Aladdin: Optimized Maximum Flow Management for Shared Production Clusters

Heng WU, Wenbo ZHANG*, Yuanjia XU, Hao XIANG, Tao HUANG, Haiyang DING, Zheng ZHANG

Institute of Software
Chinese Academy of Sciences
Beijing, China
{wuheng,zhangwenbo,xuyuanjia2017,xianghao16,tao}@otcaix.iscas.ac.cn

***Abstract*—The rise in popularity of long-lived applications (LLAs), such as deep learning and latency-sensitive online Web services, has brought new challenges for cluster schedulers in shared production environments. Scheduling LLAs needs to support complex placement constraints (e.g., to run multiple containers of an application on different machines) and larger degrees of parallelism to provide global optimization. But existing schedulers usually suffer severe constraint violations, high latency and low resource efficiency. This paper describes Aladdin, a novel cluster scheduler that can maximize resource efficiency while avoiding constraint violations: (i) it proposes a multidimensional and nonlinear capacity function to support constraint expressions; (ii) it applies an optimized maximum flow algorithm to improve resource efficiency. Experiments with an Alibaba workload trace from a 10,000-machine cluster show that Aladdin can reduce violated constraints by as mush as 20%. Meanwhile, it improves resource efficiency by 50% compared with state-of-the-art schedulers.**

***Index Terms*—Scheduling, Maximum flow, Long-lived applications, Resource efficiency**

I. Introduction

Long-lived applications (LLAs), including deep learning [1], streaming [2], and latency-sensitive online Web applications [3] are increasingly running on container-based production clusters [4]. Each LLA comprises one or more long-lived containers to encapsulate hardware resources (e.g., CPU and memory), and these containers are allocated and used for durations ranging from hours to months [5]. In this context, cluster schedulers face the challenges of more complex placement constraints [6] and larger degrees of parallelism (e.g., to augment the capabilities of applications by 100 on 11.11 e-commerce holiday [7] or Black Friday [8]). Prior efforts primarily focus on short-lived applications (e.g., MapReduce batch jobs, In-memory Spark querying) to guarantee low latency [5, 9–11]. However, these approaches may suffer severe constraint violations and high latency for LLAs. For example, Medea [6] reveals that constraint-oblivious methods incur high latency compared with constraint-aware ones by up to 3.9× for the 99th percentile. This means enterprises would lose billions of dollars in annual advertising revenues [12] if online applications are crucial and latency-sensitive, such as Google Search.

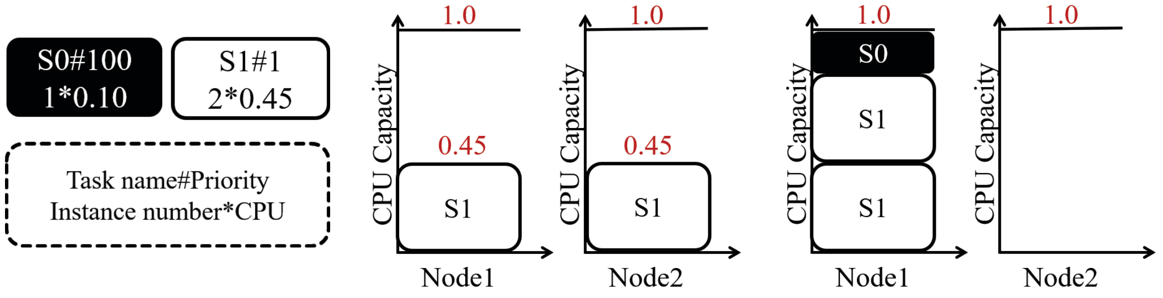
Corresponding author: zhangwenbo@otcaix.iscas.ac.cn

Nowadays, about 35% of the clusters at Microsoft are used exclusively for LLAs [6]. If schedulers cannot express constraints well, large-scale enterprises like Alibaba should at least double the size of their clusters by paying extra tens of million dollars per years [13]. In the Alibaba trace, LLA constraints mainly include anti-affinity and priority. We find that nearly 70% of LLAs have anti-affinity constraints, which means containers within or across applications should run on different machines for performance or reliability requirements [14, 15]. In addition, approximately 15% of containers should meet priority constraints, and a container with a high priority can preempt those with low priorities in the case of placement conflicts [16–18]. Recently, two novel studies were proposed to address the challenges of complex placement constraints. As shown in Figure 1, Firmament [9] can support LLAs by multi-round flow-based scheduling with a timeout mechanism [16]. However, it may encounter continuous conflicts and result in low scheduling quality, which indicates that the cluster needs more resources for those unscheduled tasks. Medea [6] presents an integer linear program (ILP) approach to balance resource efficiency and constraint violations by varied weights, but it could tolerate a few violated constraints if the weighted values are not optimized. A vast amount of research effort has argued that the weight optimization problem is very difficult to solve [16–18].

Despite these observations, support for LLAs in existing schedulers remains rudimentary. We describe the design and implementation of Aladdin to schedule LLAs. Our key contributions are as follows:

- We propose a multidimensional and nonlinear capacity function based on a flow network model to express anti-affinity and priority constraints.
- We design an optimized maximum-flow algorithm to achieve high-quality placements and global objectives, especially when massive LLAs arrive simultaneously.
- We compare our approach to state-of-the-art schedulers with an Alibaba workload trace. Our experiments show that Aladdin reduces constraint violations by as much as 20% and improves resource efficiency by 50%.

The rest of this paper is organized as follows. Section II presents the problem statement and our analysis, and Section



(a) Container requirements (b) Firmament scheduling (c) Medea scheduling

Fig. 1: Three containers, one $S0$ and two $S1$, arrive at the same time. Each container of $S1$ has a higher priority, and it is not recommended to be deployed with $S0$ on the same machine because of anti-affinity constraints. (a) shows the resource requirement for each container. (b) shows the scheduling result of Firmament, $S0$ is unscheduled to avoid anti-affinity constraints, although it is rescheduled many times. (c) shows the scheduling result of Medea. In order to minimize the number of used machines, it violates an anti-affinity constraint since containers $S0$ and $S1$ run on the same machine.

III provides the design of Aladdin. Section IV depicts a detailed overview of our maximum flow algorithm. We report the evaluation of our approach in Section V, followed by a discussion of the related work in Section VI. Finally, Section VII concludes our work and points out the future direction.

II. Problem Statement And Analysis

A. Problem Statement

A cluster consists of fully interconnected machines, each of which can host various LLAs. An LLA comprises several instances (e.g., Tensorflow, Mysql) running in isolated containers. A container is configured with a certain amount of computing resources. When 11.11 e-commerce holiday [7] or Black Friday [8] is approaching, companies will augment the capabilities of applications by about 100× by scheduling massive LLAs in parallel.

When scheduling LLAs, Aladdin satisfies all placement constraints (anti-affinity and priority) instead of tolerating a few constraint violations in Medea [6]. In this context, some containers belonging to the same application should be placed on different machines to decrease the downtime likelihood in case of hardware failures [19]. We call this **anti-affinity within an application**. Conversely, **anti-affinity across applications** means two LLAs should not be deployed on the same machine to avoid critical performance interference. In addition, all containers are classified by different priorities, and schedulers should limit resource provisioning for low-priority containers to avoid affecting high-priority ones. This means if two containers which are competing for bottleneck resources arrive at the same time, we should deploy the high-priority one first.

Hence, Aladdin is designed to meet the following global objectives:

- **Placement constraint expression.** Anti-affinity and priority should be expressed to avoid any constraint violations within or across LLAs.
- **Resource efficiency.** The resource utilization in production clusters should be improved by minimizing the number of used machines.
- **Acceptable placement latency.** The latency of placement decisions should be limited to sub-seconds.

B. Problem Analysis

For given machines, we can reduce the scheduling of LLAs to a flow network problem [20]. A typical application of this problem involves how to find the best delivery route from a factory to a warehouse where the road network has a limited capacity. Further, we propose an optimized maximum flow algorithm by replacing factories with containers and warehouses with machines.

A maximum flow algorithm takes a directed flow network $G = (V, E)$ as input. Each edge $(i, j) \in E$ connects two given vertices $i, j \in V$. Each vertex $v \in V$ has a multidimensional and nonlinear capacity, denoted by $c(i, j)$. In this case, there are two distinct vertices in the flow network: a source $s \in V$ and a sink $t \in V$. If we find a path $s \to v \to t$ without exceeding capacity constraints on any edges, we call it a flow. Formally, a flow in G is a function $f : V \times V \to R$ that satisfies the capacity constraint (Equation 1) and the flow conservation (Equation 2).

$$0 \leqslant f(i, j) \leqslant c(i, j) \tag{1}$$

$$\sum_{v \in V-\{s,t\}} f(i, j) = \sum_{v \in V-\{s,t\}} f(j, i) \tag{2}$$

A maximum flow algorithm like SPFA [21] routes the flow from source s to sink t until $f(i, j) = 0$. But the algorithm faces the two following challenges when scheduling LLAs:

- How to distinguish tasks with different priorities (III.B)
- How to express anti-affinity constraints (III.C)

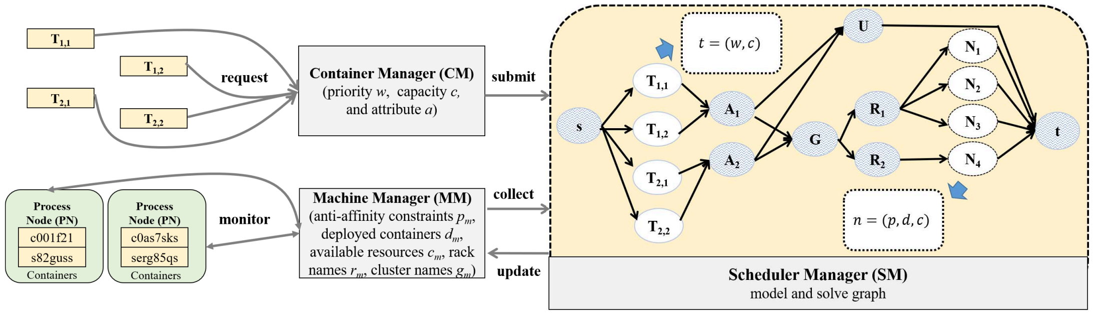


Fig. 2: Aladdin design

III. Aladdin Design

In this section, we depict the flow network structure used in Aladdin. Then, we describe the design in order to find a flow in terms of priority and anti-affinity.

A. Flow Network Structure

Unlike Firmament [9] that uses a one-dimensional and linear function capacity, Aladdin designs a multidimensional and nonlinear capacity function to express placement constraints. As shown in Figure 2, the container manager (CM) submits LLA containers at the same time. Each submission comprises a priority w_n, a resource requirement description c_n and a LLA tag a_n. The machine manager (MM) collects machine runtime statuses including anti-affinity constraints p_m, deployed containers d_m, available resources c_m, rack names r_m and cluster names g_m. The scheduler manager (SM) then solves the flow network constructed from the outputs of CM and MM.

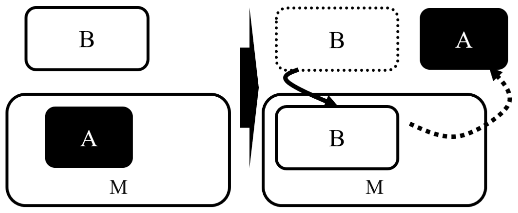
In the flow network, we introduce application vertices A_i, cluster vertices G_i and rack vertices R_i to reduce the total edges from $O(|T| \cdot |N|)$ to $O(|T| + |A| \cdot |R| + |N|)$, which can optimize the placement latency significantly. For example, if we have 100,000 containers and 10,000 machines in a cluster, the latency would be up to several seconds. We observe that all the containers belong to only 13,056 different applications. Then the placement latency can be reduced to hundreds of milliseconds because the number of edges is only about 300 thousands rather than 1 billion.

B. Supporting Priority

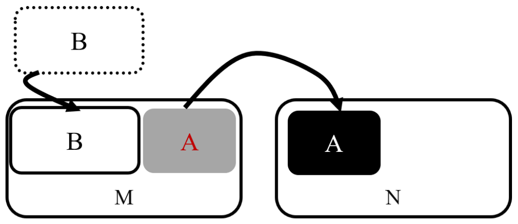
There are two mechanisms (preemption and migration) to increase the flow in the maximum flow theory, but neither of them is aware of priority constraints.

As shown in Figure 3(a), suppose that containers A and B cannot be deployed on the same machine, noting that container A has a higher priority and container B needs more resources. Since the typical maximum flow algorithm is unaware of priority constraints, container B can preempt container A. This preemption implies that a high-priority container could be preempted by a low-priority one to increase the flow.

Based on these observations, we introduce the weighted flow $w_k f(i,j)$ for high-priority containers to increase the network flow. This mechanism ensures high-priority ones cannot



(a) Preemption mechanism



(b) Migration mechanism

Fig. 3: Two mechanisms to increase flows

be preempted. We classify the priorities of all containers using Equation 3, while its output are containers with the same priority w_k. Then, we set the default weight w_1 to 1 for the lowest priority containers (Equation 4). We calculate the weighted values w_k for high-priority containers according to Equation 5, which can guarantee the weighted flow $w_k f(i,j)$ of a high-priority container is larger than any lower-priority ones.

$$x(i) = \{v_x, v_y, ...\}, v_x, v_y \in V \tag{3}$$

$$w_1 = 1 \tag{4}$$

$$w_{i+1} \geqslant \frac{minimize(x(i+1))}{maximize(x(i))}, i \geq 1 \tag{5}$$

Figure 3(b) describes the migration mechanism. We assume: (1) container A has a higher priority, and containers A and B cannot be deployed together; (2) container A is now running on machine M, but container B can only be deployed to machine M; and (3) container A can run on both machines.

In this case, the low-priority container can preempt a high-priority one, since the latter can migrate from M to N. Both containers are deployed using the maximum flow theory. We can obtain the expected result during the scheduling of the LLAs although the theory is unaware of priority constraints. Thus, we decided to follow it.

C. Supporting Anti-affinity

Capacity $c(i, j)$ in the maximum flow theory can be denoted as N-tuples $(x_1, x_2, \cdots, x_n)$ [21, 22], and every element in this tuple is a linear function. As shown in Figure 4, all edge capacities in Aladdin are infinite except those from s to T_i (as $c(s, T_i)$) and from N_j to t (as $c(N_j, t)$). In this context, it is a new flow when Equation 6 is satisfied, where the symbol $\leq$ means the resource requirement of container T_i is less than the resource provisioning of machine N_j.

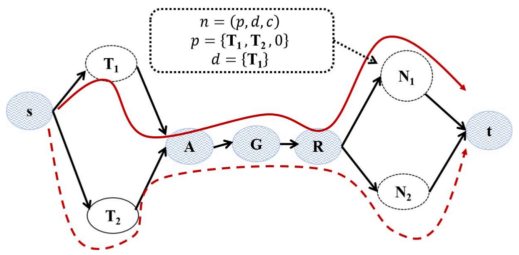


Fig. 4: Non-linear set-based function to support anti-affinity

$$c(s, T_i)(x_1, x_2, ..., x_n) \leqslant c(N_j, t)(x_1, x_2, ..., x_n) \quad (6)$$

However, it is difficult to express anti-affinity constraints by a linear function. We use a non-linear function based on the set theory to support anti-affinity. The symbol $\leq$ is extended to represent $c(s, T_i) \in c(N_j, t)$. As shown in Figure 4, we have a mandatory requirement that containers T_1 and T_2 should run on different machines, denoted by an anti-affinity constraint $p = \{T_1, T_2, 0\}$. After we find the network flow from T_1 to N_1, N_1 updates its deployed container status to $d = \{T_1\}$, which implies that T_2 is added to the blacklist of N_1 according to the anti-affinity constraint $p = \{T_1, T_2, 0\}$. Then, we continue to search new network flows and we cannot deploy container T_2 to machine N_1 by Equation 7 and Equation 8, even if the latter has enough resources.

Equation 7 gives a blacklist of containers for machine N_i, whereas Equation 8 ensures container T_j can be deployed if it is not in the blacklist of machine N_i.

$$blacklist(N_i) = p.get(key), if \forall key \in d \quad (7)$$

$$deployed(T_i) = \begin{cases} 1, T_i \notin blacklist(N_i) \\ 0, T_i \in blacklist(N_i) \end{cases} \quad (8)$$

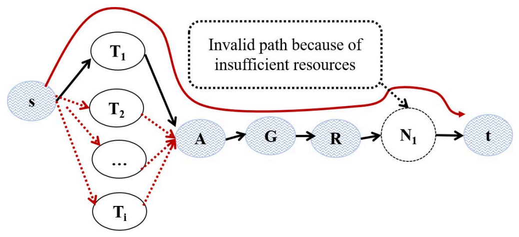
IV. SCHEDULING LLAS

Our goal is to deploy as many LLAs as possible to maximize resource efficiency and gain acceptable placement latency. We maximize Equation 9 by subjecting it to all the properties from Equation 1 to Equation 8. The previous section describes how to find a network flow in terms of priority and anti-affinity to improve resource efficiency. In this section, we present two methods to reduce placement latency, and then describe the algorithm and its implementation.

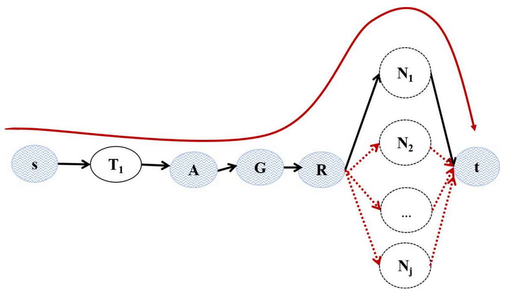
$$Maximize \sum_{(i,j) \in E} w_k f(i, j) \quad (9)$$

A. Reducing Search Space

Placement latency depends on the number of explored paths. Here, a path means an attempt from source vertex s, through T_i, A_j, G_k, R_x, N_y to sink vertex t, as shown in Figure 4. We use two techniques to reduce placement latency, as displayed in Figure 5.



(a) Isomorphism limiting



(b) Depth limiting

Fig. 5: Two techniques to reduce placement latency

The first is isomorphism limiting (IL). All containers T_i belonging to the same application A have the same resource requirement. Here, we call this isomorphism. If the path from T_i to sink vertex t is invalid because of insufficient resource provisioning, we cannot find other valid paths from any other containers T_j of the same application A. So, we can limit the search space of paths when an invalid path is found. As shown in Figure 5(a), application A consists of containers $T_1, T_2, \cdots, T_i$ with the same resource requirements. The capacities of paths from source s to containers $T_1, T_2, \cdots, T_i$ are equal. If Aladdin cannot find a valid flow from container T_1 to machine N_1 through vertices A, G, R, the other containers $T_2, \cdots, T_i$ cannot be deployed. Then Aladdin will skip the searching of containers $T_2, \cdots, T_i$ to N_1.

The second is depth limiting (DL). When we find a valid path from vertex T_i to vertex N_j, we certainly cannot find another valid path to increase its flow by Equation 6. Thus, we do not need to continue searching in this case. As shown in Figure 5(b), the upper bound of the flow in the path from container T_1 to machine N_1 is determined by the capacity of edge $\{T_1, A\}$ (the rest of the capacities are larger or infinite). This implies that the flow through vertex T_1 by any other valid paths is no larger than the capacity of edge $\{T_1, A\}$. So, it is unnecessary to search additional paths to other machines. The algorithm neglects the $red\ vertices$ in Figure 5(b) and only arranges container T_1 to machine N_1.

B. Maximum Flow Algorithm

So far, we have described how to construct a flow network, why Aladdin can meet the above complex constraints, and what techniques can be used to reduce the searching space. We implemented an optimized maximum flow algorithm for Aladdin.

As shown in Algorithm 1, the maximum flow algorithm takes a directed flow network $G = (V, E)$ as input (see Lines 1-5). We use weighted flows to meet priority constraints, and define a non-linear function named $blacklist(\sum_{i \in E} N_i)$ to express anti-affinity constraints (see Lines 6-10). Moreover, we implemented two techniques to guarantee acceptable placement latency and a few constraint violations (see Lines 11-32).

C. Implementation

We present a co-design architecture that can integrate Aladdin to Kubernetes 1.11 [23]. The github source code link is in [24]. As shown in Figure 6, the left is a Kuberentes abstraction and the right is the Aladdin implementation, and we can deploy them both to verify scheduling effectiveness. The architecture consists of three key components: one events handling center (EHC), one model adaptor (MA) and two resolvers (RE).

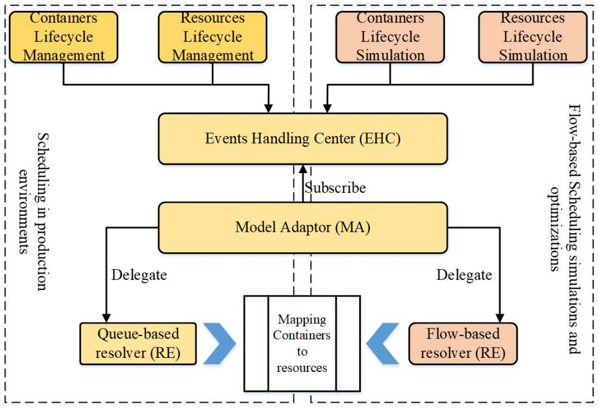

Containers Lifecycle Management
Resources Lifecycle Management
Containers Lifecycle Simulation
Resources Lifecycle Simulation
Events Handling Center (EHC)
Subscribe
Model Adaptor (MA)
Delegate
Delegate
Queue-based resolver (RE)
Mapping Containers to resources
Flow-based resolver (RE)
Scheduling in production environments
Flow-based Scheduling simulations and optimizations

Fig. 6: Aladdin's implementation

EHC receives all kinds of changes in the LLAs' lifecycles and resources. Then, it forwards pre-processed events to MA. HA decouples Kubernetes objects from their scheduling

Algorithm 1 Optimized maximum flow algorithm

```
Input:
    Anti-affinity constraints $p$
    Priority for Container $i$: $w_{T_i}$
    Deployed container set for Machine $j$: $d_{N_j}$
Output:
    The flow represents relationship between container and machine
1:  init $s$ = startVertex
2:  init $e$ = endVertex
3:  init $defCapacity$ = INFINITE
4:  init $graph$ = create ($s$, $e$, $defCapacity$)
5:  init flow $f$ to 0
6:  while getShortestPath() is Not NULL do
7:      $p$ = getShortestPath()
8:      augment $w_{T_i} f$ along $p$
9:      update $balcklist(\sum_{i \in E} N_i)$ based on $p$, $d_{N_j}$
10: end while
11: Function getShortestPath
12: init $minDist$ = INFINITY
13: for each vertex $v$ in $graph$ do
14:     int $dist(s, v)$ = INFINITY // from source to infinity
15: end for
16: init $Q$ = $graph$.getAllVertices()
17: while $Q \neq \phi$ do
18:     Vertex $current$ = $Q$.dequeue()
19:     for vertex $next \in$ neigbors(current) do
20:         if dist(s, next) $<$ dist(s,current) + dist(current,next)
            AND $vertex \notin balcklist(\sum_{i \in E} N_i)$ then
21:             $minDist$ = dist (s, next)
22:         end if
23:         if current.isIsomorphismlimiting() then
24:             Break
25:         end if
26:     end for
27:     if current.isDepthLimiting() then
28:         Break
29:     end if
30: end while
31: return $path$
32: END Function
```

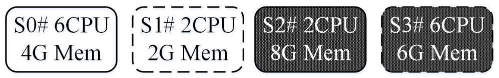
implementation by delegating the $watching$ and $binding$ APIs [23]. RE integrates Aladdin to map containers to resources.

D. Discussion

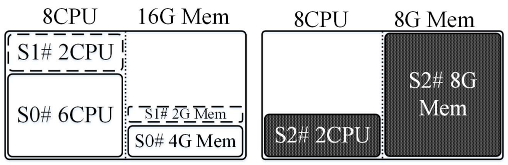
Modern clusters are used for both long- and short- lived applications. Short-lived applications are latency-sensitive and have been studied by many researchers. Aladdin also uses a traditional task-based scheduler for short-lived containers. Moreover, Aladdin can express complex constraints for LLAs. But unlike classical maximum flow algorithms, a container with 4CPUs cannot be broken down into two or more smaller containers deployed respectively. We assume such impartible flows could affect the algorithm's robustness, though our algorithm is working well in Alibaba traces.

Time Complexity. The algorithm in Aladdin is similar to typical flow-based algorithms [21] like SPFA or Bellman-Ford. The worst time complexity is $O(VE^2c)$ and the average is $O(VEc)$, where V denotes the number of vertices, E is the number of edges and c represents the dimension counts of each edge. Adding additional constraints such as memory or heterogeneous resources leads to increased c. However, the effect of c on time complexity is linear and much smaller than E, so it has a limited impact on algorithm performance.

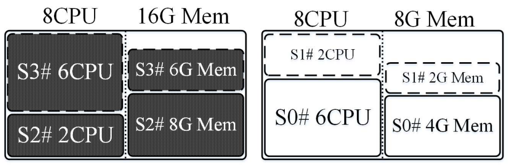
The cost of migration and preemption. Figure 7 demonstrates how preemptions and migrations can affect time complexity with heterogeneous resources. Tasks $S0$, $S1$, $S2$, $S3$ in Figure 7(a) have two-dimensional resource requirements. As shown in Figure 7(b), Aladdin first schedules them sequentially without preemption, and receives notification of the failure of deploying task $S3$. Then Aladdin migrates tasks $S0$, $S1$, $S2$ to the other machine in Figure 7(c). Rescheduling incurs a cost and it would double the scheduling time. But the cost is bound to the worst complexity $O(VE^2c)$.



(a) Container requirements



(b) Aladdin scheduling without preemptions and migrations



(c) Aladdin rescheduling to support preemptions and migrations

Fig. 7: An example of how preemptions and migrations affect Aladdin's time complexity

V. Evaluation

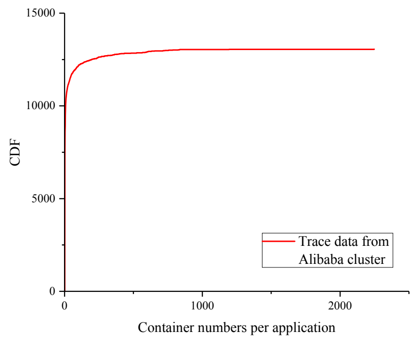
We now evaluate how well Aladdin meets its goals under the LLAs workload:

1) How does Aladdin's placement quality compared with state-of-the-art cluster schedulers?
2) Can Aladdin improve resource efficiency compared with other schedulers?
3) How does Aladdin's placement latency (algorithm overhead), and is it acceptable?

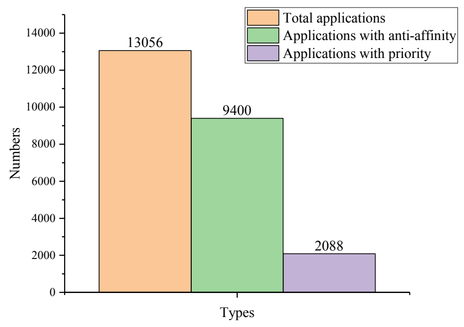
A. Methodology

Workload. We replay a production workload trace from a 10,000-machine Alibaba cluster. These machines are homogeneous with 32 CPU and 64GB memory. As shown in Figure 8(a), it includes about 100,000 tasks belonging to 13,056 LLAs, and nearly 64% of LLAs only have a single instance (container). Moreover, we find that 85% of LLAs have less than 50 containers and a few LLAs are composed of more than 2,000 containers.

Figure 8(b) shows that approximately 70% and 15% of LLAs have anti-affinity and priority constraints, respectively. In addition, we found that several LLAs cannot be co-located with at least other 5,000 containers due to anti-affinity constraints, and these applications usually have higher priorities and larger resource requirements. These features would lead to undeployed containers or container migrations in the following experiments for different schedulers. The maximum resource requirements for the LLAs are 16 CPUs and 32GB RAM.



(a) CDF of container numbers per application



(b) The number of constraints

Fig. 8: Workload features

Simulation. We run Aladdin's codes and scheduling logic on a machine with Intel(R) Xeon(R) CPU E5-2682 v4 CPU(2.50GHz), 32 GB RAM, and 1TB SSD disk, merely stubbing out RPCs and task execution. However, there are three important limitations to be noted. (i) the Alibaba trace contains multi-dimensional resource requests for each task, but we only consider CPU in order to compare Aladdin with Firmament fairly. (ii) there are eight scheduling policies in the Firmament [9] code base currently, and we select the three most used policies. (iii) we implement Go-Kube with a similar node scoring algorithm in Kubernetes 1.11. Table I shows all the state-of-the-art schedulers used in our experiments.

TABLE I: The state-of-the-art schedulers used in our experiments

Name	Description
Firmament-TRIVIAL	Containers always scheduled if resources are idle.
Firmament-QUINCY	Original Quincy cost model, lower cost priority.
Firmament- OCTOPUS	Simple load balancing based on container counts.
Medea	Balance resource efficiency and constraint violations.
Go-Kube	Scoring machines and choose the best one.

B. Placement quality

Metrics. We count the number of undeployed containers after scheduling all LLAs to a 10,000-machine cluster, and the lower number indicates the better the placement quality.

Environments. As described above, Firmament uses a multi-round scheduling and a timeout mechanism for LLAs. It may choose a non-optimized container to reschedule when it detects any constraint conflicts. However, the selected one sometimes may not be deployed to other machines to avoid constraint violations. In this scenario, the solution is to choose another container on the same machine to reschedule once again. Here, we use $reschd(i)$ to express the maximum number of rescheduling containers on the same machine if we encounter constraint conflicts, and we consider the case where i is 1, 2, 4, 8, respectively. Medea [6] uses an integer linear program (ILP) approach to place as many LLAs as possible. It minimizes constraint violations and avoids resource fragmentation using three weights. Each weight is normalized to a value ranging from 0 to 1. Here, we use $weights(a, b, c)$ to express this and consider the case where (a, b, c) is $(1, 1, 1)$, $(1, 1, 0.5)$,$(1, 1, 0)$, $(1, 0.5, 0)$, respectively. In Aladdin, we set the priority w_n to 16, 32, 64, 128 according to Equation 4 (the maximum resource requirement for one application is 16 CPUs).

For Go-Kube/Kubernetes, 21.2% of the containers in Figures 9(a) to 9(d) cannot be deployed although it supports anti-affinity and priority constraints. One of the main reasons is that Go-Kube supports them separately, which means it cannot reach global optimization when accounting for both constraints. Moreover, similar conclusions have been drawn from Medea.

For Firmament-TRIVIAL, the number of undeployed containers ranges from 4.3% to 34.7%, as shown in Figures 9(a) to 9(d). The goal of Firmament-TRIVIAL is to minimize the number of used machines, so it always tries to deploy a container to the most packed machines. It probably ignores anti-affinity and priority in the first round. Then, Firmament-TRIVIAL detects constraint conflicts and uses simple policies to choose a container to reschedule. Unfortunately, conflicts may occur again because the container selecting policy because it is difficult for the container selecting policy to achieve global objectives.

Firmament-OCTOPUS deploys a container to those machines with the least number of containers. Although the number of undeployed containers is less than 10.7%, as shown in Figures 9(a) to 9(d). However, Firmament-OCTOPUS still suffers from constraint violations.

Firmament-QUINCY was originally designed without support for anti-affinity, but it can be enhanced by the multi-round scheduling mechanism. As shown in Figures 9(a) to 9(d), Firmament-QUINCY 's number of undeployed containers varies between 3.5% and 25.1%. It may encounter a large amount of constraint violations set with the non-optimal parameter i.

Medea is designed to maximize the number of deployed containers and minimize violated constraints by weights. As shown in Figures 9(a) to 9(d), if we set the weight (the third parameter) to 0, Medea cannot tolerate violated constraints, then the number of undeployed containers is reduced to 5.2%. Otherwise, it would increase to 12.9%.

Aladdin outperforms the other schedulers on all tests as it can deploy all containers to machines without constraint violations. We can easily calculate the parameter to express constraints and priorities, while other schedulers need to identify the optimized parameters empirically.

Finally, Figure 9(e) shows that for all schedulers except Aladdin, anti-affinity violations are inevitable and the ratio of anti-affinity constraint violations is 65% at least.

C. Resource efficiency

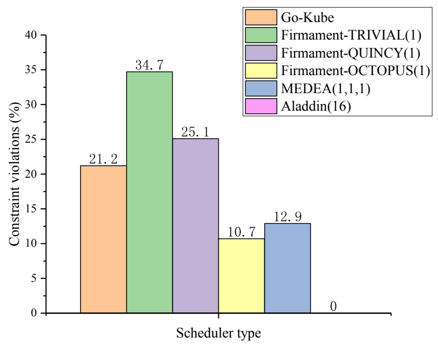
Metrics. We count the machines used for 100,000 containers, denoted $num(sched\_name)$. Then, the cluster efficiency can be calculated by Equation 10.

$$efficiency_i = \frac{num(i)}{min\{num(i), num(j), ...\}} - 1 \tag{10}$$

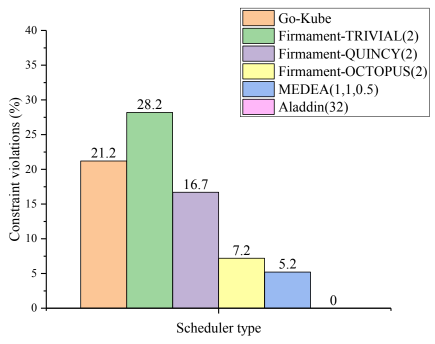
Here, $min\{num(i), num(j), \cdots\}$ denotes the minimal number of machines used for different schedulers.

Environments. We take Go-Kube, Medea(1,1,0), Firmament-QUINCY(8) and Aladdin(16) for a comparison. Related parameters are set optimally according to our previous tests. Here we consider four characteristic situations for arriving containers: (i) containers with high-priorities first, (ii) containers with low-priorities first, (iii) containers with a large number of anti-affinity constraints first, (iv) containers with a small number of anti-affinity constraints first.

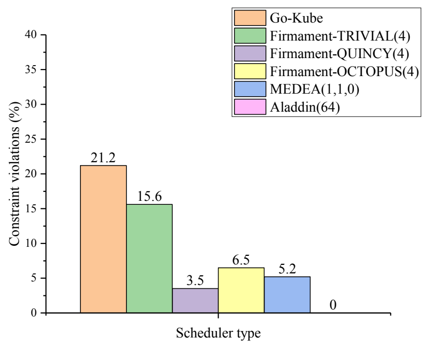
As shown in Figure 10, Aladdin outperforms the other schedulers on all tests. It only needs 9,242 machines, while Firmament-QUINCY and Medea needs more than 10,200 machines. Go-Kube needs 14,211 machines in the worst-case scenario, which is 1.54 times more than Aladdin.



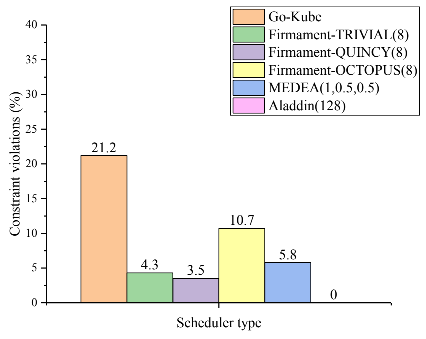
(a) Firmament(1), Medea(1,1,1), Aladdin(16)



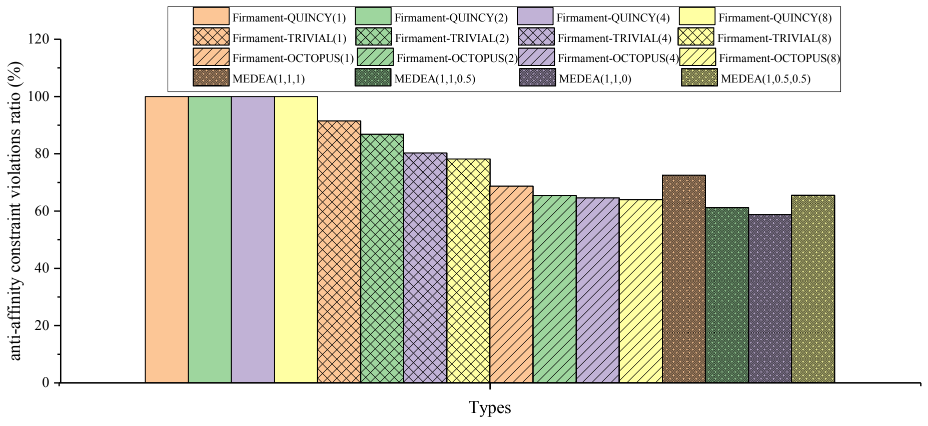
(b) Firmament(2), Medea(1,1,0.5), Aladdin(32)



(c) Firmament(4), Medea(1,1,0), Aladdin(64)



(d) Firmament(8), Medea(1,0.5,0.5), Aladdin(128)



(e) The ratio of anti-affinity constraint violations to total constraint violations

Fig. 9: Placement quality

Moreover, we find that the algorithms of Firmament-QUINCY and Aladdin are robust. Both of them use flow networks with proof-of-concept solving algorithms. Medea integrates the ILP algorithm, which is essentially an approximation algorithm. Its results are relatively steady for all tests according to Equation 10. Since the queuing model depends on the arrival order of containers, Go-Kube gives wide-ranging results on different tests.

In fact, Aladdin benefits from a multidimensional and nonlinear capacity function, hence it avoids constraint violations by using the robust maximum flow algorithm. Although Firmament-QUINCY also has the same model, it employs low-efficient multi-round scheduling mechanism to support anti-affinity.

Figure 11 also reflects the similar experiment results from a different perspective. The vertical axis represents the range of resource efficiency for all used machines. For example, resource efficiency ranging from 20% to 70% means at least at least one of used machines resource efficiency is 20%, and one of them is 70%. It is clear to see the robustness of the algorithm in Aladdin.

D. Placement latency

Metrics. Placement latency is the time period between the submission of all LLAs and the completion of placements. It also indicates the algorithm overhead. To make a fair comparison with state-of-the-art schedulers, we calculate the placement latency per container (aka average placement latency) using Equation 11.

$$latency = \frac{time(i)}{total\_containers} \tag{11}$$

Here, $latency$ means the average placement latency of 100,000 containers with the specific scheduler i.

Environments. As described in Section IV, there are two ways to reduce the placement latency: isomorphism limiting (IL) and depth limiting (DL). Three policies are used in this test: the $Aladdin$ policy refers to the maximum flow algorithm without any optimizations, the $Aladdin + IL$ policy uses the maximum flow algorithm with IL optimization, and the $Aladdin + IL + DL$ policy represents the maximum flow algorithm with both optimizations.

As shown in Figure 12, Firmament-QUINCY's latency is only about 50ms and outperforms the others, no matter how the cluster size changes. We also observe a similar trend for latency for the three Aladdin policies, whereas hundreds of milliseconds can be acceptable in the production environment. In contrast, the latency of Go-Kube and Medea exceeds one second with an increase in cluster scale. Diverse time complexity causes these latency differences: the former is an $O(n)$ algorithm and the latter is exponential. Furthermore, latency can be reduced by 50% if we use $Aladdin+IL+DL$ instead of the $Aladdin$ policy.

Using the $Aladdin+IL+DL$ policy, we then measure the algorithm's overhead for four different arrival characteristics for containers, as demonstrated in Figure 13: (i) containers with high priority first (aka CHP), (ii) containers with low priority first (aka CLP), (iii) containers with a large number of anti-affinity constraints first (aka CLA), (iv) containers with a small number of anti-affinity constraints first (aka CSA).

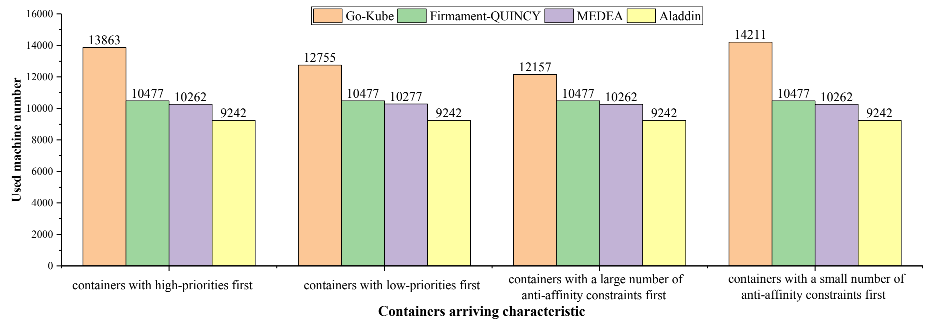


Fig. 10: Number of machines used for different container arrival characteristics

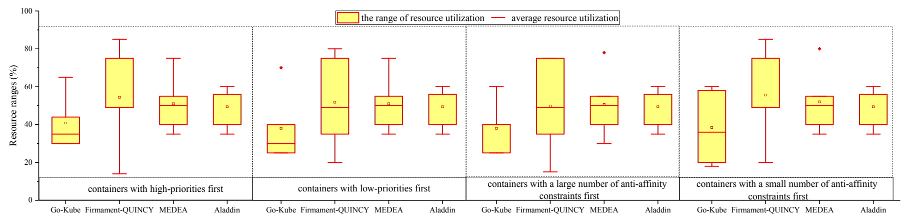


Fig. 11: Resource efficiency for different container arrival characteristics

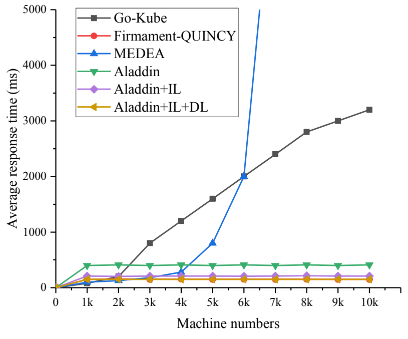
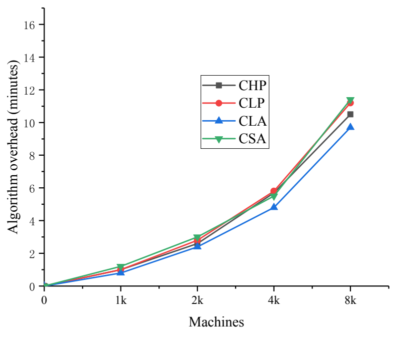
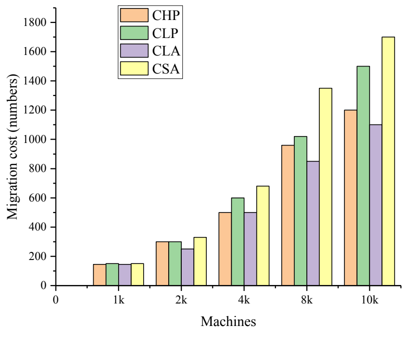


Fig. 12: Average placement latency



(a) Aladdin's approach scales as cluster size grows



(b) The cost of migration and preemption

Fig. 13: Algorithm overhead

As shown in Figure 13(a), the algorithm runtime under the four situations increases linearly when deploying the containers, and it needs about 15 minutes to complete the placement decision of 100,000 containers in the worst case (CSA). In practice, this is acceptable in the Alibaba trace. Moreover, we can see that Aladdin can reduce the latency by 30% in the best case (CLA). The overhead of different algorithms for the same LLAs size depends on the cost of migrations and preemptions.

Figure 13(b) shows the migration of CSA is nearly 1,700, which means only 1.7% of the total containers need to be migrated. The main reasons are: (i) the average resource utilization for the used machines is less than 50%, which can alleviate preemption costs, and minimize migrations; (ii) LLAs with higher priorities always have more instances and larger resource requirements. CHP and CSA policies can effectively reduce resource fragments, as described in Section IV.D.

VI. Related work

DAG-oriented Schedulers. YARN [25] and Mesos [11] were originally designed for task-based DAG jobs to increase resource efficiency. Sparrow [10], Mercury [4], Apollo [26] and 3Sigma [27] employs distributed or hybrid architectures to provide the high cluster efficiency and reduce placement latency. Carbyne [28] extends Mesos to keep fairness while shortening the job completion time (JCT). It employs DRF [29] or HUG [30] with a delay scheduling mechanism [31] to schedule DAG tasks. HaaS [32] employs sub-graphs to represent the topology of both clusters and DAG tasks. It also supports GPUs and FPGAs. Quincy [20] and Firmament [9] balance data-locality, placement latency and cluster efficiency. Unlike Aladdin, these DAG-oriented schedulers achieve optimized global objectives such as placement latency or JCT but they find it difficult to adjust LLAs (e.g., Aurora [33], Marathon [34]) due to complex constraints.

In recent years, data-driven scheduling models have been proposed to achieve higher cluster efficiency in the production environment. Microsoft [35], Alibaba [36], and Google [37] have already make their cluster traces open source. Several research efforts [38–40] have focused on tasks with complex dependency structures and heterogeneous resource demands. They focus on long-running DAG tasks with tough-to-pack resource demands and compute a DAG schedule offline, while Aladdin is an online scheduling system.

LLA-aware schedulers. Google's Borg [5] and its open-source version Kubernetes [23] manage both LLAs and short jobs in queuing models. They integrate many virtualization technologies (e.g., GCE, LXC, Docker) naturally. Borg only considers affinity constraints between tasks and machines, while Kubernetes provides more. Besides, many efforts of VM placements [41, 42] are mainly on SLAs to support elementary constraints. Although they support many constraints, neither is able to achieve optimal global objectives in the long run. Our experiments have similar results.

Apart from queue-based scheduling's locality, ILP-based scheduling is a natural choice to achieve optimal global objectives. FlowTime [43] concentrates on meeting deadlines for both individual jobs and LLAs. It uses ILP to minimize the turnaround time of ad-hoc jobs, which makes scheduling more balanced in mixed clusters. TetriSched [15] takes both placements and deadlines as constraints, but the latency could be intolerable when workload is high. Medea's [6] inputs are the LLAs with placement constraints. Medea can achieve optimal scheduling results within acceptable latency when its weighted values are properly set. ILP-based schedulers need to prove the availability of arithmetical values, otherwise they can face constraints violation or an increase in latency again. In this paper, Aladdin's non-linear function makes it much easier than ILP-based methods to express constraints within graceful latency degradation.

In conclusion, both LLAs and task-based DAG jobs have common characteristics, which makes it possible to design schedulers which consider elementary constraints. For example, many research efforts on VM placements mainly focus on SLAs [35, 41, 42, 44]. Some schedulers, like SLAQ [45] and Optimus [39], build a relationship between the resource provision and the convergence rate of machine learning applications.

VII. Conclusion

In this paper, we presented Aladdin, a scheduler for efficiently scheduling long-lived applications (LLAs) with containers. Aladdin is the first system that fully supports anti-affinity and priority constraints both within and across LLAs, which is crucial for the performance of LLAs. It designs a multidimensional and nonlinear capacity function based on flow network model. Experiments with an Alibaba workload trace from a 10,000-machine cluster show that Aladdin can effectively reduce constraint violations. We will extend the flow-based model to support heterogeneous workloads in the near future.

Acknowledgment

This work was supported by National Key Research and Development Program of China (2018YFB1003602), National Natural Science Foundation of China (61572480), and Alibaba Group through Alibaba Innovative Research (AIR) Program.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[2] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[3] Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. Perfiso: performance isolation for commercial latency-sensitive services. 2018.

[4] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *USENIX Annual Technical Conference*, pages 485–497, 2015.

[5] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

[6] Panagiotis Garefalakis, Konstantinos Karanasos, Peter R Pietzuch, Arun Suresh, and Sriram Rao. Medea: scheduling of long running applications in shared production clusters. In *EuroSys*, pages 4–1, 2018.

[7] Taobao. http://www.taobao.com.

[8] Black friday. https://blackfriday.com.

[9] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert Nicholas Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. Usenix, 2016.

[10] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[11] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[12] Google's marissa mayer: Speed wins. https://www.zdnet.com/article/googles-marissa-mayer-speed-wins.

[13] Leiphone. https://www.leiphone.com/news/201610/R0FXfe0mJlmu-eVT7.html.

[14] Bikash Sharma, Victor Chudnovsky, Joseph L Hellerstein, Rasekh Rifaat, and Chita R Das. Modeling and synthesizing task placement constraints in google compute clusters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 3. ACM, 2011.

[15] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 35. ACM, 2016.

[16] Ionel Corneliu Gog. *Flexible and efficient computation in large data centres*. PhD thesis, University of Cambridge, 2018.

[17] Malte Schwarzkopf. *Operating system support for warehouse-scale computing*. PhD thesis, University of Cambridge, 2018.

[18] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 450–462. ACM, 2015.

[19] Anup Das, Akash Kumar, and Bharadwaj Veeravalli. Reliability and energy-aware mapping and scheduling of multimedia applications on multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):869–884, 2016.

[20] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi

Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276. ACM, 2009.

[21] Ravindra K Ahuja. *Network flows: theory, algorithms, and applications*. Pearson Education, 2017.

[22] Offer Shai. Multidimensional max-flow method and its application for plastic analysis. *Advances in Engineering Software*, 36(6):401–411, 2005.

[23] Kubernetes. https://kubernetes.io/.

[24] Aladdin. https://github.com/qcase/aladdin.

[25] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[26] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *OSDI*, volume 14, pages 285–300, 2014.

[27] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A Kozuch, and Gregory R Ganger. 3sigma: distribution-based cluster scheduling for runtime uncertainty. In *Proceedings of the Thirteenth EuroSys Conference*, page 2. ACM, 2018.

[28] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *OSDI*, pages 65–80, 2016.

[29] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Nsdi*, volume 11, pages 24–24, 2011.

[30] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. Hug: Multi-resource fairness for correlated and elastic demands. In *NSDI*, pages 407–424, 2016.

[31] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278. ACM, 2010.

[32] Jiong He, Yao Chen, Tom ZJ Fu, Xin Long, Marianne Winslett, Liang You, and Zhenjie Zhang. Haas: Cloud-based real-time data analytics with heterogeneity-aware scheduling. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1017–1028. IEEE, 2018.

[33] Alibaba aurora. http://aurora.apache.org.

[34] Marathon. http://mesosphere.github.io/marathon.

[35] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167. ACM, 2017.

[36] Alibaba cluster data. https://github.com/alibaba/clusterdata.

[37] Google cluster data. https://github.com/google/cluster-data.

[38] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. G: Packing and dependency-aware scheduling for data-parallel clusters. In *Proceedings of OSDI16: 12th USENIX Symposium on Operating Systems Design and Implementation*, page 81, 2016.

[39] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, page 3. ACM, 2018.

[40] Luping Wang and Wei Wang. Fair coflow scheduling without prior knowledge. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 22–32. IEEE, 2018.

[41] Ayoub Alsarhan, Awni Itradat, Ahmed Y Al-Dubai, Albert Y Zomaya, and Geyong Min. Adaptive resource allocation and provisioning in multi-service cloud environments. *IEEE Transactions on Parallel and Distributed Systems*, 29(1):31–42, 2018.

[42] Pawel Janus and Krzysztof Rzadca. Slo-aware colocation of data center tasks based on instantaneous processor requirements. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 256–268. ACM, 2017.

[43] Zhiming Hu, Baochun Li, Chen Chen, and Xiaodi Ke. Flowtime: Dynamic scheduling of deadline-aware workflows and ad-hoc jobs. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 929–938. IEEE, 2018.

[44] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, et al. Morpheus: Towards automated slos for enterprise clusters. In *OSDI*, pages 117–134, 2016.

[45] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J Freedman. Slaq: quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 390–404. ACM, 2017.