ECSched: Efficient Container Scheduling on Heterogeneous Clusters

Yang Hu[1,2(✉)], Huan Zhou[1], Cees de Laat[1], and Zhiming Zhao[1]

[1] University of Amsterdam, Amsterdam, The Netherlands
{Y.Hu,H.Zhou,delaat,Z.Zhao}@uva.nl
[2] National University of Defense Technology, Changsha, China

Abstract. Operating system (OS) containers are becoming increasingly popular in cloud computing for improving productivity and code portability. However, container scheduling on large heterogeneous cluster is quite challenging. Recent research on cluster scheduling focuses either on scheduling speed to quickly assign resources, or on scheduling quality to improve application performance and cluster utilization. In this paper, we propose ECSched, an efficient container scheduler that can make high-quality and fast placement decisions for concurrent deployment requests on heterogeneous clusters. We map the scheduling problem to a graphic data structure and model it as minimum cost flow problem (MCFP). We implement ECSched based on our cost model, which encodes the deployment requirements of requested containers. In the evaluation, we show that ECSched exceeds the placement quality of existing container schedulers with relatively small overheads, while providing 1.1× better resource efficiency and 1.3× lower average container completion time.

1 Introduction

Operating system (OS) containers are becoming increasingly popular in cloud computing for improving productivity and code portability. Major cloud providers have recently announced container-based cloud services to cater for this popularity [1,4]. Meanwhile, container orchestration platforms, such as Docker Swarm [2], Mesosphere Marathon [12], and Google Kubernetes [8], 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, Microsoft Azure [1,4]) is based on Virtual Machines (VMs). Compared with VM-based infrastructure, container-based infrastructure (1) can be deployed on both physical and virtual machines, and the highly diverse configuration of VMs makes the clustered machines more heterogeneous; (2) can provide fine-grained resource allocation based on operating-system-level virtualization techniques, which is much more flexible than predefined VM types in VM-based infrastructure; and (3) can support users specifying affinities among containers (e.g., Affinity in Kubernetes) for a distributed application, which facilitates the coordination of containers.

© Springer International Publishing AG, part of Springer Nature 2018
M. Aldinucci et al. (Eds.): Euro-Par 2018, LNCS 11014, pp. 365–377, 2018.
https://doi.org/10.1007/978-3-319-96983-1_26

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

1. Multi-resource demands from each container are often specified as a combination of constraints of CPU, memory, network, etc., which have to be considered with the diverse capacity and capability of the underlying heterogeneous cluster.
2. Containers of a distributed application often have strong affinities with other containers (due to frequent data communication) or specific machines (due to data locality). Placing containers on the appropriate node can significantly reduce the latency of container communication and the volume of data transferred. Thus, affinity also has to take into account in the deployment scheduler.
3. Scheduling overheads in large clusters are relatively high, which may hurt the performance of quality critical applications [13,24,27,28], especially for very short jobs like real-time analytics [19,23]. Moreover, the scheduling algorithm is frequently invoked during the execution of an application when scaling out or recovering from failure, which often has critical time constraints. Thus, the scheduler should be fast to scale to large clusters.

During the past years, container orchestration and scheduling have attracted quite a lot research attention. In the containers orchestration platforms, such as Swarm [2] and Kubernetes [3], they typically adopt queue-based scheduler which process one container at a time (process one pod at a time in Kubernetes). The requested container first waits in a queue until the scheduler fetches it and performs the scheduling algorithm. Regarding the scheduling algorithms to the queue-based scheduler, variants of heuristic packing algorithms, such as Best-Fit Decreasing (BFD) and First-Fit Decreasing (FFD) [6,16], are often used to achieve practical solutions.

Container-by-container scheduling has the advantage of being suitable for concurrent, parallel decisions in distributed scheduler [9,19]. On the contrary, scheduling one container at a time also has a crucial disadvantage: the scheduler makes a decision early for a container and restricts its choices for the waiting containers, where it is difficult to make a high-quality placement. To schedule a batch of tasks concurrently, the most common method is using meta-heuristic algorithms [17,25], which consider the scheduling problem as a whole and find an optimal solution offline. However, they often face difficulties online for a real-time response to dynamic requests [26].

In this paper, we propose ECSched, an efficient container scheduler that can make high-quality and fast placement decisions for concurrent deployment requests on heterogeneous clusters. We map the scheduling problem to a graphic data structure and model it as minimum cost flow problem (MCFP). In the model, edge weights and capacities encode the container demands of multiple resources and container/machine affinities. We implement ECSched based on classical MCFP algorithms and problem-specific optimizations, which can compute the optimal solution online according to our cost model. We evaluate

ECSched in a small-scale cluster and large-scale simulations. In the evaluation, we show that ECSched exceeds the placement quality of state-of-the-art container schedulers with relatively small overheads, while providing 1.1× better resource efficiency and 1.3× lower average container completion time.

2 Problem Formulation

In this section, we first formulate the containers scheduling problem with networked heterogeneous machines in the cluster. Then, we analyze different requirements for container deployment.

2.1 Model Description

In container-based infrastructure, the cluster is typically composed of a set of networked heterogeneous machines $\{\mathbb{M} = \{m_1, m_2, ..., m_M\}$ where $M = |\mathbb{M}|$ is the number of machines. We consider R types of resources $\mathbb{R} = \{r_1, r_2, ..., r_R\}$ (e.g., CPU, memory, or network bandwidth) in each machine. For machine m_i, let $\overrightarrow{V_i} = (V_i^1, V_i^2, ..., V_i^R)$ 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 model the deployment request in the scheduler as a set of containers $\mathbb{C} = \{c_1, c_2, ..., c_N\}$ that are to be deployed on M machines, and $N = |\mathbb{C}|$ is the number of containers. For container c_i, let $\overrightarrow{D_i} = (D_i^1, D_i^2, ..., D_i^R)$ 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 matrix $\mathbb{CA} = [CA_{ij}]_{N\times N}$ denote the container affinity. If $CA_{ij} = 1$, it means that the container c_i has a affinity with container c_j. Let matrix $\mathbb{MA} = [MA_{ij}]_{N\times M}$ denote the machine affinity. If $MA_{ij} = 1$, it means that the container c_i has a 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 paper. Let 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, otherwise X_{ij} is 0.

2.2 Deployment Requirements

By analyzing the features of container-based infrastructure, we desire a placement solution that satisfies the following objectives.

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, which 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 the same machine should not exceed its capacity.

$$\sum_{c_i \in \mathbb{C}} X_{ij} D_i^k \leq V_j^k \quad \forall m_j \in \mathbb{M},\ \forall r_k \in \mathbb{R} \tag{1}$$

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 data locality. As distributed applications, especially data-intensive applications, transfer data frequently, the network performance 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 take advantage of this information to adjust container placement. The intuitive and effective solution is to co-locate the containers which have container affinities on the same machine,

$$\sum_{m_k \in \mathbb{M}} X_{ik} X_{jk} \geq CA_{ij} \quad \forall c_i, \forall c_j \in \mathbb{C} \tag{2}$$

and place the container on the affinity machine.

$$X_{ik} \geq MA_i^k \quad \forall c_i \in \mathbb{C}, \forall m_k \in \mathbb{M} \tag{3}$$

With these objectives, the challenge for a scheduler is how to make placement decisions fast to improve cluster resource utilization while maintaining container performance.

3 ECSched Approach

As existing queue-based schedulers process one container at a time, the entire workload cannot be considered in the decision-making phase. Consequently, it is hard for the scheduler to make a high-quality placement. In this paper, we choose a graph-based approach to achieve concurrent containers scheduling and model the scheduling problem as minimum cost flow problem (MCFP) [5]. In the rest of this section, we describe how to construct the graph of MCFP to solve the container scheduling problem and what MCFP algorithms to use.

3.1 Minimum Cost Flow Problem

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 capacity $c_{u,v} > 0$

and cost $a_{u,v}$. The edge $e_{u,v}$ can be assigned a flow $f_{u,v} \geq 0$, and the cost of sending this flow is $f_{u,v} \cdot a_{u,v}$. The problem requires an amount of flow K to be sent from source s to sink t, and the goal is to minimize the total cost of the flow over all edges:

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

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

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

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

3.2 Flow Network Structure

To map the container scheduling problem to MCFP, we represent it using a specific flow network. Figure 1 shows an example of the flow network, in which we only annotate the capacity on edges. This graph corresponds to an instantaneous status of the container cluster, encoding a set of requested containers and clustered machines. The overall structure of the graph can be described as follows.

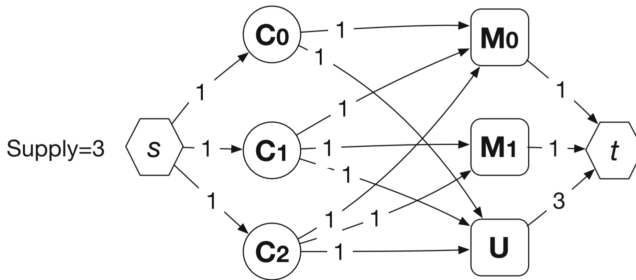


Fig. 1. An example of the flow network

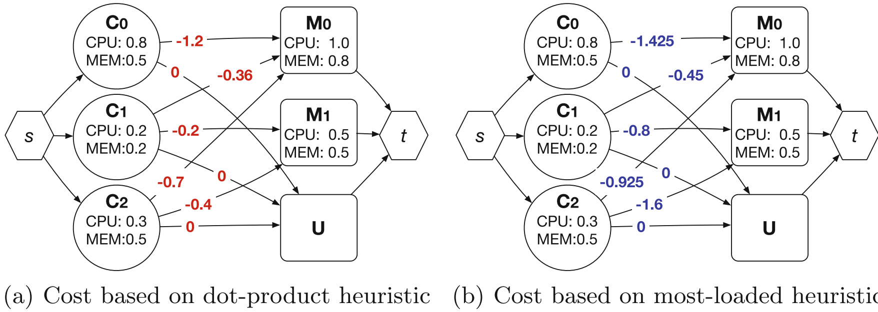
- **Source Node:** The source node s on the left hand with a supply K, which represents how many containers can be scheduled at a time in our context. By default, the supply is set to the total number of requested containers in the scheduler ($K = N$).
- **Container Node:** Each requested container is represented as node C_i in the graph, and has an edge from source node s with capacity 1.
- **Machine Node:** Each clustered machine is represented as node M_i in the graph, and has an edge from the container node with capacity 1 if the machine is eligible to place the container.

- **Unscheduled Node:** Inspired by the work [14], we add a new node, called unscheduled node U. All container nodes have an outgoing edge to node U with capacity 1.
- **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 sink with capacity 1, and the unscheduled node has an edge to sink with capacity N.

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

3.3 Encoding Deployment Requirements

As the goal of the MCFP problem is to minimize the total cost of the flow over all edges, we can flexibly assign the costs on the edges to make the MCFP algorithms return a solution which we desire for the container placement. Considering two deployment requirements from containers, we propose following methods to encode them on edges.



(a) Cost based on dot-product heuristic (b) Cost based on most-loaded heuristic

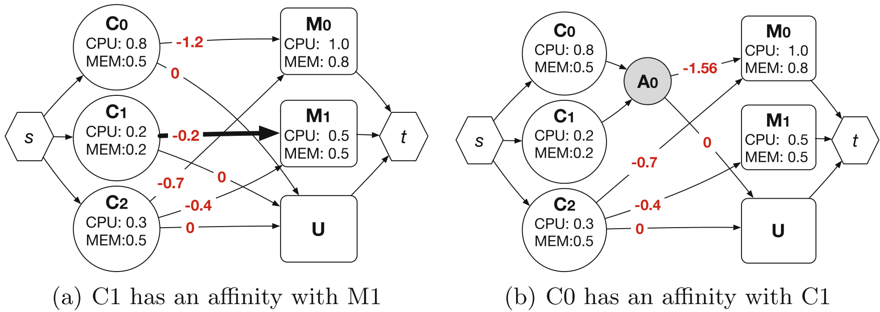
Fig. 2. An example for encoding the multi-resource requirements

Multi-resource Guarantee. 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 corresponding maximum capacity independently. After normalization, the scheduler checks which machines 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 capacity 1. The challenge here is how to assign the costs on the edges to differentiate the quality of different placements. We introduce two strategies which are inspired

by vector bin packing algorithms [20]: dot-product heuristic and most-loaded heuristic.

In dot-product heuristic, dot product between the demand vector of container c_i and the capacity vector of machine m_j is defined as $dp_{ij} = \sum_{r_k \in \mathbb{R}} D_i^k V_j^k$. The higher dp_{ij} is, the better the placement is. 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. Nevertheless, the cost on the edge between them is assigned to $-dp_{ij}$, because the lower the cost is, the better the flow is in MCFP. For the edge from container node to unscheduled node, the cost is 0 which is the highest. An example is shown in Fig. 2(a).

In most-loaded heuristic, the container tends to be placed on the most loaded machine. In our cost model, it is also based on a scalar value $ml_{ij} = \sum_{r_k \in \mathbb{R}} \frac{D_i^k}{V_j^k}$ between the container c_i and the machine m_j to prioritize the placement. The higher ml_{ij} is, the more loaded the machine is. Similarly, the cost on the edge is assigned to $-ml_{ij}$. An example is shown in Fig. 2(b).



(a) C1 has an affinity with M1 (b) C0 has an affinity with C1

Fig. 3. An example for encoding the affinity requirements (dot-product heuristic)

Affinity Awareness. 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 specific machine). Figure 3(a) 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 the edge to machine m_1 but no edge to machine m_0. Figure 3(b) shows an example with container affinity, where container c_0 and container c_1 have an affinity. In the flow network, we add a new node, called aggregator node A_i (A_0 in the example). Both container c_0 and container c_1 have an edge to aggregator node A_0. Hence, the scheduler can treat the two containers as one to handle container affinity.

3.4 MCFP Algorithms

After constructing the flow network, the scheduler will perform a MCFP algorithm to find the optimal placement solution with respect to the costs we have assigned. Known worst-case complexity bounds on the MCFP are $O(M \log(N)(M + N \log(N)))$ [18] and $O(NM \log(N) \log(NC))$ [11], Where N is the number nodes, M the number of edges and C the number of the largest edge capacity. In the container scheduling problem, it is the case as $M > N > C$. We currently implement the latter algorithm in our ECSched. However, MCFP algorithms have variable runtimes depending on the input graph. The comparison of different algorithms and the optimization of algorithms can be explored as future work. The design of ECSched is based on a heartbeat mechanism. On a heartbeat, ECSched fetches all the deployment requests to construct a flow network, and performs the MCFP algorithm to find a placement solution.

4 Evaluation

We implement ECSched with a container manager and the above MCFP algorithm in Python. In this section, we evaluate our ECSched on a 30-machine cluster in ExoGENI to compare the placement quality. To understand the overhead of ECSched, we do large-scale simulations using synthetic workloads.

4.1 Comparison of Placement Quality

Cluster. We create a container cluster with 30 virtual machines (VM) in ExoGENI [7] testbed. Considering the heterogeneity, we choose three types of VM configurations in our experiments. Thus, the container cluster is composed of 10 VMs of "XOMedium" type (1 core, 3 GB of memory), 10 VMs of "XOLarge" type (2 core, 6 GB of memory) and 10 VMs of "XOXLarge" type (4 core, 12 GB of memory). After normalization, the capacity vectors are: (CPU: 0.25, MEM: 0.25), (CPU: 0.5, MEM: 0.5), and (CPU: 1, MEM: 1).

Workloads. To test our prototype, we constructed container deployment requests based on the Google cluster trace [21], which provides data from a 12,500-machine cluster over a month-long period. As we chose to spend 5 hours at each experiment, we analyzed the trace of the first five hours. There are 83,241 tasks completed, and the average duration of the tasks is 764 s. Considering the scale of our testbed cluster, we randomly sample 8,300 tasks (10%) 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 6% probability according to the percentage of task constraints in the trace [21].

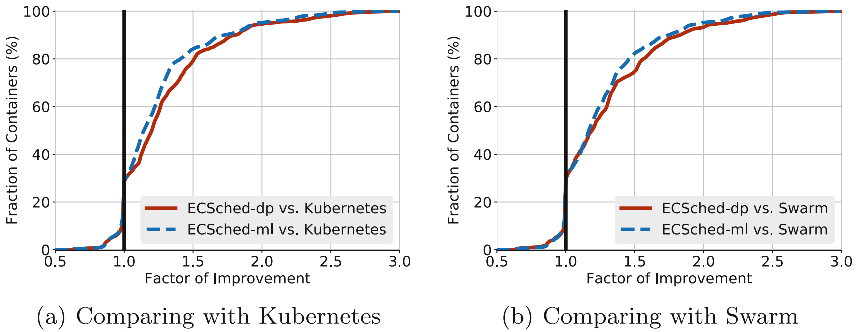
Baselines. We compare ECSched to state-of-the-art scheduling algorithms implemented in Google Kubernetes [3] and Docker Swarm [2]. Under multi-resource requirements, the default scheduler of Kubernetes tends to distribute

pods (smallest deployable units in Kubernetes) evenly across the cluster to balance the resources, while the scheduler of Swarm tends to place containers on the most loaded machines to improve resource utilization. Both are queue-based schedulers, which schedule one unit at a time.

Metrics. We consider two metrics: the average container completion time and average cluster resource utilization to compare the placement quality of different schedulers. The improvement of average container completion time is computed as:

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

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



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

Fig. 4. CDF of factors of improvement in average container completion time

Figure 4 compares the performance of ECSched with baseline schemes to handle 8,300 container requests on the cluster. We use two strategies in our scheduler to do the comparisons, where ECSched-dp is based on dot-product heuristic, and ECSched-ml is based on most-loaded heuristic. In the figure, the results show that for more than 68% of the containers, ECSched performs better than the alternatives, and only 10% of the containers slow down. For two different strategies, ECSched-dp performs better than ECSched-ml in our evaluation. To the scheduler of Kubernetes, ECSched-dp speeds up containers by 1.2× at the median, 1.28× at the 60th percentile, and 1.5× at the 80th percentile. To the scheduler of Swarm, ECSched-dp speeds up containers by 1.21× at the median, 1.3× at the 60th percentile, and 1.57× at the 80th percentile. Overall, ECSched improves over the alternatives by up to 1.3× on average. The improvements accrue from the increase in the number of simultaneously running containers (less waiting time in the queue), as ECSched takes entire workloads into consideration to make placement decisions.

To evaluate the resource efficiency, we make some changes to the workloads. All the container requests are submitted at the beginning in the experiment.

Table 1 shows the average cluster resource utilization of the experiment. Due to the better placement (cause less resource fragmentation), ECSched sustains higher cluster resource utilization than the baselines. Overall, ECSched provides 1.1× better resource efficiency. Consequently, it demonstrates that the ECSched approach can achieve higher quality placements for deploying containers on heterogeneous clusters.

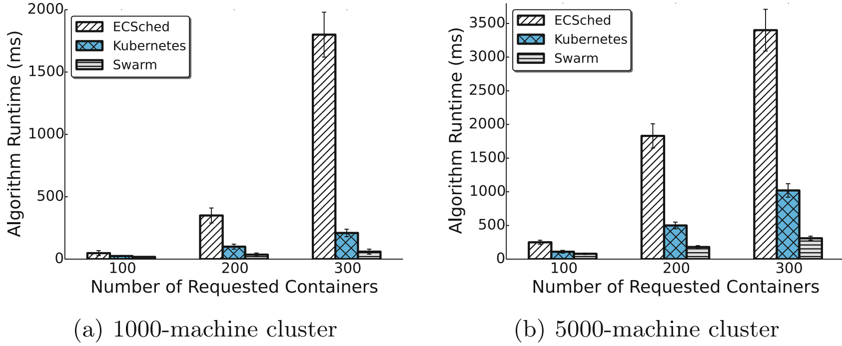
Table 1. Average cluster resource utilization in the experiment

Resource type	ECSched-dp	ECSched-ml	Kubernetes	Swarm
CPU	**76.57**%	75.80%	70.00%	69.98%
MEMORY	**76.71**%	75.93%	70.12%	70.10%

4.2 Overheads Evaluation

As we model the scheduling problem as a MCFP, the scheduling algorithm in our scheduler is more complex than existing schedulers. To estimate overheads, we simulate large-scale clusters to run our scheduling algorithm. We consider two cluster sizes: 1000-machine cluster and 5000-machine cluster (largest cluster which Kubernetes can support currently). The configuration of each machine is chosen uniformly at random from Amazon EC2 instances (19 kinds of general purpose instances) in order to make the simulated cluster more heterogeneous, and each machine is half loaded in the simulation. By analyzing the trace [21], the scheduler needs to make hundreds of task placement decisions per second in peak hours. Thus, we try to submit 100, 200 and 300 container deployment requests to the scheduler to evaluate the algorithm runtime. In order to fairly compare the algorithm runtime, we also implement the scheduling algorithm of Kubernetes and Swarm in Python. We conduct this experiment on a server with 48 cores and 128 GB memory.

Figure 5 shows the results of the experiment which we repeated ten times. We see that the algorithm runtime of ECSched is longest while Swarm is shortest. The algorithm of Swarm is a simple greedy search to place requested containers. Compared with Swarm, the algorithm of Kubernetes is complex, which has multiple predicated policies and priorities policies to filter and score machines. Obviously, our algorithm is the most complicated one. Nevertheless, ECSched can respond in sub-second time when the number of requested containers 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 (764 s in our experiments) of the containers in the cluster [21], the overhead is relatively small and acceptable. We believe that our scheduler is effective and usable in practice.



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

Fig. 5. Comparing algorithm runtime in large-scale simulation

5 Related Work

The problem investigated in this paper - container scheduling on heterogeneous clusters - 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 [15,16]. Mark Stillwell et al. [22] studied variants of FFD concluding that the algorithm that reasons on the sum of the resource needs of the tasks are the most effective. Panigrahy et al. [20] presented a generalization of the classical first fit decreasing (FFD) heuristic. In their experiments, it showed that the Dot-Product heuristic often outperforms FFD-based heuristics. These contributions focus on VM packing, and only consider each request independently.

Metaheuristics. In recent years, many metaheuristic techniques have become prevalent for the approximate solution of multi-objective optimization problems [25]. Mi et al. [17] proposed a genetic algorithm based approach, namely GABA, to adaptively self-reconfigure the VMs in virtualized large-scale data centers consisting of heterogeneous nodes. Xu et al. [25] presented a modified genetic algorithm with fuzzy multi-objective evaluation for efficiently searching the large solution space and conveniently combining possibly conflicting objectives. However, these approaches often take minutes or hours to generate a solution, which face difficulties for a online response.

Cluster Schedulers. Many cluster schedulers have been proposed for different purposes. Sparrow [19] and Tarcil [9] are distributed schedulers developed for clusters that achieve a high throughput for short tasks. Quincy [14], a cluster fair scheduler, models the fair scheduling problem as a minimum cost flow problem to schedule jobs into slots. Firmament [10], a centralized scheduler, achieves low latency via a min-cost max-flow (MCMF) optimization. Differently, ECSched shows that how to encode multi-resource requirements and affinity requirements in MCFP.

6 Conclusion

In this paper, we have presented ECSched, an efficient container scheduler to schedule concurrent containers on heterogeneous clusters. ECSched is a graph-based scheduler, which takes entire deployment requests into consideration for placement decisions. We demonstrate that ECSched can achieve better placement quality than state-of-the-art scheduler in the evaluation. The large-scale simulation shows there are small overheads of ECSched, but it is acceptable in practice. In the future work, we will consider container dependencies and resource dynamics for the scheduler to adopt more sophisticated situations.

Acknowledgments. This research has received funding from the European Union's Horizon 2020 research and innovation program under grant agreements 643963 (SWITCH project), 654182 (ENVRIPLUS project) and 676247 (VRE4EIC project). The research is also funded by Chinese Scholarship Council.

References

1. Amazon web services. https://aws.amazon.com/
2. Docker swarm. https://docs.docker.com/engine/swarm/
3. Google kubernetes. https://kubernetes.io/
4. Microsoft azure. https://azure.microsoft.com/
5. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows. Elsevier, New York (2014)
6. Ajiro, Y., Tanaka, A.: Improving packing algorithms for server consolidation. In: International CMG Conference, vol. 253 (2007)
7. Baldin, I., et al.: ExoGENI: a multi-domain infrastructure-as-a-service testbed. In: McGeer, R., Berman, M., Elliott, C., Ricci, R. (eds.) The GENI Book, pp. 279–315. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33769-2_13
8. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, omega, and kubernetes. Commun. ACM **59**(5), 50–57 (2016)
9. Delimitrou, C., Sanchez, D., Kozyrakis, C.: Tarcil: reconciling scheduling speed and quality in large shared clusters. In: Proceedings of the Sixth ACM Symposium on Cloud Computing, pp. 97–110. ACM (2015)
10. Gog, I., Schwarzkopf, M., Gleave, A., Watson, R.N., Hand, S.: Firmament: fast, centralized cluster scheduling at scale. USENIX (2016)
11. Goldberg, A.V., Tarjan, R.E.: Finding minimum-cost circulations by canceling negative cycles. J. ACM (JACM) **36**(4), 873–886 (1989)
12. Hindman, B., et al.: Mesos: a platform for fine-grained resource sharing in the data center. In: NSDI, vol. 11, p. 22 (2011)
13. Hu, Y., et al.: Deadline-aware deployment for time critical applications in clouds. In: Rivera, F.F., Pena, T.F., Cabaleiro, J.C. (eds.) Euro-Par 2017. LNCS, vol. 10417, pp. 345–357. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-64203-1_25
14. Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., Goldberg, A.: Quincy: fair scheduling for distributed computing clusters. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, pp. 261–276. ACM (2009)

15. Lee, S., et al.: Validating heuristics for virtual machines consolidation. Microsoft Research, MSR-TR-2011-9 pp. 1–14 (2011)
16. Lodi, A., Martello, S., Vigo, D.: Recent advances on two-dimensional bin packing problems. Discret. Appl. Math. **123**(1), 379–396 (2002)
17. Mi, H., Wang, H., Yin, G., Zhou, Y., Shi, D., Yuan, L.: Online self-reconfiguration with performance guarantee for energy-efficient large-scale cloud computing data centers. In: 2010 IEEE International Conference on Services Computing (SCC), pp. 514–521. IEEE (2010)
18. Orlin, J.B.: A faster strongly polynomial minimum cost flow algorithm. Oper. Res. **41**(2), 338–350 (1993)
19. Ousterhout, K., Wendell, P., Zaharia, M., Stoica, I.: Sparrow: distributed, low latency scheduling. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pp. 69–84. ACM (2013)
20. Panigrahy, R., Talwar, K., Uyeda, L., Wieder, U.: Heuristics for vector bin packing (2011). research.microsoft.com
21. Reiss, C., Tumanov, A., Ganger, G.R., Katz, R.H., Kozuch, M.A.: Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In: Proceedings of the Third ACM Symposium on Cloud Computing, p. 7. ACM (2012)
22. Stillwell, M., Schanzenbach, D., Vivien, F., Casanova, H.: Resource allocation algorithms for virtualized service hosting platforms. J. Parallel Distrib. Comput. **70**(9), 962–974 (2010)
23. Taherizadeh, S., Jones, A.C., Taylor, I., Zhao, Z., Stankovski, V.: Monitoring self-adaptive applications within edge computing frameworks: a state-of-the-art review. J. Syst. Softw. **136**, 19–38 (2018)
24. Wang, J., et al.: Planning virtual infrastructures for time critical applications with multiple deadline constraints. Future Gen. Comput. Syst. **75**, 365–375 (2017)
25. Xu, J., Fortes, J.A.: Multi-objective virtual machine placement in virtualized data center environments. In: Proceedings of the 2010 IEEE/ACM International Conference on Green Computing and Communications & International Conference on Cyber, Physical and Social Computing, pp. 179–188. IEEE Computer Society (2010)
26. Zhan, Z.H., Liu, X.F., Gong, Y.J., Zhang, J., Chung, H.S.H., Li, Y.: Cloud computing resource scheduling and a survey of its evolutionary approaches. ACM Comput. Surv. (CSUR) **47**(4), 63 (2015)
27. Zhao, Z., et al.: A software workbench for interactive, time critical and highly self-adaptive cloud applications (switch). In: 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 1181–1184. IEEE (2015)
28. Zhou, H., et al.: Fast resource co-provisioning for time critical applications based on networked infrastructures. In: 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), pp. 802–805. IEEE (2016)