Provisioning with Fine-grained Affinity for Container-enabled Cloud-edge System

Dingkun Yang, Nan Zhao, Hongshuang Ma, Jiasheng Yang
Jiangsu Electric Power Information Technology Co. Ltd, Jiangsu Nanjing 210000, China
Email: {{yangdk1_js, mhshuang, yjshengshe}@126.com, 784526430@qq.com}

***Abstract*—Containerization effectively decouples the applications and facilitates those user-defined dependencies for extreme elasticity via existing container management frameworks. However, the variance occurs among container-enabled hosts at edges, since their copies of images and libraries, and available resources are quite different. Then, even for the same container, the costs of retrieving the dependency also vary over hosts, since existing mechanism forces each host to retrieve the missing parts from the central repository only upon its own status. Although previous researches have optimized the management of dependencies, the deployment mainly focuses on the entire image. Instead, upon the fine-grained difference on dependencies, we propose to minimize the overall latency of both retrieving the missing parts and the executions under constrained resources. We propose an integer linear program and then design a randomized algorithm to select suitable hosts to deploy the containers. The probability calculated in algorithm shows the preference and the affinity of hosts, and is derived from the status of fine-grained dependencies. Via rigorous proof, the result of our algorithm is concentrated on its optimum with high probability. Extensive simulations confirm that our algorithm outperforms the alternatives up to 44% regarding the average completion time of containers.**

I. Introduction

Traditionally, virtual machine is the mainstream deployment approach for applications, as it provides strictly isolated environments over heterogeneous hosts [1, 2]. However, hardware virtualization has various limitations, such as weak resilience, long startup time, and low resource utilization [3–5]. To improve system elasticity and application performance, operating system-level virtualization, better known as containerization, becomes an attractive alternative for deploying and running the applications. Essentially, containerization decouples the applications from the operating system, shrinking the deployment sizes by order of the magnitude for better efficiency.

However, provisioning applications over container-enabled hosts at edges is non-trivial and faces multiple challenges:

First and foremost, each container-enabled host is individual, i.e., existing mechanisms for those container management frameworks retrieve the missing dependencies for each host only upon its current status [6–8]. Then, since the status of stored images and libraries, as well as available resources varies over hosts, shown in Table I, even deploying the same container, the cost regarding retrieving the dependencies also varies. Therefore, it is preferred to deploy the containers to the hosts with sufficient dependencies stored, in order to decrease the costs for retrieving the missing parts after the deployment.

Second, those container-enabled hosts deployed at the edge of the network, e.g., within an edge cluster [9], are resource-constrained [10]. If too many containers are deployed within a few hosts, although the cost for retrieving necessary missing dependencies is optimized, the resource-constrained hosts fail to support such deployment. Furthermore, due to heterogeneous capabilities over the hosts, the execution latency for the same container varies as well. Then, considering both capacity and capability of heterogeneous container-enabled hosts, the overall latency for the deployment and execution should be optimized simultaneously for the whole system.

TABLE I: Variance over Container-enabled Hosts

	Variance (Status of Dependencies and Rsources)			
Hosts	Image Layer[1]	Size (GB)	CPU Cores	RAM (GB)
Inspur 1	316	18	48	73
Inspur 2	169	7.4	40	115
Inspur 3	143	8	48	91
PowerEdge 1	163	16	56	93

Third, all the hosts within an edge cluster communicate with the central repository for necessary missing dependencies over wide area network [11], which often incurs heavy traffic. For a batch of containers needed to be deployed, the overall cost of retrieving missing dependencies should be optimized in terms of the traffic and the bandwidth usage over wide area network, instead of individual optimization, which is determined by the deployment of the containers and current status of hosts.

Existing research falls insufficient for addressing the aforementioned challenges. Some [12–15] have studied the optimization for container deployment, but fail to consider current status of images and libraries. Others [6, 16–18] have designed the mechanism for runtime optimization for containers, but fail to consider the minimization of retrieving missing dependencies. And the rest [19–21] have proposed the provisioning of edge resources, but fail to combine heterogeneous container-enabled hosts with fine-grained dependency status.

In this paper, we propose to minimize the overall latency regarding both retrieving the missing dependencies among all container-enabled hosts and the executions of all containers, under the consideration of constrained resources. After that, we formulate such optimization problem as an integer linear program, whose objective is to minimize the overall latency and the constraints consider the capacity of resources at edges. Due to the NP-hardness of our proposed problem, we propose to relax the domain of the problem, and we use the results

[1]"Image Layer" is used for Docker, i.e., a line in a Dockfile defines a layer.

978-0-7381-2646-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00225

solved from the relaxed problem as a guidance for container deployment. Essentially, the results solved are a series of reals, and they actually show the preference of all candidate hosts upon current status of dependencies and available resources. By using such fine-grained affinity between the containers and hosts upon dependencies and available resources, the overall latency is optimized. We thus use such reals as the probabilities to deploy all of the containers to these hosts.

Via rigorous proof, the result obtained from our proposed algorithm is concentrated on its optimum with high probability, based on Martingale Analysis. Furthermore, the violation in terms of the resource constraints is also studied after applying our designed algorithm. Extensive simulations confirm that our designed algorithm Clipper outperforms the alternatives up to 45.9% and 44% in terms of the deployment efficiency and average completion time, respectively, and manifests excellent stability under skewed package distribution.

II. Related Work

Since the introduction of Docker, many container orchestration systems have become available [13, 22, 23]. However, these container schedulers are agnostic to the container image, leading to non-trivial deployment latency [12]. We investigate the literature and classify them into two categories: optimization for container deployment and runtime optimization.

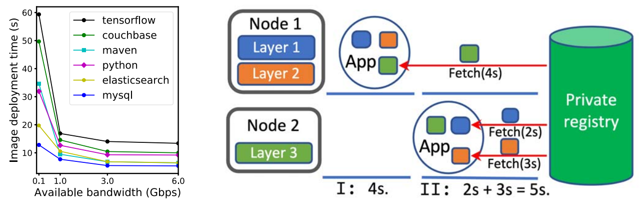
A. *Optimization for Container Deployment*

PASch [12] used the mapping function to map the container to two package-affinity nodes, containing the largest package required by the container. The node with the least load would spawn the container unless the node's load exceeded the pre-defined threshold. If so, the scheduler would then fall back to the least loaded scheduling policy and dispatch the container to the least load node. Docker Swarm [13] changed the default scheduling policy from bin-packing to leverage the potentially under-utilized resources. Such spread strategy attempted to schedule a container to the least loaded node based on assessing the resources available on cluster nodes. Mao *et al.* [14] analyzed the default spread strategy and found it is insufficient to leverage heterogeneous resources. They proposed DRAPS, a resource-aware placement scheme to improve system performance in the heterogeneous cluster. Boza *et al.* [15] analyzed the impact of scheduling strategies on container performance from two dimensions of container initialization time and container execution performance. They further revealed that the performance of containerized applications is closely related to deployment strategies.

However, they fail to propose performance-aware deployment algorithms. Instead, we propose Clipper for optimizing average completion time by considering container deployment efficiency and container execution performance.

B. *Runtime Optimization for Containers*

Oakes *et al.* [16] proposed Pipsqueak to improve function launch times by pre-importing the packages required by the function. As the popularity of packages changed rapidly, the



(a) Various Images (b) Various Strategies

Fig. 1: Cost regarding the Deployment for Containers

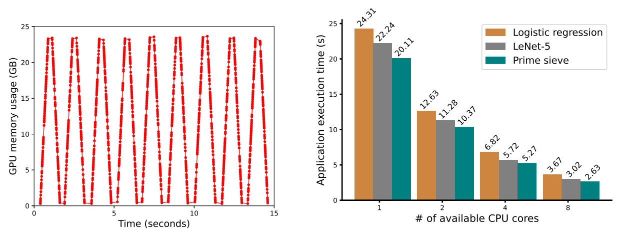
pre-import approach may cause memory thrashing, leading to low resource utilization and degraded application performance. Oakes *et al.* [17] also built SOCK, which used lightweight isolation primitives to reduce container initialization latency and designs a generalized Zygote-provisioning strategy to reduce the initialization cost of the runtime environment. SOCK potentially assumed that container images are already available on the assigned nodes and ignores container deployment costs. Zheng *et al.* [6] designed Wharf, which shared images through a shared storage layer to reduce network and storage overhead when running large-scale containerized applications. Wharf supported concurrent image retrieval and data access through a layer-based lock mechanism, potentially incur data synchronization overhead. Wharf provided a promising solution for reducing container deployment costs in large-scale data processing scenarios. FogDocker [18] observed that a bloated container image might degrade the deployment efficiency.

However, only a small part of packages are required during container startup. Therefore, they propose restructuring the container image and downloading only the necessary packages when creating the container. The lazy retrieval mechanism may incur high runtime overhead when the available network bandwidth is scarce. As a result, we propose Clipper.

III. Background and Motivation

Cost for Container Deployment. The container image is structured by several layers. When we send a request to the Docker daemon for spawning a new container from the specific container image, the Docker daemon first checks whether the required container image is available locally. If not, it retrieves the image manifest from the Docker registry, then downloads and extracts the missing image layers to local storage. Once the container image is available locally, the Docker daemon will instantiate a container from the specific container image to run the application code. To quantify the deployment costs of container images, we remove all image layers on the node and build a private registry to avoid the impact of pull rate limits of Docker Hub[2]. As shown in Fig. 1(a), when the available bandwidth is scarce (e.g., 0.1Gbps), the average deployment time of six representative container images is as high as 34.68s. When the available bandwidth is increased to 1Gbps, the deployment costs of container images drop rapidly. Unfortunately, deployment performance gains decrease quickly when the available bandwidth beyond the best trade-off point. The main reason

[2]Docker Hub: https://hub.docker.com/. Accessed Feb 2021.



(a) Runtime Details (b) Resource-execution Trade-off

Fig. 2: Dynamic Resource Usage Patterns

is that different image layers within the same container image are dependent during extraction. Therefore, the extraction time depends on the single-core processor's restricted clock speed, which dominates the deployment process, especially when the available bandwidth is sufficient for retrieving missing parts.

Case Study. As shown in Fig. 1(b), The container image of the containerized application is structured by layer-1, layer-2, and layer-3. Node-1 caches layer-1 and layer-2, and node-2 caches layer-3. We assume the retrieval costs of layer-1, layer-2, and layer-3 are 2s, 3s, and 4s, respectively. Scheduler-I assigns the application to node-1, upon fine-grained package affinity. Then, node-1 needs to retrieve layer-3 from the private registry. The deployment cost is 4s. Scheduler-II deploys the application to node-2, based on the largest package required by the application. The deployment cost is 2s + 3s = 5s. The illustrative example shows that the deployment strategy with fine-grained package affinity achieves better deployment efficiency compared to the simple greedy strategy.

Dynamic Resource Usage Patterns. Fig. 2(a) shows the GPU memory of ResNet-50 model training on one NVIDIA Tesla V100 accelerator during a 15s snapshot. The maximum and minimum usages of GPU memory are 23.4GB and 0.31GB. The GPU memory usage during the model training only reaches peak resource usage at few moments. Although the NVIDIA Tesla V100 is equipped with 32GB memory capacity, the static resource allocation strategy based on peak resource usage can only train one model exclusively, leading to a significant waste of expensive resources and degraded overall application performance. Unlike static policy, dynamic resource allocation strategy based on cgroups primitives [24] can allocate resources according to real-time resource demands to exploit potential utilization and parallelism [25].

Case Study. To explore the impact of the resources available on the node on application performance, we train the LeNet-5 model, the logistic regression model and write an OpenMP implementation to find all prime numbers up to 10^9 under different resource constraints. As shown in Fig. 2(b), resource-intensive applications can seamlessly use idle resources to improve parallelism. Therefore, the execution performance of compute-intensive applications is almost linearly correlated with available computing resources.

Summary. Primary case studies have two implications for designing efficient container scheduling strategies. First, the deployment strategy with fine-grained package affinity that perceives the hierarchical structure of the container image may achieve the best deployment efficiency. Second, scheduling containers to nodes with sufficient available resources could exploit potential parallelism to improve application execution performance. Therefore, we consider the joint optimization of deployment efficiency and execution performance to minimize the average completion time of containerized applications.

IV. System Model and Problem Formulation

Let S denotes the collection of containerized applications. The container-$s \in [S]$[3] has the following attributes: C_s, the *least* number of CPU cores required to meet the minimum computing resource requirements; M_s, the amount of memory size required to execute; E_s, the execution time on a single CPU core, which can be obtained from historical information or application performance modeling [26]; and I_s, the collection of image layers for creating the container-s. Besides, we have $I_s \subseteq I$, where I denotes the universe collection of all container image layers. For each image layer $i \in I$, l_i represents the retrieve cost of the image layer i. Let V denotes the collection of resource nodes. The node-$v \in [V]$ also has the related attributes: C_v, the number of available CPU cores; M_v, the amount of available memory size; and I_v, with $I_v \subseteq I$, the collection of image layers that are already available.

We model the mapping between the container-s and the node-v with binary decision variable b_v^s: $b_v^s = 1$, if the container-s is deployed on the node-v; $b_v^s = 0$, otherwise. For the node-v, we define the number of allocated CPU cores as $C(v) = \sum_{s\in[S]} C_s \cdot b_v^s$, the amount of allocated memory size as $M(v) = \sum_{s\in[S]} M_s \cdot b_v^s$, the total deployment time on the node-v as $D(v) = \sum_{s\in[S]} \sum_{i\in I_s \setminus I_v} l_i \cdot b_v^s$. Considering the characteristics of elastic resource allocation, we define the total execution time of containerized applications on the node-v as $E(v) = \frac{1}{C_v} \sum_{s\in[S]} E_s \cdot b_v^s$. We denote the processing time of the node-v as the sum of deployment time and execution time on the node. In order to optimize the average application completion time, we minimize the maximum node processing time. The detailed description is illustrated as follows:

$$
\begin{aligned}
[\min] \quad & \max_{v} \left\{ \sum_{s\in[S]} \left(\sum_{i\in I_s \setminus I_v} l_i \cdot b_v^s + \frac{E_s}{C_v} \cdot b_v^s \right) \right\} \\
\text{s.t.} \quad & C_1: \ C(v) = \sum_{s\in[S]} C_s \cdot b_v^s, && (1) \\
& C_2: \ M(v) = \sum_{s\in[S]} M_s \cdot b_v^s, && (2) \\
& C_3: \ C(v) \le C_v, \qquad \forall\, v \in [V] && (3) \\
& C_4: \ M(v) \le M_v, \qquad \forall\, v \in [V] && (4) \\
& C_5: \ \sum_{v\in[V]} b_v^s = 1, \qquad \forall\, s \in [S] && (5) \\
& C_6: \ b_v^s \in \{0, 1\}, \qquad \forall\, s \in [S],\ v \in [V] && (6)
\end{aligned}
$$

Resource capacity constraints (3) and (4) guarantee that the resource allocated by the node-v cannot exceed the available

[3]We define $[S] = \{1, 2, \ldots, |S|\}$, where S can be different quantities.

resource capacity. The scheduling constraints (5) and (6) guarantee that each container is assigned to one and only one node. Unfortunately, the min-max ILP problem has been proven to be an NP-hard problem [27]. To effectively obtain the approximate optimal deployment decisions, we use linear programming relaxation and randomized rounding tools to design the randomized algorithm in the next section.

V. Design Randomized scheme

The key to minimize the maximum node processing time is to construct the optimal mapping between containers and resource nodes. Unfortunately, due to the inherent computational complexity of the ILP, we usually cannot obtain the optimal solution in a reasonable time, even for a moderate scale problem. Therefore, we use the linear programming relaxation and randomized rounding techniques to obtain the approximate optimal ILP solution in polynomial time.

Linear Programming Relaxation. To obtain effective scheduling scheme in polynomial time, we relax the ILP to linear program (LP) by removing the constraint that the decision variables have to take value from binary values. Specifically, we relax the domain of decision variables from the binary domain $\{0,1\}$ to the probability interval $[0,1]$. The relaxed model is illustrated as follows:

$$\begin{aligned}
[\min] \quad & \max_{v} \left\{ \sum_{s\in[S]} \left(\sum_{i\in I_s\backslash I_v} l_i \cdot p_v^s + \frac{E_s}{C_v} \cdot p_v^s \right) \right\} \\
\text{s.t.} \quad & C_1: C(v) = \sum_{s\in[S]} C_s \cdot p_v^s, \\
& C_2: M(v) = \sum_{s\in[S]} M_s \cdot p_v^s, \\
& C_3: C(v) \le C_v, \qquad \forall\, v \in [V] \\
& C_4: M(v) \le M_v, \qquad \forall\, v \in [V] \\
& C_5: \sum_{v\in[V]} p_v^s = 1, \qquad \forall\, s \in [S] \\
& C_6: p_v^s \in [0,1], \qquad \forall\, s \in [S],\ v \in [V]
\end{aligned}$$

Through existing efficient algorithms, we can find the optimal fractional solution of the LP in polynomial time. Then we take the fractional solutions as the guidance to design the randomized rounding scheme for container deployment.

Randomized Rounding Scheme. We consider the fractional solution of the LP as the preference of the container to choose resource node. More specifically, the value of p_v^s indicates the correlation between the container-s and the node-v. For any container-$s \in [S]$, we randomly select a value r_s from $(0,1]$. For completeness, we let $p_0^s = 0$. If $r_s \in \left(\sum_{j=0}^{v-1} p_j^s,\ \sum_{j=0}^{v} p_j^s\right], v \in [V]$. Then $b_v^s = 1$, otherwise, $b_v^s = 0$. The randomized rounding scheme sensibly ensures the scheduling constraints, and indicates the preference of the container to resource nodes simultaneously. The detailed scheduling algorithm is shown in Algorithm 1. Line 1 initializes the scheduling decision with empty set. Line 1 finds the fractional solution of LP with the polynomial algorithm. Then

Algorithm 1 Container deployment algorithm.

```
Require: The container-$s$ and its attributes, $\forall s \in [S]$;
         The resource node-$v$ and its attributes, $\forall v \in [V]$.
Result: Scheduling decision.
 Result ← {}
 $\{p_v^s\} \leftarrow$ LP solver
 foreach container-$s$ in $[S]$ do
     $r_s \leftarrow$ random(0, 1)
     foreach node-$v$ in $[V]$ do
         if $r_s \in (\sum_{j=0}^{v-1} p_j^s, \sum_{j=0}^{v} p_j^s]$ then
             $b_v^s = 1$;
         else
             $b_v^s = 0$;
foreach container-$s$ in $[S]$ and node-$v$ in $[V]$ do
    if $b_v^s == 1$ then
        Result ← Result ∪ $\{< s, v >\}$;
return Result
```

we use the randomized rounding scheme to obtain the original problem's integer solution. Finally, we summarize and return the scheduling decision to real container deployment.

VI. Analysis of the randomized algorithm

In this section, we prove that the deployment scheme of Clipper concentrates on the optimal solution with high probability, i.e., $1 - \mathcal{O}(e^{-t^2})$, where t is the concentration bound. First, we show that the difference between the completion time of node-v contributed by container-s and its expectation could be bounded through Martingale analysis. Then, we use Azuma's inequality to illustrate the gap between the feasible and optimal solutions. For ease of description, we denote SOL as the solution solved by Clipper and OPT as the optimum.

Theorem 1. *$Pr\left[SOL - OPT \le t\right] \ge 1 - \mathcal{O}(e^{-t^2})$.*

Proof. Firstly, we denote the contribution of each container-s to the load of node-v as follows:

$$T_v^s = \left(\sum_{i\in I_s\backslash I_v} l_i + \frac{E_s}{C_v} \right) * b_v^s = Z_v^s * b_v^s, \tag{7}$$

where Z_v^s is a constant value when given container-s and node-v. According to randomized rounding strategy in Algorithm 1, we have $Pr[b_v^s = 1] = p_v^s$. The expectation of T_v^s is represented as follows:

$$E[T_v^s] = E[b_v^s] * Z_v^s = (Pr[b_v^s = 1] * 1 + 0) * Z_v^s = p_v^s * Z_v^s. \tag{8}$$

For each node-v, the difference between the load contributed by container-s, i.e., T_v^s and its expectation, i.e., $E[T_v^s]$ is defined as follows:

$$D_v^s = T_v^s - E[T_v^s]. \tag{9}$$

We define the sum of D_v^s as $L_v^{|S|}$, i.e.,

$$L_v^{|S|} = \sum_{s=1}^{|S|} D_v^s = L_v^{|S|-1} + D_v^{|S|}. \tag{10}$$

The expectation of L_v^s, on the condition $L_v^1, L_v^2, \cdots, L_v^{s-1}$ is expressed as follows:

$$\begin{aligned}
&E[L_v^s|L_v^1, L_v^2, \cdots, L_v^{s-1}] \\
&\overset{(10)}{=} E[L_v^{s-1} + D_v^s|L_v^1, L_v^2, \cdots, L_v^{s-1}] \\
&= E[L_v^{s-1}|L_v^1, L_v^2, \cdots, L_v^{s-1}] + E[D_v^s|L_v^1, L_v^2, \cdots, L_v^{s-1}] \\
&\overset{(9)}{=} L_v^{s-1} + E[T_v^s - E[T_v^s]|L_v^1, L_v^2, \cdots, L_v^{s-1}] \\
&= L_v^{s-1} + E[T_v^s|L_v^1, \cdots, L_v^{s-1}] - E[E[T_v^s]|L_v^1, \cdots, L_v^{s-1}] \\
&= L_v^{s-1} + E[T_v^s] - E[T_v^s] \\
&= L_v^{s-1}. \qquad (11)
\end{aligned}$$

From the previous equation (11), we conclude that the sequence $L_v^1, L_v^2, \cdots, L_v^{|S|}$ is the martingale sequence. Without loss of generality, we assume $L_v^0 = 0$. Then, $\forall s \in [S]$, we have the following inequation:

$$|L_v^s - L_v^{s-1}| \overset{(10)}{=} |D_v^s| \overset{(9)}{=} |T_v^s - E[T_v^s]| \leq g_v^s,$$

where $g_v^s = \max\{Z_v^s - E[T_v^s], E[T_v^s]\}$. When we obtain the fractional solutions from the efficient LP algorithm, $E[T_v^s]$ is then determined. Therefore, the consecutive items in the martingale sequence has a constant bound, which meets the conditions of Azuma's inequality. Then, we have

$$Pr\left[L_v^{|S|} - L_v^0 \geq t\right] \leq \exp\{-\frac{t^2}{2\sum_{s=1}^{|S|}(g_v^s)^2}\},$$

where t is the concentration bound. According to equation (10) and equation (9), we have

$$Pr\left[\sum_{s=1}^{|S|} T_v^s - \sum_{s=1}^{|S|} E[T_v^s] \geq t\right] \leq \exp\{-\frac{t^2}{2\sum_{s=1}^{|S|}(g_v^s)^2}\},$$

which is equivalent to

$$\begin{aligned}
Pr\left[\sum_{s=1}^{|S|} T_v^s \leq \sum_{s=1}^{|S|} E[T_v^s] + t\right] &\geq 1 - \exp\{-\frac{t^2}{2\sum_{s=1}^{|S|}(g_v^s)^2}\} \\
&= 1 - \mathcal{O}(e^{-t^2}). \qquad (12)
\end{aligned}$$

For ease of description, we let $S_v = \sum_{s=1}^{|S|} T_v^s$, $E_v = \sum_{s=1}^{|S|} E[T_v^s]$. After substituting the corresponding variables in inequation (12), we have

$$Pr\left[S_v \leq E_v + t\right] \geq 1 - \mathcal{O}(e^{-t^2}), \qquad (13)$$

where S_v denotes the actual load of node-v, and E_v denotes the expectation load of node-v. As LP provides a lower bound of ILP, which is a minimization problem. $\forall v \in [V]$, we have

$$E_v \leq OPT. \qquad (14)$$

Without loss of generality, we assume u and v are the indexes of the maximal S_v and E_v, respectively, i.e.,

$$u = \arg\max_i S_i, \qquad (15)$$

$$v = \arg\max_j E_j. \qquad (16)$$

Then, we have the following inequations,

$$SOL \overset{(15)}{=} S_u \overset{(13)}{\leq} E_u + t \overset{(16)}{\leq} E_v + t \overset{(14)}{\leq} OPT + t. \qquad (17)$$

Based on inequation (13) and (17), we conclude that

$$Pr\left[SOL \leq OPT + t\right] \geq 1 - \mathcal{O}(e^{-t^2}).$$

In real scenarios, the concentration bound is acceptable given a certain probability, e.g., $1 - \mathcal{O}(1 - e^{-t^2}) = 0.9$. □

Theorem 2. *The expected violation of constraints is 0.*

Proof. All constraints are linear. And by using the linearity of the expectation, we conclude such theorem.

We take Constraint (3) as an example. By using previous rounding strategy, we have $E[b_v^s] = p_v^s$. Then, we have

$$\begin{aligned}
E[C(v) - C_v] &= E[\sum_{s\in[S]} C_s \cdot b_v^s - C_v] \\
&= \sum_{s\in[S]} C_s \cdot E[b_v^s] - C_v \\
&= \sum_{s\in[S]} C_s \cdot p_v^s - C_v \\
&\leq C_v - C_v = 0,
\end{aligned}$$

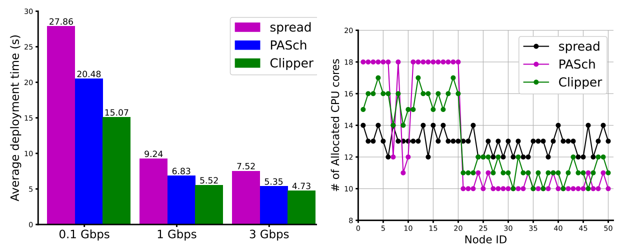
where it refers to the expected violation of constraints. □

VII. Performance Evaluation

A. Experimental Setup

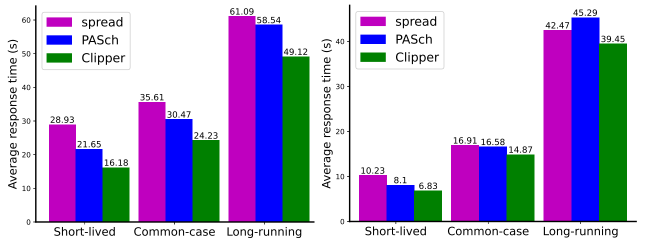
We use the SimPy[4] simulation framework to implement the simulation experiments. For ease of description, we regard the image layer as the package throughout the paper and set the following configuration parameters: The number of worker nodes is 50. The number of available CPU cores on each node follows an exponential distribution with a mean of 20. The size of the package set is 100. The number of package replicas is 10. Packages are cached uniformly distributed among nodes. The number of packages cached by each node follows an exponential distribution with a mean of 20. The average image deployment times derive from the preliminary experimental: 34.68s, 11.97s, and 8.81s under 0.1Gbps, 1Gbps, and 3 Gbps network bandwidth, respectively. The number of containerized applications is 500. The number of CPU cores required by the application is uniformly sampled from $\{1, 2\}$. Each container image requires a random number of packages, sampled from an exponential distribution with a mean of 10. The data size of the packages in an image follows Zipf distribution[5], with parameter s = 1.1. The package retrieval time is proportional to the data size of the package. To evaluate the performance of Clipper, we use three types of workloads: short-lived, common-case, and long-running workloads. The execution time of these three types of workloads on a single CPU core follows the exponential distributions with means of 2s, 16s, and 64s, respectively. The load threshold parameter in PASch [12] is set to 0.9 to avoid overload.

[4]SimPy: https://simpy.readthedocs.io/en/latest/. Accessed Feb 2021.
[5]Zipf's law: https://en.wikipedia.org/wiki/Zipf's_law. Accessed Feb 2021.



(a) Various Bandwidth (b) Skewed Distribution

Fig. 3: Results under Various Settings for Containers



(a) Low Bandwidth (b) High Bandwidth

Fig. 4: Results under Various Bandwidth and Workloads

B. Experimental Results

Results for Container Deployment. To compare the deployment performance of different container schedulers, we evaluate the deployment cost of short-lived workloads under different available bandwidths. When available bandwidth is scarce, the deployment cost of bloated container images dramatically depends on the deployment strategy. Fig. 3(a) shows that the average deployment time of Clipper is 15.07s, which is 45.9% lower than the 27.86s of spread, and 26.4% lower than the 20.48s of PASch. As the available bandwidth increases, the image download time drops rapidly, and the extraction stage dominates the container deployment process. When the available bandwidth is sufficient, Clipper improves deployment efficiency by significantly reducing the data size of the image layers that need to be retrieved. Compared with spread and PASch, Clipper achieves 37.1% and 11.5% deployment time optimizations, respectively. Experiment results confirm that Clipper greatly optimizes container deployment performance by considering fine-grained package affinity.

Impact of Skewed Package Distribution. To investigate the impact of package distribution on Clipper's performance, we use the first 20 nodes to cache packages and clean up the remaining 30 nodes' packages. Fig. 3(b) shows the number of allocated CPU cores on each node for common-case workloads. Clipper and PASch typically prefer to assign containers to the first 20 nodes where required packages are cached to improve deployment efficiency. Although PASch uses the power-of-two-choice strategy to reduce the degree of load imbalance, 85% of the nodes with cache packages have reached the load threshold limits. Moreover, the loads on the 7th, 9th, and 10th nodes are significantly lower than expected. After carefully checking the software packages cached by these three nodes, we find that they are almost all small-size packages. PASch only considers the largest package's affinity, potentially ignoring the role of small-size packages, hence regards these three nodes as trivial nodes. Moreover, the load threshold parameter of PASch is not easy to tune to appropriate value under diverse scenarios manually, and it dramatically affects application execution performance. Compared with the extreme distribution of PASch, Clipper has better stable performance under skewed package distribution.

Results of Completion Time. As illustrated in Fig. 4, the average completion time of Clipper consistently outperforms alternatives under different network scenarios and diverse workloads. More specifically, when the available bandwidth is scarce, the average completion times of the three types of workloads scheduled by Clipper are 16.18s, 24.23s, and 49.12s, respectively, which are 44%, 31.9%, and 19.6% lower than spread. As application runtime increases, the execution period gradually dominates the container lifecycle. Clipper tends to deploy containers to nodes with sufficient resources rather than nodes with lower deployment costs, which sacrifices deployment efficiency in exchange for better execution performance. For long-running workloads, sufficient available resources typically are the primary consideration of the scheduler. Spread greedily deploys each container to the node with the lowest load to achieve *temporary* load balancing, resulting in sub-optimal global resource allocation. Instead of simple greedy strategies, Clipper considers the optimal container placement scheme from the overall perspective of resource requirements. Therefore, Clipper can make global optimal scheduling decisions to improve resource utilization and application execution performance. Experiment results confirm that Clipper reduces the average application completion time of long-running workloads by 19.6% and 7.1% under different available bandwidths compared with the spread. PASch inherently tolerates high load imbalance, which dramatically deteriorating the application performance of long-running workloads. Specifically, PASch prolongs the average application completion time of long-running workloads by 6.6% compared with the spread when bandwidth is sufficient.

VIII. Conclusion

In this paper, we design Clipper towards an efficient container scheduler by jointly optimizing the cost of retrieving the dependencies and the executions. In particular, we model the container deployment problem as an integer linear program and design an efficient randomized algorithm to find suitable hosts with fine-grained affinity. Additionally, we prove the result is concentrated on its optimum with high probability. Simulations demonstrate that Clipper significantly outperforms alternatives in terms of deployment performance, average completion time, and scalability under different settings. As future work, we plan to explore the impact of node failure and dynamic re-balancing of already deployed containerized applications over their lifetimes towards greater resource utilization and scheduling efficiency.

References

[1] K. N. Vhatkar and G. P. Bhole, "Optimal container resource allocation in cloud architecture: A new hybrid model," *Journal of King Saud University - Computer and Information Sciences*, 2019.

[2] J. Zhang, H. Huang, and X. Wang, "Resource Provision Algorithms in Cloud Computing: A Survey," *Journal of Network and Computer Applications*, vol. 64, pp. 23–42, 2016.

[3] M. Mao and M. Humphrey, "A Performance Study on the VM Startup Time in the Cloud," 2012, pp. 423–430.

[4] M. Turowski and A. Lenk, "Vertical scaling capability of openstack - survey of guest operating systems, hypervisors, and the cloud management platform," in *ICSOC Workshops*, 2014.

[5] Q. Zhang, L. Liu, C. Pu, Q. Dou, L. Wu, and W. Zhou, "A Comparative Study of Containers and Virtual Machines in Big Data Environment," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 178–185.

[6] C. Zheng, L. Rupprecht, V. Tarasov *et al.*, "Wharf: Sharing Docker Images in a Distributed File System," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18, p. 174–185.

[7] C. L. Abad, E. F. Boza, and E. V. Eyk, "Package-Aware Scheduling of FaaS Functions," *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, p. 101–106, 2018.

[8] Z. Huang, S. Wu, S. Jiang, and H. Jin, "Fastbuild: Accelerating docker image building for efficient development and deployment of container," in *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2019, pp. 28–37.

[9] E. Nygren, R. K. Sitaraman, and J. Sun, "The akamai network: a platform for high-performance internet applications," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 3, pp. 2–19, 2010.

[10] Y. Jin, Z. Qian, S. Guo, S. Zhang, X. Wang, and S. Lu, "Ran-gjs: Orchestrating data analytics for heterogeneous geo-distributed edges," in *Proceedings of the 47th International Conference on Parallel Processing*, 2018, pp. 1–10.

[11] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica, "Low latency geo-distributed data analytics," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 421–434, 2015.

[12] G. Aumala, E. Boza, L. Ortiz-Avilés *et al.*, "Beyond Load Balancing: Package-Aware Scheduling for Serverless Platforms," in *CCGRID 2019*, pp. 282–291.

[13] I. M. A. Jawarneh, P. Bellavista, F. Bosi *et al.*, "Container Orchestration Engines: A Thorough Functional and Performance Comparison," in *ICC 2019*, pp. 1–6.

[14] Y. Mao, J. Oak, A. Pompili *et al.*, "DRAPS: Dynamic and Resource-Aware Placement Scheme for Docker Containers in a Heterogeneous Cluster," in *IEEE IPCCC 2017*, pp. 1–8.

[15] E. F. Boza, C. L. Abad, S. P. Narayanan *et al.*, "A Case for Performance-Aware Deployment of Containers," in *Proceedings of the 5th International Workshop on Container Technologies and Container Clouds*, ser. WOC '19, p. 25–30.

[16] E. Oakes, L. Yang, K. Houck *et al.*, "Pipsqueak: Lean Lambdas with Large Libraries," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pp. 395–400.

[17] E. Oakes, L. Yang, D. Zhou *et al.*, "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers," in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '18, p. 57–69.

[18] L. Civolani, G. Pierre, and P. Bellavista, "FogDocker: Start Container Now, Fetch Image Later," pp. 51–59.

[19] Y. Jin, L. Jiao, Z. Qian, S. Zhang, N. Chen, S. Lu, and X. Wang, "Provisioning edge inference as a service via online learning," in *2020 17th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*. IEEE, 2020, pp. 1–9.

[20] J. Meng, H. Tan, C. Xu, W. Cao, L. Liu, and B. Li, "Dedas: Online task dispatching and scheduling with bandwidth constraint in edge computing," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 2287–2295.

[21] Z. Zhou, S. Yu, W. Chen, and X. Chen, "Ce-iot: Cost-effective cloud-edge resource provisioning for heterogeneous iot applications," *IEEE Internet of Things Journal*, vol. 7, no. 9, pp. 8600–8614, 2020.

[22] K. Hightower, B. Burns, and J. Beda, "Kubernetes: Up and Running: Dive into the Future of Infrastructure."

[23] A. Verma, L. Pedrosa, M. Korupolu *et al.*, "Large-Scale Cluster Management at Google with Borg," ser. EuroSys '15, 2015.

[24] C. Arango, R. Dernat, and J. Sanabria, "Performance Evaluation of Container-based Virtualization for High Performance Computing Environments," *CoRR*, 2017. [Online]. Available: http://arxiv.org/abs/1709.10140

[25] Y. Xu, W. Chen, S. Wang *et al.*, "Improving Utilization and Parallelism of Hadoop Cluster by Elastic Containers," in *IEEE INFOCOM 2018*, 04, pp. 180–188.

[26] S. Kundu, R. Rangaswami, K. Dutta *et al.*, "Application performance modeling in a virtualized environment," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–10.

[27] Karp, Richard M., *Reducibility among Combinatorial Problems*, pp. 85–103.