Application Oriented Dynamic Resource Allocation for Data Centers Using Docker Containers

Xinjie Guan, Xili Wan, Baek-Young Choi, Sejun Song, Jiafeng Zhu

Abstract— **Docker offers an opportunity for further improvement in data centers' (DCs) efficiency. However, existing models and schemes fall short to be efficiently used for Docker container based resource allocation. We design a novel Application Oriented Docker Container (AODC) based resource allocation framework to minimize the application deployment cost in DCs, and to support automatic scaling while the workload of cloud applications varies. We then model the AODC resource allocation problem considering features of Docker, various applications' requirements and available resources in cloud data centers, and propose a scalable algorithm for DCs with diverse and dynamic applications and massive physical resources.**

Index Terms—**Data center, resource allocation, application oriented, cost efficient**

I. Introduction

Resource allocation and management in data centers (DCs) have been widely studied to save the cost and improve resource usage efficiency. For the applications deployed in a multi-tier structure, a single computing job is divided into multiple tasks by a central manager, and each task is assigned to a worker node. While the workload varies, scaling is easy by adjusting the amount of physical resources assigned to each task [1]. For complicated applications, e.g., game hosting, video conferences, resources are allocated in the application level as Hypervisor based **virtual machines (VMs)** [2, 3]. As the workload changes, resources allocated to applications could automatically scale from the horizontal direction, i.e., adding more VMs [4], and the vertical direction, i.e., allocating more resources to deployed VMs [5]. However, horizontal scaling may take dozens of seconds to deploy VMs or wake up a **physical machine (PM)**, while vertical scaling needs additional support from both the host operation system (OS) and guest OS. Power management techniques, e.g., Dynamic Voltage Frequency Scaling (DVFS), have been utilized to improve energy efficiency in various cloud environments [6, 7], but may reduce performance and increase latency.

As an alternative to Hypervisor based VMs, Linux containers [8] provide superior system efficiency and isolation. Due to its light-weight, containers can be promptly deployed and swiftly migrated between different PMs. Compared with Hypervisor based VMs, deploying applications with containers has three main differences: 1) the number of containers and their capacities need not to be specified but adaptively adjusted based on available physical resources and applications' workload on the fly. 2) OS and applications' supporting libraries could be layered and reused to improve the memory efficiency of containers by utilizing recently invented container management tool Docker [9]. 3) The deployment cost of the same container may be different on different PMs considering libraries reusing. Despite that there are a few papers [10, 11] focusing on resource allocation in container based clouds, they did not consider the features of containers.

We introduce a framework for Application Oriented Docker Container (AODC) based resource allocation in DCs and formulate the AODC resource allocation problem as an optimization problem considering the features of container and Docker. The new features of the AODC framework and model can be summarized as the followings: 1) The number and capacity of containers are adaptively determined based on not only applications' requirements but also available resources on PMs. 2) The containers deployment cost is related to available supporting libraries on PMs and required libraries of applications. 3) Resource management and application execution functions are decoupled, and resource management is performed in a distributed manner. To the best of our knowledge, we are the first one to study the resource allocation mechanism using Docker containers. We further develop an communication-efficient and scalable algorithm to solve the AODC optimization problem. The benefits of the proposed framework and algorithm are validated through evaluations.

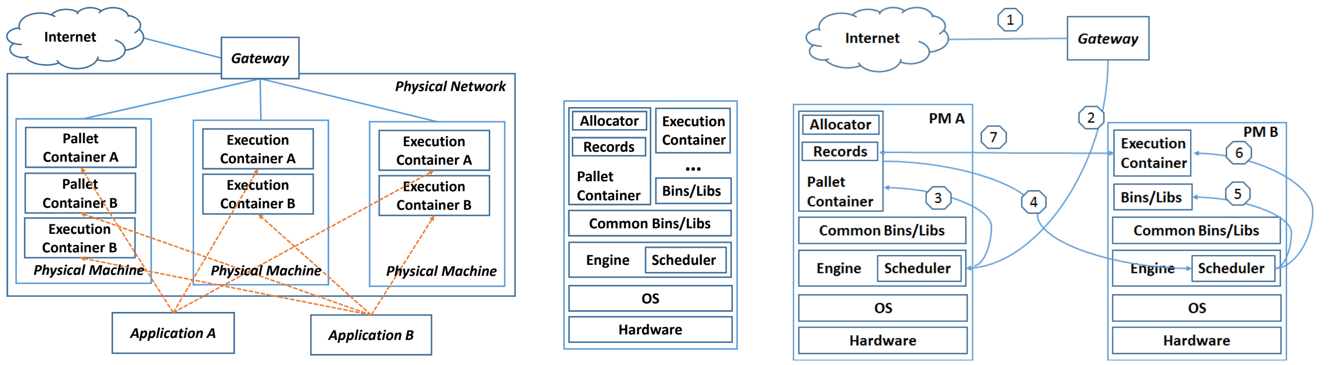
II. Application Oriented Resource Allocation Framework using Docker Container

To efficiently deploy network applications with diversity requirements and varying workloads, we introduce an AODC based resource allocation framework, including a new scheduler and two kinds of containers, namely **Pallet Containers (PCs)** and **Execution Containers (ECs)**, which decouple the resource management and task execution for each application. Unlike the Hypervisor based VM placement, the number of ECs and their demands on physical resources are dynamically determined by not only the applications' workload but also on the available resources in the DC.

Figure 1(a) illustrates the overview for the AODC framework. When deploying an application, resources are allocated to the application as a set of containers distributed on multiple

Xinjie Guan and Xili Wan are with Department of Computer Science and Technology, Nanjing Tech University, Nanjing, 211816, China (e-mail: xjguan,xiliwan@njtech.edu.cn). Xili Wan is the corresponding author of this paper. Baek-Young Choi and Sejun Song are with the School of Computer Science and Electric Engineering, University of Missouri-Kansas City, 5110 Rockhill Rd, MO 64110, USA (e-mail: choiby,sjsong@umkc.edu). Jiafeng Zhu is with Futurewei Technology Inc., 2330 Central Expy, CA 95050, USA (e-mail: jiafeng.zhu@huawei.com).

This work is supported in part by the National Nature Science Foundation of China (61602235), and the Natural Science Foundation of Jiangsu Province of China (BK20161007).



(a) Framework overview (b) Components (c) Workflow of deploying an application in a DC

Fig. 1. Application Oriented Docker Container (AODC) based resource allocation framework

PMs. In particular, each application has a PC and at least one EC. A **PC** makes resource allocation decisions, requests resources for ECs, tracks the task status on ECs, and manages the life cycle of ECs. **ECs** complete the assigned tasks, and report to the PC about their task execution status compared with the expected progress.

Both PCs and ECs are embedded on PMs. Figure 1(b) depicts the main components in a PM. Here, each PM has a OS, on which Docker runs an engine to maintain the operating environment for containers, assist embedding containers, and isolate containers running on the same PM. In the Docker engine, we introduce a novel **scheduler** to manage the PCs' life cycle. When activating an application, the scheduler creates a PC and assigns resources to it based on the application's requirements and available resources on the PM. In addition, the Docker engine enables the OS kernel and common libraries to be shared by multiple execution containers, so that aggregating applications that share some common libraries could further save costs by reducing redundancy.

We illustrate the workflow of deploying an application in a DC in Figure 1(c). When an application request arrives (steps (1-2)), the scheduler of this application initiates a PC and assigns physical resources for the PC (step (3)). The PC analyzes the application's requirements, makes resource allocation decisions using the AODC algorithm, and requests resources on PM B based on the decision (step (4)). The AODC algorithm is discussed in Section III. If the request has been approved by the scheduler on PM B, resources are provisioned and an EC is created (steps (5-6)). After that, tasks are assigned to the EC according to the resource allocation decision. A PC may repeat the steps (4-6) to create multiple ECs based on the resource allocation decision. The created ECs work on assigned tasks and update task execution status to the PC (step (7)). Based on the real time workload of the application, a PC may dynamically adjust the number, location and assigned resources of ECs. When the application is deactivated, the PC is terminated and its resources are collected by the scheduler.

Using this framework, physical resources allocated for each application could dynamically shrink or expand based on the application's requirements, real time workload and available resources in the DC. To further minimize the cost of deploying applications in the framework, we model the AODC based resource allocation as an optimization problem and develop a scalable algorithm to solve this problem.

III. System Model and Algorithm

We model the AODC based resource allocation problem as an optimization problem aiming to minimize the application deploying cost while satisfying QoS requirements. Considering scalability, we design a distributed AODC algorithm called by PCs to incrementally search proper PMs and make resource allocation decisions.

A. System model

We consider a physical network $G(N^p)$ with a set N^p of PMs. Each PM $j \in N^p$ is equipped with limited physical resources. c_j^p represents the residual resources on PM j. Here, we use computational resources as an example to model the resource allocation problem. We assume that these PMs are identical in capacity and price. However, the PMs may install different libraries in advance to support different applications.

We represent each application job A_i as a tuple $(w_i, T_i, \alpha_i, F_i)$. w_i is the workload of job A_i. We quantify w_i as the unit amount of time that the job could be finished by a unit amount of computational resource. T_i specifies the maximum allowed service delay to complete job A_i. A job can be divided into multiple small tasks that are parallel executed on different ECs. Assuming that all the ECs serving job A_i are started and terminated at the same time, at least $C_i^a = \frac{w_i}{T_i}$ unit amount of computational resources are demanded to guarantee the service delay bound T_i. Meanwhile, a task cannot be smaller than an atomic operation. Using α_i to denote the ratio of the workload of an atomic operation to the total workload of job A_i, each task should be larger than $w_i \cdot \alpha_i$. F_i indicates the expected total amount of internal traffic between the PC and all ECs. After the scheduler of this application initiates a PC, the PM embedding this PC is determined. We denote this PM as I_i.

Given a physical network G and a set of applications $\{A_i | i \in \{1, .., M\}\}$, we want to determine the number and placement of ECs, and the amount of tasks assigned to each EC so that the total cost of applications is minimized. The cost of an application in the AODC framework basically comes from the node cost U_i^{node} and link cost U_i^{link}. Specifically, U_i^{node} includes deployment cost and execution cost of ECs, while U_i^{link} quantifies the communication cost between the PC and ECs.

We follow the widely accepted energy consumption model [3, 12] to estimate the cost of a PM j that consists of baseline cost P_s and operation cost P_o. Here, we assume that the operation cost is proportional to the workload assigned to the PM j. To save energy cost, we utilize PMs that support the sleep/awake mode, and assume no baseline cost for a PM in sleep mode. Besides the baseline cost and operation cost, deploying an EC may bring additional cost P_d for installing and configuring supporting libraries of the application.

Considering the PM's sleep/awake status and available libraries, we model node cost U_i^{node} as the summation of the cost for waking up an inactive PM, the cost for installing supporting libraries, and the cost for executing assigned tasks, as shown in Eq. (1).

$$U_i^{node} = \sum_{j \in N^p} ((P_s \cdot s_j + P_d \cdot c_{ij}^b) \cdot x_{ij} + P_o \cdot p_{ij} \cdot w_i) \quad (1)$$

Here, s_j indicates the sleep/awake status of a PM j. s_j is set to 1 if PM j is in sleep mode, otherwise 0. c_{ij}^b is the number of additional libraries for deploying the EC of job A_i on PM j. p_{ij} is the portion of job A_i assigned to the PM j, and w_i is the total workload of job A_i. We use a binary variable x_{ij} to indicate the EC placement. x_{ij} equals 1 if an EC of job A_i has been placed on PM j, else 0. Currently, the voltage and frequency of PMs are assumed to be fixed. In the future, we would consider dynamic voltage and frequency for further energy saving as in [6].

The link cost U_i^{link} describes the data exchange cost between the PC and ECs, which is highly related to the considered technology. Here, we consider the general scenario without assumptions on the underlying technology. We further assume that traffic between the PC and ECs is unsplitable, and always goes through the shortest path for simplicity. Then, the link cost is modeled as the product of the path length and expected traffic amount between two containers as:

$$U_i^{link} = \sum_{j \in N^p} l_{I_i j} \cdot f_{ij} \cdot x_{ij} \quad (2)$$

Here, I_i denotes the PM embedding the PC of job A_i. $l_{I_i j}$ is the length of the shortest path between PM I_i and PM j. Note $l_{I_i j}$ is known since PM I_i is determined when the job A_i arrives in a DC. We do not consider link failure and reconfiguration, so $l_{I_i j}$ is fixed for job A_i and PM $j, j \in N^p$. f_{ij} indicates the amount of traffic passing through this path, and we assume $f_{ij} = F_i \cdot p_{ij}$. Thus, ECs with heavier traffic are preferred to be embedded near the PC.

To minimize the total cost of deploying a set of M applications considering both the node cost and link cost, we have

$$\min\{\sum_i (U_i^{node} + U_i^{link})\} \quad (3)$$

while satisfying a set of constraints:

$$\forall j \in N^p : \sum_i p_{ij} \cdot C_i^a < c_j^p \quad (4)$$

Algorithm 1 Application Oriented Docker Container (AODC) based resource allocation algorithm on PC for application i

```
1: Query PMs in local region, e.g., a rack for available resources
2: Solve the optimization problem (8) subject to (9) - (12) using LP solver, e.g., CPLEX
3: if there is a feasible solution $\{x_{ij}, p_{ij}\}$ then
4:     Send a resource request to PM $j$ for the amount of $p_{ij} \cdot C_i^a$ resources, receive a response from PM $j$, and record the results $x_{ij}^r$ in $set_r$
5:     for every $result_j$ in $set_r$ do
6:         if $result_j$ is accept then
7:             Build an EC on PM $j$, record $x_{ij}$ and $p_{ij}$ into $directory_i$, and update workload and available PM
8:         else
9:             Record $p_{ij}$ into $set_u$, and update available PM
10:        end if
11:    end for
12: end if
13: if $set_r$ is null, or $set_u$ is not null then
14:     Extend searching area, e.g., a pod, and call AODC algorithm again
15: end if
```

$$\forall i \in \{1,..,M\} : \sum_{j \in N^p} p_{ij} = 1 \quad (5)$$

$$\forall i \in \{1,..,M\}, \forall j \in N^p : \alpha_i \leq p_{ij} \leq 1 \quad (6)$$

$$\forall i \in \{1,..,M\}, \forall j \in N^p : x_{ij} \geq p_{ij}, x_{ij} \in 0,1 \quad (7)$$

Constraint (4) guarantees that the assigned resources will not exceed the available resources on PM j. Constraint (5) ensures every portion of the job has been allocated. Constraint (6) checks the lower and upper bounds of each portion of the job. Constraint (7) builds the relationship between variables.

B. Algorithm

We design an AODC based resource allocation algorithm to compute a feasible solution for ECs' placement and task assignment. The AODC algorithm runs independently on each PC in a distributed manner to acquire the resources' status from a limited number of Docker engines and to make resource allocation decisions. For each PC, the AODC resource allocation problem becomes the following optimization problem:

$$\min(U_i^{node} + U_i^{link}) \quad (8)$$

subject to

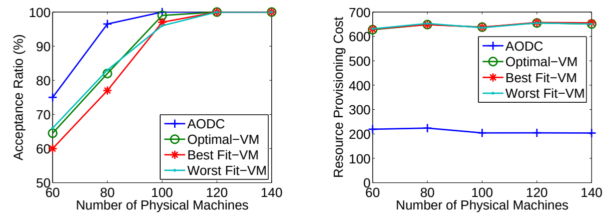
$$\forall j \in N^p : p_{ij} \cdot C_i^a < c_j^p \quad (9)$$

$$\sum_{j \in N^p} p_{ij} = 1 \quad (10)$$

$$\forall j \in N^p : \alpha_i \leq p_{ij} \leq 1 \quad (11)$$

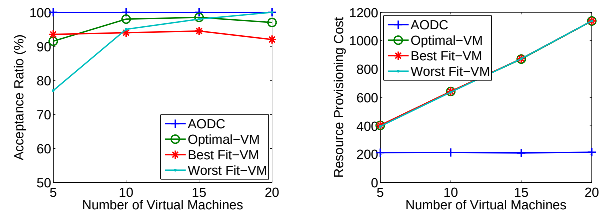
$$\forall j \in N^p : x_{ij} \geq p_{ij}, x_{ij} \in 0,1 \quad (12)$$

As shown in Algorithm 1, the PC starts querying available physical resources within a small local region, e.g., a rack



(a) Acceptance ratio (b) Total energy cost

Fig. 2. Comparison for varied number of PMs



(a) Acceptance ratio (b) Total energy cost

Fig. 3. Comparison for varied number of virtual machines

(Step (1)). Based on the available resources reported by the Docker engine on each PM, the AODC optimization problem (Step (2)) could be solved by adopting Linear Programming (LP) solver, e.g., CPLEX. Then, the PC requests physical resources based on the optimization problem's solution (Steps (4)). If the request is approved, an EC is built on the selected PM, and the PM is marked as used (Steps (6-7)). If the request is rejected, this PM is marked as infeasible and the assigned task is moved to set_u (Step (9)). When there is no feasible solution for the optimization problem, or some task has not been successfully assigned, the PC queries PMs in a larger scale, e.g., a pod, and solves the optimization problem again for not assigned tasks (Steps (13-15)).

Note that each PM is queried by a PC for at most once, and each PC does not need to query every PM in the physical network if there are enough available resources in current searching area. Thus, the AODC algorithm is communication-efficient and scalable in large scale physical networks.

IV. Evaluations

We compare the performance of AODC resource allocation algorithm with the optimal solution of VM placement achieved by CPLEX (Optimal-VM), greedy best fit algorithm for VM placement (Best Fit-VM), and greedy worst fit algorithm for VM placement (Worst Fit-VM), with respect to total energy cost and acceptance ratio using various parameter settings.

We follow the same setting in [3]. The status of each PM is randomly set as awake with the probability 0.8. The workload of each job is randomly set between [50, 80], and the total amount of internal traffic is randomly set between [5, 20]. In addition, we randomly set the type of each PM and that of each application as integers between [0, 5]. The types implicitly determine the cost for installing and configuring the supporting libraries on different PMs. Specifically, the number of additional libraries to deploy an application is quantified as the differences between the type of a PM and that of an application, while the average size of each library is set to be 25 MB, and the size of a VM is set as 100 MB.

We first vary the number of PMs from 60 to 140, while the number of VMs is fixed to 10. In Figure 2, AODC outperforms all VM placement algorithms in acceptance ratio and total cost. When only a few PMs are available, the VM placement algorithms have less possibility to find the PMs with enough resources to fit VMs with fixed requirements on the resources, while an AODC could adaptively adjust container sizes based on available resources. In Figure 3, we also examine the impact of the number of VMs when it varies from 5 to 20. Still the AODC achieves the best acceptance ratio with the least total cost among the four algorithms.

V. Conclusion

Considering the features of emergent Docker container, we have designed a framework for application oriented dynamic resource allocation and proposed a communication-efficient and scalable resource allocation algorithm to minimize the application deployment cost constrained by capacity and the service delay bound. Simulation results indicate that the proposed AODC resource allocation framework and algorithm outperform existing Hypervisor based VM placement approaches in terms of acceptance ratio and total cost.

References

[1] Z. Liu, Q. Zhang, M. Zhani, and *et al.* Dreams: Dynamic resource allocation for mapreduce with data skew. In *Int'l Symp. Integrated Network Management*, pages 18–26, 2015.
[2] Z. Xiao, W. Song, and Q. Chen. Dynamic resource allocation using virtual machines for cloud computing environment. *IEEE Trans. Parallel Distrib. Syst.*, 24(6):1107–1117, 2013.
[3] X. Guan, B. Y. Choi, and S. Song. Energy efficient virtual network embedding for green dcs using dc topology and future migration. *Computer Communications*, 69:50 – 59, 2015.
[4] J. Jiang, J. Lu, G. Zhang, and G. Long. Optimal cloud resource auto-scaling for web applications. In *Int'l Symp. Cluster, Cloud and Grid Computing (CCGrid)*, pages 58–65, 2013.
[5] X. Shi, J. Dong, S. Djouadi, and *et al.* Papmsc: Power-aware performance management approach for virtualized web servers via stochastic control. *J. Grid Computing*, pages 1–21, 2015.
[6] E. Baccarelli, D. Amendola, and N. Cordeschi. Minimum-energy bandwidth management for qos live migration of virtual machines. *Computer Networks*, 93, Part 1:1 – 22, 2015.
[7] M. Shojafar, N. Cordeschi, and E. Baccarelli. Energy-efficient adaptive resource management for real-time vehicular cloud services. *IEEE Trans. Cloud Computing*, PP(99):1–1, Apr. 2016.
[8] S. Soltesz, H. Pötzl, M. Fiuczynski, and *et al.* Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Syst. Rev.*, volume 41, pages 275–287, 2007.
[9] Docker. https://www.docker.com/.
[10] A. Vigliotti and D. Batista. Energy-efficient virtual machines placement. In *Brazilian Symp. Computer Networks and Distributed Systems (SBRC)*, pages 1–8. IEEE, 2014.
[11] X. Xu, H. Yu, and X. Pei. A novel resource scheduling approach in container based clouds. In *17th Int'l Conf. Computational Science and Engineering (CSE)*, pages 257–264, 2014.
[12] X. Chen, C. Li, and Y. Jiang. Optimization model and algorithm for energy efficient virtual node embedding. *IEEE Communications Letters*, 19(8):1327–1330, Aug 2015.