
[9] J. Wang, A. Taal, P. Martin, Y. Hu, H. Zhou, J. Pang, C. de Laat, Z. Zhao, Planning virtual infrastructures for time critical applications with multiple deadline constraints, Future Gener. Comput. Syst. 75 (2017) 365–375.
[10] Z. Zhao, A. Taal, A. Jones, I. Taylor, V. Stankovski, I.G. Vega, F.J. Hidalgo, G. Suciu, A. Ulisses, P. Ferreira, et al., A software workbench for interactive, time critical and highly self-adaptive cloud applications (switch), in: Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on, IEEE, 2015, pp. 1181–1184.
[11] H. Zhou, Y. Hu, J. Wang, P. Martin, C. De Laat, Z. Zhao, Fast and dynamic resource provisioning for quality critical cloud applications, in: Real-Time Distributed Computing (ISORC), 2016 IEEE 19th International Symposium on, IEEE, 2016, pp. 92–99.
[12] K. Ousterhout, P. Wendell, M. Zaharia, I. Stoica, Sparrow: distributed, low latency scheduling, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, ACM, 2013, pp. 69–84.
[13] S. Taherizadeh, A.C. Jones, I. Taylor, Z. Zhao, V. Stankovski, Monitoring self-adaptive applications within edge computing frameworks: a state-of-the-art review, J. Syst. Softw. 136 (2018) 19–38.
[14] C. Reiss, A. Tumanov, G.R. Ganger, R.H. Katz, M.A. Kozuch, Heterogeneity and dynamicity of clouds at scale: Google trace analysis, in: Proceedings of the Third ACM Symposium on Cloud Computing, ACM, 2012, p. 7.
[15] Mesosphere marathon, https://mesosphere.com/.
[16] Google kubernetes, https://kubernetes.io/.
[17] A. Lodi, S. Martello, D. Vigo, Recent advances on two-dimensional bin packing problems, Discrete Appl. Math. 123 (1) (2002) 379–396.
[18] Y. Ajiro, A. Tanaka, Improving packing algorithms for server consolidation, in: Int. CMG Conference, Vol. 253, 2007.
[19] C. Delimitrou, D. Sanchez, C. Kozyrakis, Tarcil: reconciling scheduling speed and quality in large shared clusters, in: Proceedings of the Sixth ACM Symposium on Cloud Computing, ACM, 2015, pp. 97–110.
[20] D. Breitgand, A. Marashini, J. Tordsson, Policy-driven service placement optimization in federated clouds, IBM Research Division, Tech. Rep 9 (2011) 11-15.
[21] J. Tordsson, R.S. Montero, R. Moreno-Vozmediano, I.M. Llorente, Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers, Future Gener. Comput. Syst. 28 (2) (2012) 358–367.
[22] W. Shi, B. Hong, Towards profitable virtual machine placement in the data center, in: 2011 Fourth IEEE International Conference on Utility and Cloud Computing, IEEE, 2011, pp. 138–145.
[23] A. Schrijver, Theory of Linear and Integer Programming, John Wiley & Sons, 1998.
[24] R.K. Ahuja, Network Flows: Theory, Algorithms, and Applications, Pearson Education, 2017.
[25] A.V. Goldberg, R.E. Tarjan, Finding minimum-cost circulations by successive approximation, Math. Oper. Res. 15 (3) (1990) 430–466.
[26] A.V. Goldberg, M. Kharitonov, On implementing scaling push-relabel algorithms for the minimum-cost flow problem, in: Network Flows and Matching, 1991, pp. 157–198.
[27] I. Gog, M. Schwarzkopf, A. Gleave, R.N. Watson, S. Hand, Firmament: Fast, centralized cluster scheduling at scale, in: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), 2016, pp. 99–115.
[28] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, A. Goldberg, Quincy: fair scheduling for distributed computing clusters, in: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, ACM, 2009, pp. 261–276.
[29] R. Panigrahy, K. Talwar, L. Uyeda, U. Wieder, Heuristics for vector bin packing, research. microsoft.com (2011).
[30] M. Klein, A primal method for minimal cost flows with applications to the assignment and transportation problems, Manage. Sci. 14 (3) (1967) 205–220.
[31] J. Edmonds, R.M. Karp, Theoretical improvements in algorithmic efficiency for network flow problems, J. ACM 19 (2) (1972) 248–264.
[32] A.V. Goldberg, An efficient implementation of a scaling minimum-cost flow algorithm, J. Algorithms 22 (1) (1997) 1–29.
[33] E.A. Dinic, Algorithm for solution of a problem of maximum flow in networks with power estimation, in: Soviet Math. Doklady, Vol. 11, 1970, pp. 1277–1280.
[34] J.B. Orlin, A faster strongly polynomial minimum cost flow algorithm, Oper. Res. 41 (2) (1993) 338–350.
[35] I. Baldin, J. Chase, Y. Xin, A. Mandal, P. Ruth, C. Castillo, V. Orlikowski, C. Heermann, J. Mills, Exogeni: A multi-domain infrastructure-as-a-service testbed, in: The GENI Book, Springer, 2016, pp. 279–315.
[36] S. Lee, R. Panigrahy, V. Prabhakaran, V. Ramasubramanian, K. Talwar, L. Uyeda, U. Wieder, Validating heuristics for virtual machines consolidation, Microsoft Research, MSR-TR-2011-9 (2011) 1-14.
[37] S. Rampersaud, D. Grosu, Sharing-aware online virtual machine packing in heterogeneous resource clouds, IEEE Trans. Parallel Distrib. Syst. 28 (7) (2017) 2046–2059.
[38] M. Gabay, S. Zaourar, Vector bin packing with heterogeneous bins: application to the machine reassignment problem, Ann. Oper. Res. 242 (1) (2016) 161–194.
[39] W. Leinberger, G. Karypis, V. Kumar, Multi-capacity bin packing algorithms with applications to job scheduling under multiple constraints, in: Parallel Processing, 1999. Proceedings. 1999 International Conference on, IEEE, 1999, pp. 404–412.
[40] M. Stillwell, D. Schanzenbach, F. Vivien, H. Casanova, Resource allocation algorithms for virtualized service hosting platforms, J. Parallel Distrib. Comput. 70 (9) (2010) 962–974.
[41] J. Xu, J.A. Fortes, Multi-objective virtual machine placement in virtualized data center environments, in: Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing, IEEE Computer Society, 2010, pp. 179–188.
[42] Y. Gao, H. Guan, Z. Qi, Y. Hou, L. Liu, A multi-objective ant colony system algorithm for virtual machine placement in cloud computing, J. Comput. System Sci. 79 (8) (2013) 1230–1242.
[43] M.H. Ferdaus, M. Murshed, R.N. Calheiros, R. Buyya, Virtual machine consolidation in cloud data centers using aco metaheuristic, in: European Conference on Parallel Processing, Springer, 2014, pp. 306–317.
[44] Z.Á. Mann, Allocation of virtual machines in cloud data centers-a survey of problem models and optimization algorithms, ACM Comput. Surv. (CSUR) 48 (1) (2015) 11.
[45] H. Mi, H. Wang, G. Yin, Y. Zhou, D. Shi, L. Yuan, Online self-reconfiguration with performance guarantee for energy-efficient large-scale cloud computing data centers, in: Services Computing (SCC), 2010 IEEE International Conference on, IEEE, 2010, pp. 514–521.
[46] Y. Hu, H. Zhou, C. de Laat, Z. Zhao, Ecsched: Efficient container scheduling on heterogeneous clusters, in: European Conference on Parallel Processing, Springer, 2018, pp. 365–377.
[47] R. Grandl, S. Kandula, S. Rao, A. Akella, J. Kulkarni, G: Packing and dependency-aware scheduling for data-parallel clusters, in: Proceedings of OSDI'16: 12th USENIX Symposium on Operating Systems Design and Implementation, 2016, p. 81.
[48] R. Grandl, M. Chowdhury, A. Akella, G. Ananthanarayanan, Altruistic scheduling in multi-resource clusters, in: OSDI, 2016, pp. 65–80.
[49] S.A. Jyothi, C. Curino, I. Menache, S.M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, et al., Morpheus: Towards automated slos for enterprise clusters, in: OSDI, 2016, pp. 117–134.

[50] X. Xiao, Y. Shi, Y. Gao, Q. Zhang, Layerp2p: A new data scheduling approach for layered streaming in heterogeneous networks, in: IEEE INFOCOM 2009, IEEE, 2009, pp. 603–611.
[51] H. Cho, A. Easwaran, Flow network models for online scheduling real-time tasks on multiprocessors, arXiv preprint arXiv:1810.08342 (2018).

Yang Hu received the BSc degree and the MSc degree in computer science focusing on network of cloud computing from the National University of Defense Technology (NUDT), China. He is currently a Ph.D. candidate at the University of Amsterdam (UvA), Netherlands. His research interests include resource management, network, cloud computing and deep reinforcement learning.

Huan Zhou received the Master's degree in computer science from the National University of Defense Technology (NUDT), China. He is currently a Ph.D. student of University of Amsterdam (UvA), Netherlands. His research focuses on cloud computing and network communication.

Cees de Laat chairs the System and Network Engineering (SNE) laboratory in the Informatics Institute of the Faculty of Science at University of Amsterdam. The SNE lab conducts research on leading-edge computer systems of all scales, ranging from global-scale systems and networks to embedded devices. Across these multiple scales our particular interest is on extra-functional properties of systems, such as performance, programmability, productivity, security, trust, sustainability and, last but not least, the societal impact of emerging systems-related technologies. Prof. de Laat serves on the Lawrence Berkeley Laboratory Policy Board for ESnet, is co-founder of the Global Lambda Integrated Facility (GLIF), founder of GRIDforum.nl and founding member of CineGrid.org. His group is/was part of a.o. EU projects GN4-2, SWITCH, CYCLONE, ENVRIplus and ENVRI, Geysers, NOVI, NEXTGRID, EGEE, and national projects DL4LD, SARNET, COMMIT, GIGAport and VL-e. He is a member of the Advisory Board Internet Society Netherlands and Scientific technical advisory board of SURF Netherlands.

Zhiming Zhao obtained his Ph.D. in computer science in 2004 from University of Amsterdam (UvA). He is currently a senior researcher in the System and Network Engineering group at UvA. He coordinates research effort on quality critical systems on programmable infrastructures in the context of European H2020 projects of SWTICH and ENVRIPLUS. His research interests include software defined networking, workflow management systems, multi agent system and big data research infrastructures.