 doi:10.20944/preprints202105.0230.v1

A Resource-aware Container Management Scheme in a Heterogeneous Cluster

BY

Yiwen Chen

MS, Fordham University, 2021

 doi:10.20944/preprints202105.0230.v1

Abstract

Yiwen Chen

MS, Fordham University

A Resource-aware Container Management Scheme in a Heterogeneous Cluster

Cloud computing nowadays is not an emerging topic, and virtualization is an indispensable technology to expedite cloud computing to become the next sign of the coming Internet revolution. In real life, scientists never stop at exploring the possibilities from such technology by investigating millions of experiments and applications to enhance the quality of virtual services. However, isolated construction for the virtual machine doesn't save the technology from unwanted data volumes or insensitive processing time. Containers are created to address such problems, by distributing applications without initiating the entire virtual machine. Docker, as an important player in this game, is an open-source application of the container family. The management tool from Docker containers, Swamskit, does not take heterogeneities in either virtualized containers or physical nodes. There are different nodes in the cluster, and each node is different in configurations, resource availability, or concerning resource, etc. Furthermore, the requirements initiated by different services change all the time. The demand might be CPU-intensive (e.g. Clustering services) and also memory-intensive (e.g. Web services), or completely at the opposite. In this paper, we focus on exploring the Docker container cluster and designing, `DRAPS`, a resource-aware placement scheme, to improve the system performance in a heterogeneous cluster.

 doi:10.20944/preprints202105.0230.v1

INTRODUCTION

In this age of information explosion, millions of web users consume different services every moment, from websites searching to game playing. Virtualization then is one of the essential technologies used in the data center, to provide a stable cloud environment to ensure hardware and development efficiency. There are already different virtualization infrastructures, such as Amazon Web Service [1] and Microsoft Azure [2].

Cloud computing technology has already attracted different users around the world and helps as the main platform for many scientific projects to the fault-tolerant mechanism. There are still some open research areas in cloud computing that could be explored [3]. System virtual machines are widely adopted today. They provide the functionality to execute the entire operating system. [4] However, when in a large-scale system, numerous duplicate instances of the same OS and redundant boot volumes would be a problem for virtual machine users to suffer [3]. Virtual machines today still have the problem such as discernible performance overhead, enormous storage requirement, and limited scalability.

For the limitation issue, containers are used to solve such a problem. Containers could be used for deploying and running distributed applications without starting the entire virtual machine because the working principle of the container is to share the host operating system and physical

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1

resources. This technology is not news today, since containers are being used as basic computing units by big data processing platforms today everywhere and Unix-like operating systems have utilized such technology for more than a decade [5]–[8]. In other words, a new containerization platform, such as Docker, has already become the mainstream of application development. According to previously open-source technologies (e.g. cgroup), Docker simplifies the tooling requirement for container creation and management. The magic of containers could be shown in comparing with the physical machine and system view: While at the physical machine, containers are a normal process; however, in virtualized resource environment, containers could explore all its conveniences, not only just CPU and memory usage, also with bandwidth, ports, disk I/O, etc.

The "docker images ls" used after the docker daemon started, is a command used to list out all top-level images with each repository, tag, ID, created date, and size. A simple command helps the user realize the detailed information for each of the images which at last hold different containers. While the user needs to select a different physical machine to host the containers, the "docker run" is the command. If the users do not set up any prerequisites, the default container is always assigned to the node with the fewest running containers, and such in a system is called Spread. The principal for Spread working is to allocate tasks impartially among all the nodes, but it excludes two major problems in reality. First, there could be different nodes in a cluster. In this case, Spread still treats the task as the same while high-end servers run more processes than an off-the-shelf desktop. Second, different types of resources may be required by containers. Diverse resource demands happen because the various types of images and services are provided by different containers. For instance, a logging service such as Logstash requires larger bandwidth but Kmeans clustering requires more computational power, and these are two completely different types of resources.

In this project, we aim at the problems that Spread omit and seek to solve the problems with a new distribution approach. Then, we propose a new container placement scheme, `DRAPS`, a Dynamic and Resource-Aware Placement Scheme. The major difference between `DRAPS` and default Spread scheme is that `DRAPS` assigns tasks to different containers based on the service's simultaneous and dynamic demands. At the start of the beginning, `DRAPS` monitors all the different tasks from all

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1

working containers and then calculates to analyze the major resource type of service from each. After that, DRAPS places those containers with the same or similar resource types to the same machine. Through such allocation, DRAPS scheme could reduce and balance the resource usages among nodes intellectually. When a system displaced as overloaded by a single resource, DRAPS relocates the resource-intensive containers to other available nodes.

Two different algorithms for testing are introduced to achieve such monitoring function for DRAPS : majority vote resource demand and average resource demand. The resource usage of an image's containers is ranked as the dominant level across containers in the majority vote. The dominant level is considered by: the dominant resource per container assessment and the overall resources dominant level among all the containers. In average resource demand, the average vote sums up the total real amount usage by each container and calculates the resource to the capacity percentage by that container.

To obtain the instructive metrics for the algorithms, this project utilizes regular docker commands such as "docker stats". We believe that by better monitoring and distributing resources, DRAPS scheme could outperform the Spread scheme, and DRAPS can reduce the waste or used across nodes significantly. All of these improvements can be proven through testing and evaluation.

Our main contributions are as follows:

- We introduce the concept of dominant resource type which considers the simultaneous and dynamic demands from different services.
- We propose a new, complete container placement scheme, DRAPS, which allocates different tasks to applicable nodes and distributes resource usage in a heterogeneous cluster in balance. in a heterogeneous cluster.
- We implement DRAPS into general container orchestration tool, Swarmkit, and conduct an investigation with 18 services in 4 types. The evaluation result illustrates that DRAPS defeats the default Spread scheme by reducing usage as much as 42.6% on one specific node.

 doi:10.20944/preprints202105.0230.v1

RELATED WORK

Cloud computing and distributed systems [9], [10] has become a popular economical computing paradigm with the capability to scale computing resources on-demand and provide a simple user-friendly business model. The data storage and computing capability without direct management by users are the reasons why cloud computing becomes highly demanding in the industry. For cloud computing systems, virtualization is one of the elemental technologies to power. A huge amount of large enterprises such as Netflix and Foursquare [11] have transformed their business services from steadfast computing infrastructure to Amazon Elastic Computing Cloud (EC2) [12]. The power of VM in cloud computing is that an individual physical server could manage various VMs, and VMs can operate independently through the technology [13] [14].

Virtualization machines (VMs) is not an emerging topic, and it has been studied and tested for decades. However, VMs in reality nowadays is still struggling with the enormous performance overhead, huge storage requirement, and limited scalability. One of the main reasons behind this is VM shares computing resources in data centers. Such computing models, at last, will conduct an enormous performance overhead. From the paper of [15], the researchers even conclude different reasons leading to performance overhead of VMs. The inducements could be an individual mega data center, single-server virtualization, or different geo-distributed data centers. Recently, a lightweight

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1

virtualization operation system known as containerization is introduced and raise the popularity of the discussion or application of such technology from different aspects and on different platforms. Containerization-based virtualization outperforms VM due to the more delicate operation system and better scaling capability in the field of cloud and edge computing [16]–[27].

Such a conclusion is not from a one-time experiment. Scientists in the cloud computing area have already conducted multiple comparison experiments between virtualization and containerization for years: When back to 2007, Soltesz has already compared System V and Xen in his research [28]; Three years later, Regola et al. conducted experiments to compare KVM, OpenVZ and EC2 in [29]; In 2014, Felter et al., started to compare the difference between KVM and Docker [30] [31]. In [32], researchers conclude that containers provide an environment where applications are independent of each other and bring convenience to developers with regards to easy deployment, testing and compositions [33] [34]. Furthermore, the provisioning time from containers is much faster than in VM in many cases [35].

Containerized system is studied in many areas to explore its potential. There is an extensive performance study presented in [36], to investigate the traditional virtual machine deployments, and to compare these traditional deployments with the usage of Linux containers. The results of container evaluation which focus on overhead are equal or even higher-ranking than the results of VMs. Although such a conclusion happens and indicates that containers outperform VMs, research [37] displays that the startup latency is substantially larger than what was expected before. The larger startup latency is caused by the layered and distributed image architecture – copying package data requires most of the container startup time. Slacker is suggested by the researchers since it can remarkably decrease the startup intermission. The startup latency could be reduced or even faster if the image is already locally available, and Slacker could decrease the number of packages copying and relocate. Collaboratively sharing the image is recommended by CoMICon [38] to solve the problem. In an altered aspect, SCoPe [39] attempts to manage the provisioning time from large-scale containers: SCoPe specifies the provisioning time about system features with a statistical model, to address the provisioning strategy from the research. Furthermore,

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1

applications possess various types of behavior and resource requirements. Before giving out a conclusion between the container and traditional hypervisor-based VMs, scientists still work at analyzing and comparing the running performances of applications with VMs and Docker containers. In the field of deep learning, authors in [40]–[44] studied how efficient scheduling algorithms would benefit the training of models in a congested environment.

Different container orchestration systems achieve different requirements and also offer different services to users. To study those orchestration systems is one of the facing challenges for scientist today. A similar challenge has been faced by VM resources managers a few years before, and there are multiple researches [45]–[47] that investigate this topic in deep and provide users a detailed evolution of VM-based infrastructure.

Individual container experiments are not enough, the cluster of containers is also another important region that needs to be explored in this field. As the two most popular and important cluster management tools in today's market: Docker Swarmkit [48] and Google Kubernetes [49] are compared by studying each scalability from the researchers of [50]. In the end, using multiple min-cost max-flow algorithms lead to the result that Firmament achieves low latency in large-scale clusters. Firmament selects the identical high-quality placements as an advanced centralized scheduler, at the scale and speed are constantly connected with distribution scheduler. From another point of view, the researchers from [51] display an Ant Colony Optimization algorithm for a Docker container cluster by focusing mainly on workload scheduling. The result from this algorithm displays that workloads deployed by Ant Colony Optimization outperform those from a greedy algorithm by nearly 15%, which are both on the same host configuration. Although containers always demand a diverse requirement of resources, this algorithm cannot differentiate a huge number of containers.

Based on our previous work [52], in this paper, we experiment and explore container orchestration from the view of resource awareness. Although users can specify limits on different resources, containers still need to get involved in competing resources in physical machines. At the moment when there are different choices of images, the containers need to target different types of

 doi:10.20944/preprints202105.0230.v1

services, and such results in various demands on resources. Through investigating and analyzing the dynamic resource requirements, our research focuses on studying node placement scheme which could distribute the resource usages in a heterogeneous cluster equally.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1

BACKGROUND AND MOTIVATION

3.1 Docker Containers

A local Docker daemon participates in the system as a Docker worker machine. The command such as “docker run -it ubuntu bash” sent to containers is how a worker creates new containers. Run-time, code, system libraries, system tools, and settings, are all included in a Docker container image package. Although everything to execute is included in a Docker container image package, such software is still delicate, independent, and practical. In the macro view, every individual container takes responsibility for each specific service of an application. To be more detailed, if the application demands to scale up the specific service, it will start multiple containers by utilizing the identical image. In general, one physical machine is capable to host different applications with multiple services in a self-sustaining model.

3.2 Container Orchestration

It’s intractable to reach resilience or scalability on an individual container host while it distributing applications into a production environment. Normally, the infrastructures utilized to run containers

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1

at scale is called a multi-node cluster. SwarmKit, presented by Docker, is an open-source toolkit for container orchestration while under a cluster environment. When Swarmkit running, there are two different types of nodes in a cluster to finish the job: worker nodes and manager nodes. Worker nodes manage all the running tasks, and manager nodes manage to receive requirements from users and then accommodate the demanding state into a real cluster state. Particular demands, such as specific memory or CPU usage requirements, and user-defined labels are all applicable to initiate a docker container.

The scheduler runs on the manager firstly to synthesize the information input from users, with states of each node to apply different scheduling decisions, e.g. selecting the leading node to accomplish the following tasks. In other words, the scheduler uses filters and scheduling strategies to allocate the jobs. There are four applicable filters to choose: *ResourceFilter:* check whether the node has enough resources to perform the task; *ReadyFiler:* check whether the node is ready to schedule tasks; *ConstraintFilter:*choose the nodes that are only qualified the certain labels; *PluginFilter:* check whether the node has a particular volume plugin installed.

When multiple nodes are passing the filtering process, Docker Swarm applies the three scheduling strategies: binpack, spread (applicable currently), and random (under development based on Swarm Mode). To be more in detailed, *Binpack strategy* is to deploy the container on the most filled node in the cluster; *Spread strategy* is to deploy the container on the node with minimal running containers; Random strategy is to deploy the container into cluster inconsistently.

The default Spread strategy is the one that tries to schedule a service task according to the number of active containers on every node. Although the strategy is capable to evaluate the resources on the nodes approximately, the evaluation could not display different nodes in a heterogeneous cluster setting. The reason causes such an issue might be that in a heterogeneity cluster setting, each node has individual configurations in terms of CPU, memory, and network usage. Then, various experiences would be the result of running the same number of containers on these various nodes.

Fig 3.1 displays the average starting delay of and overall makespan of the set of TomCat Docker

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1

containers. We deploy the experiments on two different machines: M1 with 8GB memory, 4-core CPU, and M2 with 16GB memory and 8-core CPU. During the testing on each machine, M1 or M2, we find out that the more containers on that specific machine, the larger the starting delay and makespan would occur. Although M1 takes 23,67s on average to start 30 Tomcat containers and M2 takes 18.32s on average to start 40 Tomcat containers. Furthermore, when we attempt to start 80 Tomcat containers job, M1 fails to complete the task but M2 makes it.

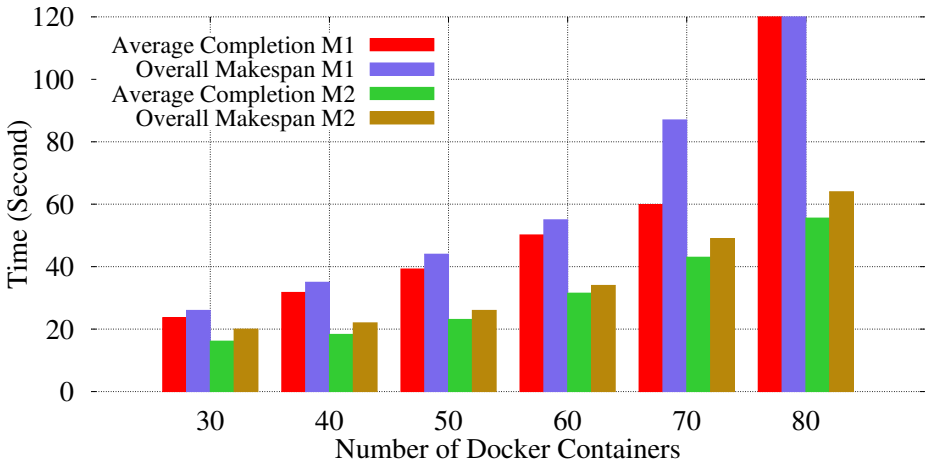


Fig. 3.1: Starting Dockers on a single machine

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1

DRAPS System

4.1 Framework of Manager and Worker Nodes

As mentioned in the above section, there are may have several managers and workers in the system. There are six hierarchical modules in a manager. The job for **Client API** is to receive commands sent from clients and to generate service objects. The job for **Orchestrator** is to take charge of the life cycle of service objects and to conduct the mechanics for load balancing and service discovery. The job for **Allocator** is to arrange network model particular distribution functionality and to assign the IP address to jobs. The job for **Scheduler** is to allocate tasks to the worker nodes. And finally, the **Dispatcher** is to interface with different worker nodes, to check every node's state, and then to assemble the **heartbeats** from nodes.

Nevertheless, a worker node takes charge of the Docker containers and updates the container states to the manager by sending the periodical heartbeat information. The job for **executor** is to manage and run the tasks which are allocated to the containers in this specific worker.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1

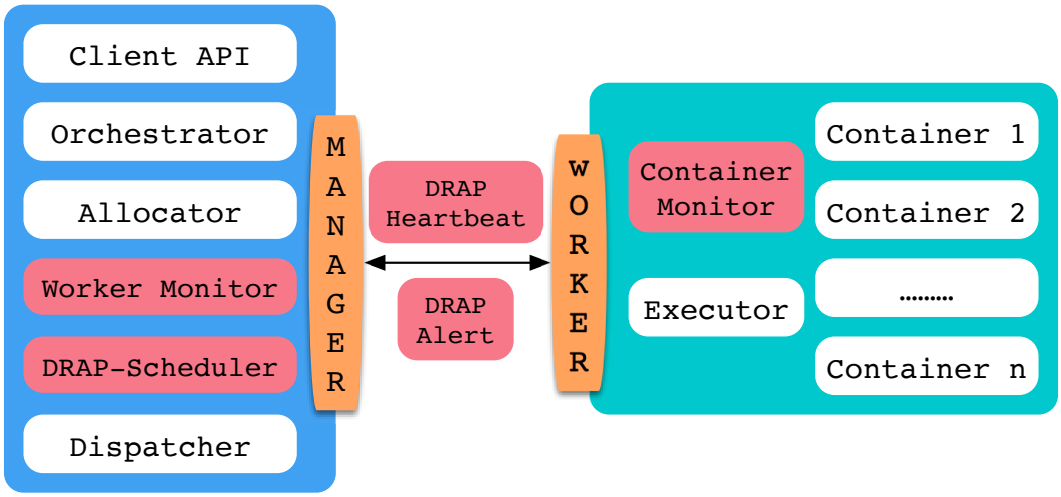


Fig. 4.1: Docker Framework with DRAPS Implementation

4.2 DRAPS modules

To present the implementation easily, we incorporate the DRAPS constituents with the contemporary framework. As displayed in Fig 4.1, the graph is designed by three main portions: a worker monitor, a container monitor in the worker nodes, and a DRAPS scheduler that is applied in manager nodes.

Work Monitor: A worker monitor is to manage the messages from worker nodes. Such a monitor supports a table for each worker and its relevant containers. The worker container will initiate tasks (e.g. transforming a resource-intensive container to another host) by investigating the data.

Container Monitor: A container monitor accumulates the run-time resources usage information from Docker containers on worker nodes. At each separate application level, the demanding resource information includes Memory, CPU usage percentage, block I/O, and network I/O. The top users will send out the average usage static report in a set-up time window, and such messages will be added into the DRAPS -Heartbeat information and then reported back to managers. At the host system level, the monitor data contains I/O wait, CPU usage percentage, reminder percentage of available memory, and bandwidth. The worker nodes would conduct a self-analysis from these data and find out the individual bottleneck. If a bottleneck is tested, a DRAPS -Alert message will be generated and reported

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1

to managers.

DRAPS-Scheduler: The DRAPS-Scheduler distributes a task to a particular node depending on the presently available resources. DRAPS-Scheduler investigates a duplicated Docker container's distinctive resource utilization, such as memory intensity, through the reports from former containers in the same services.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1

Problem Formulation

The DRAPS scheduler targets improving the container arrangement so that the obtainable resources from each worker node could be magnified. In this research, we assume that the resources for a container looking for are memory, CPU, I/O for running services, and bandwidth. The demands for these resources from a container would be different since the services and workloads in a container are dynamic. In this case, we generate a formula for the resource demands of a container as a function of time. $r_i^k(t)$ represents that kth resource requirement of the ith container at time t. The Container arrangement indicator then would be $x_{i,j} = \{0, 1\}$. $x_{i,j} = 1$ represents that the ith container is placed in the jth work node. W_j^k represents that the general amount of the kth resource in the jth work node. $\mathcal{C}$, represents the set of containers; $\mathcal{N}$, represents the work nodes; $\mathcal{K}$, represents the resources. Then, the utilization ratio of the resource k in the jth work node can be displayed as:

$$u_j^k(t) = \frac{\sum_{i \in \mathcal{C}} x_{i,j} r_i^k(t)}{W_j^k} \tag{5.1}$$

We presume that the utilization ratio of the jth work node is designated by its frequently used resource. At this case, the utilization of jth work node is $\max_{k \in \mathcal{K}} u_j^k(t)$.The frequently-used

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1

resource among all work nodes can be expressed as:

$$\nu = \max_{j \in \mathcal{N}} \max_{k \in \mathcal{K}} u_j^k(t). \tag{5.2}$$

The target for the DRAPS scheduler is to enhance the usage of all the attainable resources in each individual worker node. Then, the DRAPS scheduling problem is identified as:

$$\max_{x_{i,j}} \quad \nu \tag{5.3}$$

$$s.t. \quad \sum_j x_{i,j} = 1; \forall i \in \mathcal{C}; \tag{5.4}$$

$$u_j^k(t) \leq 1, \forall k \in \mathcal{K}, \forall j \in \mathcal{N}. \tag{5.5}$$

In E.q. (4), there is a restriction to require that each container should be arranged in one worker node. In E.q. (5), there is a restriction that to ensure the utilization ratio of any types of resources in a worker should be less than one.

5.1 Lemma 1. The DRAPS scheduling problem is NP-hardness

Proof: To prove the Lemma, we set up that the resource requirements of each container are constant over the test time as a simple case of DRAPS scheduling problem. Then, the simple case DRAPS scheduling problem is the same as the multidimensional bin packing problem, which is NP-hard [53]–[55]. Consistently, the lemma can be solved by decreasing any instance of the multidimensional bin packing to the simple case DRAPS scheduling problems. Since the simpleness, we exclude the detailed proof process in this paper.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1

DRAPS IN A HETEROGENEOUS CLUSTER

In the previous sections, we explained multiple and various modules in DRAPS and their jobs. We formulated the DRAPS scheduling problem and clarified the problem is NP-hard. In this section, we generate a comprehensive design of DRAPS with heuristic container deployment and transformation algorithms in a heterogeneous cluster, which targets to enhance the resource accessibility on each worker node and to increase the service performance by approaching the best solution of the DRAPS scheduling problem. In order to achieve the goals, DRAPS system concludes three main strategies:

- Identify dominant resource demands of containers (Resource Demands Analysis)
- Initial container placement
- Migrate a container

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1

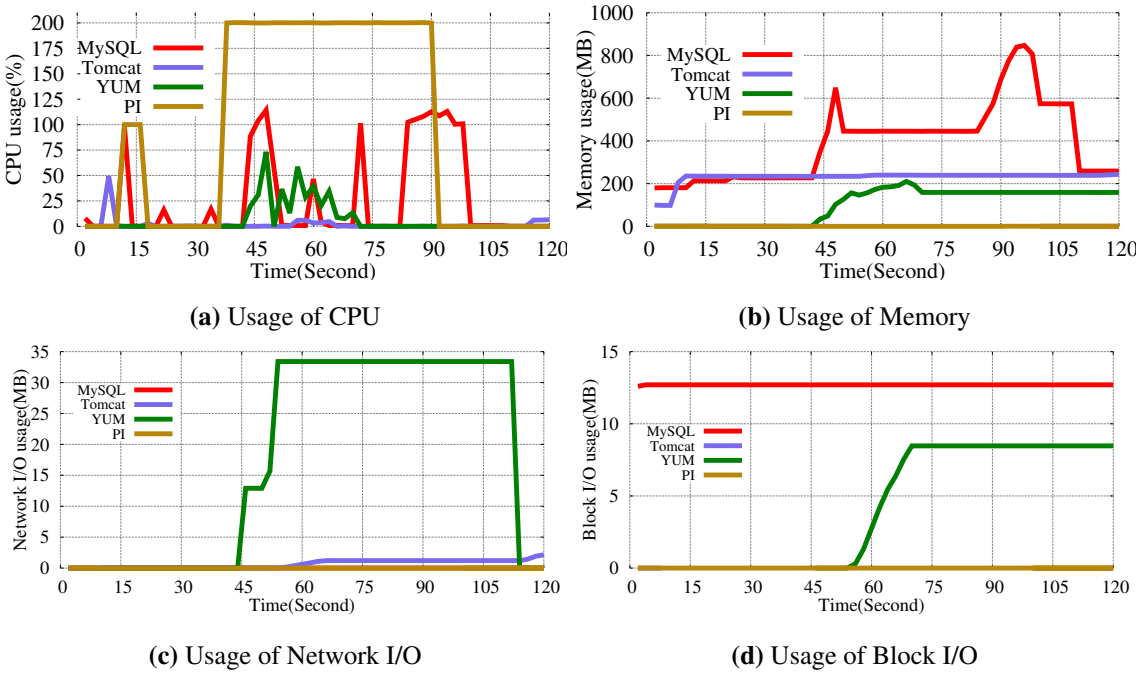
6.1 Resource Demands Analysis

The system should understand the dynamic resource demands from different containers at the start of the beginning, to improve the general resource efficiency. A container often aims at providing a specific service, such as database querying, web browsing, and data sorting. Since that different algorithms or operations will be distributed to the services, various resource demands would occur. For example, we designed experiments on NSF Cloudlab [56]] (M400 node hosted by University of Utah). The containers are launched by applying the following four images and the data is allocated through the "docker stats" command.

1. MySQL: The Relational Database Management System. Tested workloads: scan, select, count, join.
2. YUM: a Software Package Manager that installs, updates, and removes packages. Tested workloads: download and install the "vim" package.
3. Tomcat: Implementation in Java language to provide HTTP web services. Tested workloads: HTTP queries at 10/second and 20/second of a HelloWorld web page.
4. PI: a Service to Calculate PI. Tested workloads: top 3,000 digits with a single thread, top 7,500 digits with two threads.

Fig 6.1a to Fig 6.1d displays the dynamic resource demands under different workloads on the four Docker containers mentioned above. The figures explain that there are very different usage patterns on four types of resources: CPU, memory, network I/O, and block I/O. For instance, the PI container utilizes limited resources without the workload. Notwithstanding, the CPU usage boosts up to 100% for a single thread job and 200% for a two-threads job when the task arrives at the 10th and 38th second. Still, then consumption of the other three types of resources remains at very low levels. For the MySQL container, when the tested workload is added in, the CPU usage displays a split when a request is sent from clients. At the 84th second, when a "join" command that requires three tables to involve is sent, the CPU and memory usage both bounce. The reason for this is that the 'join' option

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1



(a) Usage of CPU

(b) Usage of Memory

(c) Usage of Network I/O

(d) Usage of Block I/O

Fig. 6.1: Resource demands under different workloads on four services, MySQL, Tomcat, YUM, PI.

operation requires a huge amount of calculation and copies of tables in memory. However, there are different usage trends from YUM and Tomcat containers. For example, YUM utilizes more network I/O and block I/O because of the downloading and package installation, and less CPU and memory usage. Tomcat requires very little network I/O and block I/O because of the size of the tested page but utilizes more than 200MB of memory to maintain the service. In order to assign the resource usage equally, deploying the containers with supportive demands on the same worker is essential. As displayed from the figures, there is always a main resource demand from a service in a given time, even though various types of resources are available.

In DRAPS , the first important task is to classify the dominant resource demand for each service. In the system, a manager is capable to keep track of all the containers' resource usage and distribute them in groups by associated service ID. Assume the service $s_i \in S$ includes m running containers which are stored in a set, RC_{s_i}. R_{c_i} is a vector to implicate the resources utilized by $c_i \in RC_{s_i}$. For each attribute, r_i, is the vector to clarify a type of resources, such as memory and CPU. If there are q types of resources in the system, the average resource consumption of s_i is vector R_{s_i},

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1

$$R_{s_i} = \sum_{c_i \in RC_{s_i}} R_{c_i}$$
$$=< \sum_{c_i \in RC_{s_i}} r_1/m, \sum_{c_i \in RC_{s_i}} r_2/m, ..., \sum_{c_i \in RC_{s_i}} r_q/m >$$

There is a finite amount of resources in each type on the worker nodes. A vector that includes q attributes is the resource limit, $< l_1, l_2, ..., l_q >$. The sum of vectors from workers leads to the limit of a system, $< L_1, L_2, ..., L_q >$. At this case, R_{s_i} is represented by a percentage of the general total resource in the system, for the i^{th} type, the container consumption for s_i on average is $\sum_{c_i \in RC_{s_i}} r_i/m \div L_i$. In conclusion, we make the definition to the dominant function, $DOM(s_i) = max\{\sum_{c_i \in RC_{s_i}} r_i/m \div L_i\}$ The type of a dominant resource demand of service s_i can be calculated by the $DOM(s_i)$ function, within a given time period. The running containers for s_i and the current consumption of them would change the value of $DOM(s_i)$ along with the system.

6.2 Initial Container Placement

Clients, first, utilize the command "docker run" to initiate a new container while using the Swarm cluster. The job is submitted to the manager of the cluster and the manager needs to select a worker node to host the container. As explained above, the default container placement scheme fails to detect and assign the dynamic resource usage, because the managers in Docker Swarm could not discover currently available resources without the mechanism. However, DRAPS solves the problem by utilizing the DRAPS-Heartbeat. DRAPS-Heartbeat is an improved heartbeat message that contains the states of the worker node, and also the containers' resource usage within a given time period. The resource usage includes memory, CPU, bandwidth, and block I/O. All of these data will be assembled into a table which to monitor the current attainable resource on each worker nodes and relevant containers' resource usages on the manager side.

Algorithm 1 is running on managers to allocate incoming containers and classify the images on the workers. In the system, managers maintain a list of the following items along with the system time(Line 1-2).

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1

- Active workers (w_i) in the cluster;
- Running containers (c_i) on each worker;
- Known image set that stores the images that have been used before, e.g. the containers from previously launched services.

When an incoming job arrives with new image information and a new container initialization request (Line 3), the algorithm starts selecting the best worker for it with the following two steps. Firstly, the system has to decide whether the image is known before to proceed different actions.

- When I_{new} is a known image, which means the resource usage pattern is known from previously launched containers, the algorithm finds the known image (I_j), extract the latest resource demand, $R_{I_j}(t)$, and assign it to the new container (Lines 4-6).
- When the system has never seen I_{new} before, it first adds I_{new} to know image set (Lines 7-8). Next, if the type of I_{new} is persist (e.g. database), the system assigns average resource usage of persistence containers, $R_p(t)$ to I_{new}. If it is in a non-persistence type, the system uses the average resource usage of non-persistence containers, $R_{np}(t)$ to I_{new}. In the scenarios that the type is not specified by the user, the average resource usage of all running containers, $R_{avg}(t)$, is in use (Lines 9-14).

Secondly, for each worker, we need to calculate the available resources on this worker ($A_{w_i}(t)$) (Lines 15-16). If the available resources are less than a predefined threshold, α, this worker is marked as unavailable, which means that it does not accept any new containers. The reserved α resource is used to maintain the system itself. If $W_i.Accept$ is True, it means this worker can host more containers (Lines 17-20).

Thirdly, the algorithm calculates a score for each worker based on the new container's resource demand in each resource type m, $R_{c_{new}}.DOM$ divided by the available dominant resource type, $A_{w_i}.DOM$ (Lines 21-23). This strategy attempts to satisfy the container with its dominant resource type.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1

Algorithm 1 Initial Container Placement on Manager

System Parameters:
$W_i \in W$
$C_j \in W_i$
$I_j \in I$
System Time: t;

Incoming Container: I_{new}, C_{new};
if $I_{new} \in I$ **then**
    $I_{new} = I_j$, where j is the same image in I.
    $R_{c_{new}}(t) = R_{I_j}(t)$
else
    $I_{new} \rightarrow I$
    if $I_{new}.p$ = True **then**
        $R_{c_{new}}(t) = R_p(t)$
    else if $I_{new}.p$ = False **then**
        $R_{c_{new}}(t) = R_{np}(t)$
    else
        $R_{c_{new}} = R_{avg}(t)$

for $W_i \in W$ **do**
    $A_{w_i}(t) = 1 - R_{w_i}(t)$
    if $A_{w_i} < \alpha$ **then**
        $W_i.Accept$ = False
    else
        $W_i.Accept$ = True

for $W_i \in W$ & $W_i.Accept$ = True **do**
    $W_i.Score = R_{c_{new}.DOM} \div A_{w_i.DOM}$
    Select the W_i with minimum $W_i.Score$.

6.3 Monitor the workers

When containers are running, the system has to monitor the workers to discover the abnormal resource usages to prevent a system crash. It is because the container's resource usage is dynamic and keeps changing based on its workloads and other containers running on the same worker. A module of DRAPS called container monitor takes charge of recording resource usages of host containers on each worker. Furthermore, the worker monitors the attainable resource by itself. When it detects a draining type of resources coming as a bottleneck, it will send a DRAPS alert to managers.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1

Algorithm 2 is running on the worker and responsible for monitoring the resource usage on workers. Firstly, the system initializes the parameters, worker, W_i, containers, c_i, and known images, I_j. The algorithm is running periodically with an interval t (Lines 1-2).

For all the containers that run on top of worker, W_i, the system tries to find the abnormal ones such that this container is consuming more resources than others with the same image, I_j. If it is more than a threshold, α, which is defined as the minimum system required resources, the algorithm increases the counter for this specific container, c_j (Lines 3-5). If c_j is continued consuming more resources than others for the consecutive β times, it will be inserted into the abnormal container set, AB (Lines 6-7). When the consecutive breaks, the algorithm reset the counter to 0 and remove c_j from the abnormal set. (Lines 8-10).

In the case that this worker is out of resources, the algorithm calculates the container, c_i with the highest AB_{c_i} from the abnormal list. Then, the worker sends an alert message to the manager (Lines 11-14).

Algorithm 2 Monitoring active containers on W_i

```
1:  System Parameters:
    W_i ∈ W, C_j ∈ W_i, and I_j ∈ I;
    AB: Abnormal container set;
    AB_{c_j}: Abnormal count for container c_j;
2:  Interval: t;

3:  for c_j ∈ W_i do
4:      if R_{c_j} > (1 + α) × R_{I_j} then
5:          AB_{c_j} + +
6:          if AB_{c_j} > β then
7:              AB.insert(c_j)
8:      else
9:          AB_{c_j} = 0
10:         AB.remove(c_j)

11: if A_{w_i}(t) < α then
12:     for c_i ∈ AB do
13:         Select c_i with highest AB_{c_i}
14:         Send Alert Msg (c_i)
```

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1

6.4 Migrating Containers in Alerts

Algorithm 3 Process Alert Msg from Manager

1: System Parameters: $W_i \in W$, $C_j \in W_i$, and $I_j \in I$;
2: System Time: t;

3: Function Receive Alert Msg(c_i)
4: **for** $I_j \in I$ **do**
5: Find I_i for c_i
6: $R_{c_i}(t) = R_{I_i}(t)$
7: **if** $I_i.p =$ True **then**
8: Stop c_i on w_i
9: Call Algorithm. 1 with (c_i, w_i)
10: **else if** $I_i.p =$ False **then**
11: Save c_i on w_i
12: Call Algorithm. 1 with (c_i)
13: **else**
14: Stop c_i on w_i
15: Call Algorithm. 1 with (c_i)

When the worker is running out of resource, it sends an alert message to manager asking to migrate the abnormal container.

Algorithm 3 indicates the procedure of processing the alert message from w_i.

At first, the system initialize the parameters and system time (Lines 1-2). Upon receiving the alert message that requests to reallocate c_i, it the image, I_i, which is used by this specific container. In addition, it extracts the resource usage pattern of this image $R_{I_i}(t)$ and assign it to the $R_{c_i}(t)$ (Lines 4-6). Depending on the type of this image I_i, the algorithm process it differently.

- For a persist image that keeps running to provide services, the algorithm force stop it on its current host, w_i. Then, call Algorithm-1 to reallocate it and exclude w_i from the candidate host of c_i.
- If it is a non-persist image that has a termination time, the algorithm saves the container and then call Algorithm-1 for redistribution. After the redistribution, this container can resume the job on the new host.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1

- If the type is unknown for I_i, the algorithm will force stop the container and call Algorithm-1, but not exclude the w_i.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1

Performance Evaluation

In this chapter, we evaluate the effectiveness and efficiency of DRAPS through intensive, cloud-executed experiments.

7.1 Implementation, Testbed and Workloads

7.2 Evaluation Results

We implement our new container management scheme, DRAPS, on Docker Engine v19. As described in Section 4, the main modules of DRAPS are integrated into the existing Docker Swarm framework.

The DRAPS aims to improve the resource management in a heterogeneous cluster. We build a heterogeneous cluster on Google Cloud Platform **gcp** with different configurations in CPU and memory, two major resource types in the computing node. Specifically, we utilize four different types of instances to simulate a production environment.

- Small: 2 vCPU and 2G Memory

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1

- Medium: 4 vCPU and 4G Memory
- Large: 6 vCPU and 6G Memory
- xLarge: 8 vCPU and 8G Memory

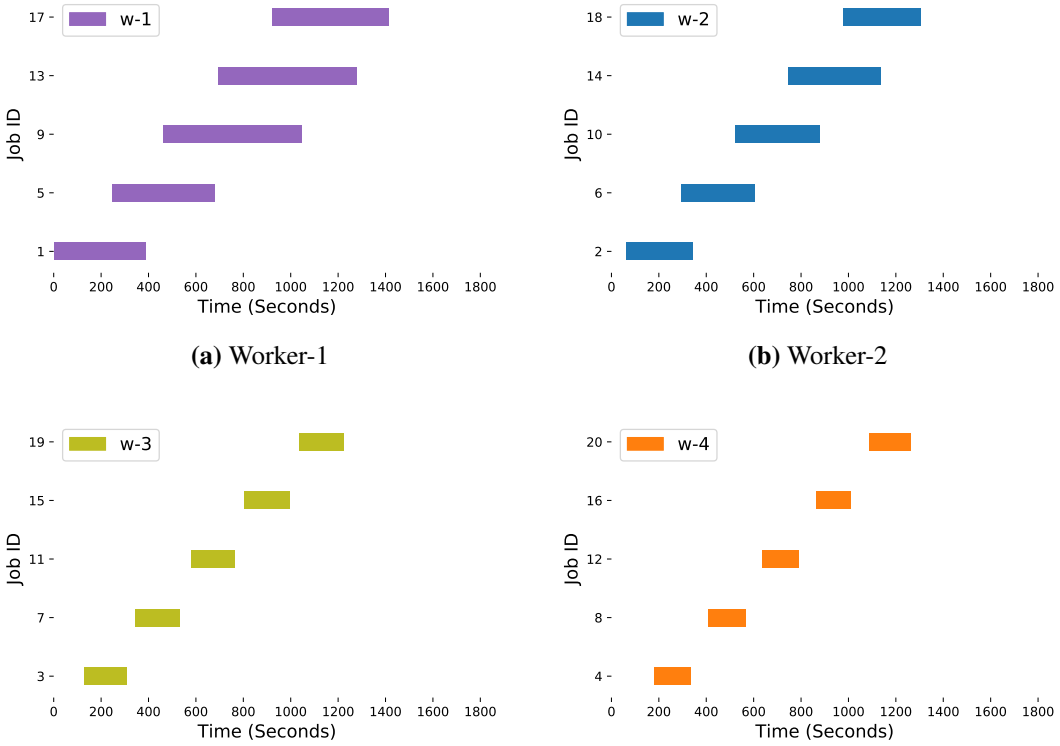
In the small-scale testing, we setup a cluster with 4 worker nodes, one of each type. Besides, there is a dedicated manager in the cluster for scheduling purpose. Therefore, the cluster is configured with 1 manager and 4 workers. In experiments on scalability, we configure the cluster with 1 manger and 8 workers that consist 2 instances of each type.

We mainly focus on two types of workloads. (1) Database services: the containers are launched to provide the database access to the clients. This is a represent type of **persistence** job such that the container will not terminate itself even with no clients using it. (2) Deep learning training services: the containers are launched to provide training services, such as Pytorch and Tensorflow. This is a **non-persistence** container, which targets on a model training job. When the job completes, the container will be terminated to release the resources. In our experiments, we have the Tensorflow, Pytorch and MongoDB as our workloads.

In the experiments, we consider two evaluation metrics. (1) Resource usage: the CPU and memory resource usage of each worker. If the resource usage is higher than the minimum system requirement, the worker might dead due to out of resources (e.g. memory) and significantly reduce the system performance. (2) Completion Time: this value only applies to deep learning services. It represents how quickly the system can complete a specific job given a shared worker.

For following experiments, we evaluates `DRAPS` with (1) Single Image Jobs: containers are launched with the same image. (2) Multiple Images Jobs: containers are launched randomly with different images. To better evaluate the system, we compare `DRAPS` with two other resource management schemes. (1) **Default**: it comes with Docker Swarm framework and utilize the standard information that contained in heartbeats , such as the number of containers and initial resource configurations. (2) **Spread**: this management scheme simply applies a rotation algorithm and select

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1



(a) Worker-1 **(b)** Worker-2

(c) Worker-3 **(d)** Worker-4

Fig. 7.1: Spread Scheme (Deep Learning): Container Distribution

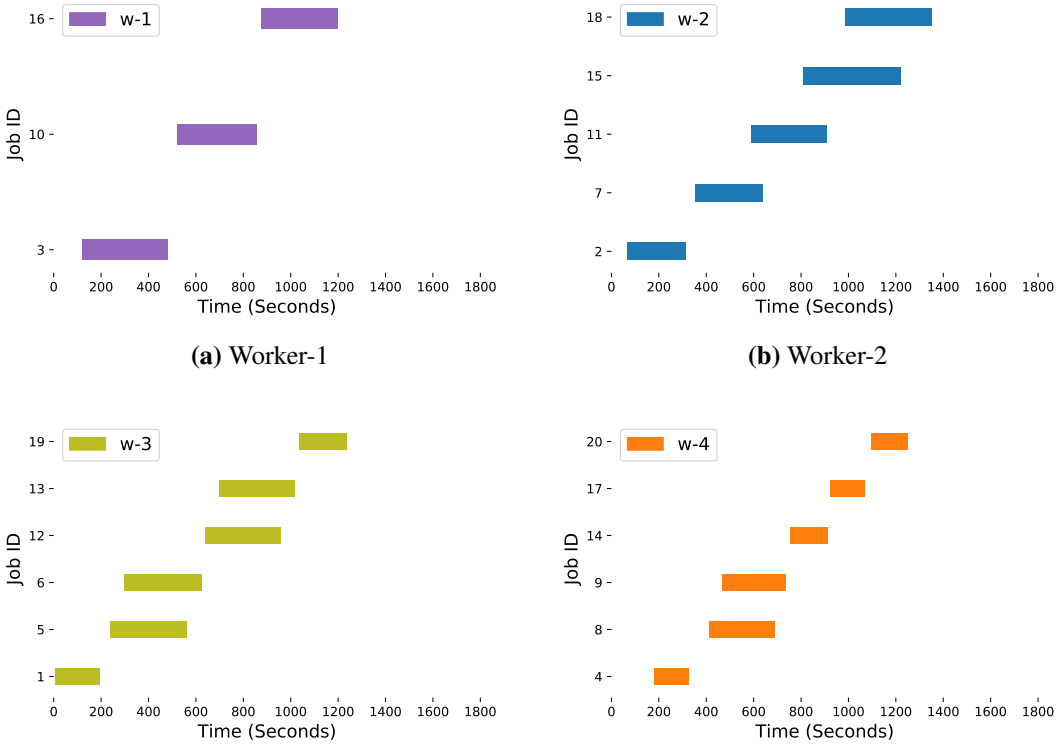
the workers one-by-one.

7.2.1 Single Image Jobs

In the single image experiment, the containers are launched to the system with a specific image. We utilize the non-persist Pytorch service to conduct the experiment that 20 containers are submitted with a 50-second interval.

Figures 7.1, 7.2 and 7.3 plot the container placement of the experiment. Please note that worker 1, 2, 3, and 4 are in different resource configurations, such that they are labeled as small, medium, large and xlarge in terms of available vCPUs and memory spaces. With these configurations in mind, we can clearly see that Spread scheme, shown in Figure 7.1, is not working well. Since Spread scheme fails

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1



(a) Worker-1 **(b)** Worker-2

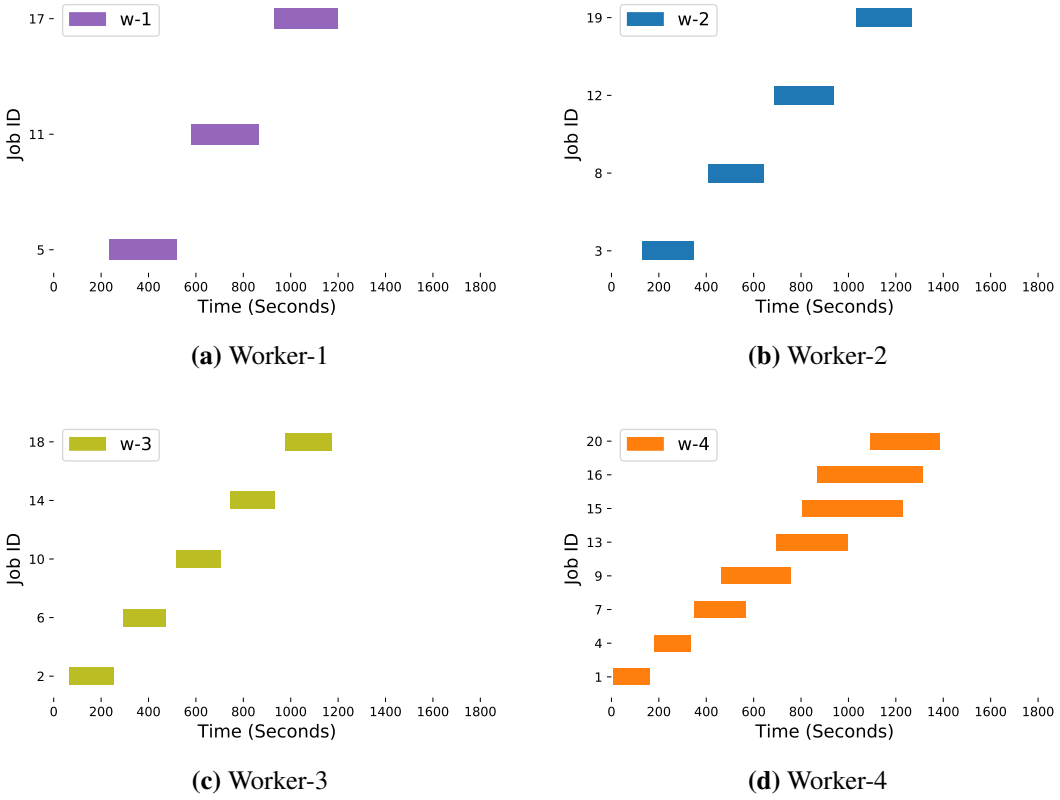
(c) Worker-3 **(d)** Worker-4

Fig. 7.2: Default Scheme (Deep Learning): Container Distribution

to consider the heterogeneity within the cluster, it treats all four workers equally and evenly distribute the workloads. As we can see on Figure 7.1 that each worker is assigned 5 containers. While it may work well in a homogeneous cluster, this scheme is not for a heterogeneous environment.

Comparing with Default and DRAPS, both of them are able to conducting the scheduling with respect to the resource availability. Figures 7.2 and 7.3 illustrate the container distribution of the 20 deep learning jobs. As the worker with least resources, Worker 1 under both schemes gets 3 containers. This distribution helps prevent overwhelming the worker 1. With Default scheme, it does not distinguish the differences between worker 3 and worker 4. It assigns 6 jobs each to them, even though worker 4 has more resources. Running with DRAPS, however, it can dynamically discover the available resource and make decisions in real-time to choose the best host. In this experiment, DRAPS assigns 3, 4, 5, 8 to each of the workers.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1



(a) Worker-1 **(b)** Worker-2

(c) Worker-3 **(d)** Worker-4

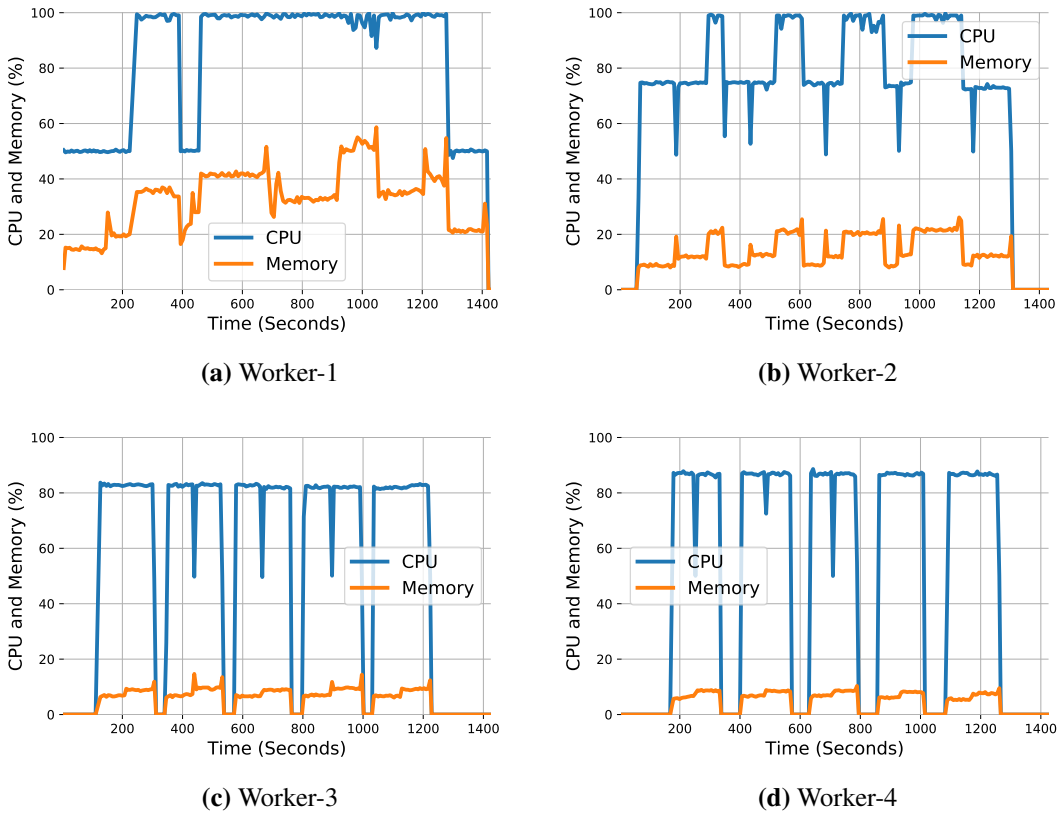
Fig. 7.3: DRAPS Scheme (Deep Learning): Container Distribution

Figure 7.5, 7.4 and 7.6 present the resource usages of CPU and memory on each worker. As we can see on Figure 7.4a, with Spread scheme, worker 1 is under a 100% resource usage on CPU in most of the time. This is 5 running container on it. Comparing with Default and DRAPS, the CPU usage is around 50% for most of the time, which lead to more stable performance on this worker.

When comparing worker 3 in Default and DRAPS, Figure 7.5c and 7.6c demonstrate worker 3 in DRAPS has a CPU utilization rate at around 81% for most of the time and with Default, the value jumps to 100% from 220s to 620s and 680s to 970s. DRAPS shows a more stable performance. While considering worker 4 in Figure 7.5d and 7.6d, DRAPS has more jitters on the figure, this is because the scheme tries to adjust the resource usage and reserved a minimum amount of resources for the system itself.

Figure 7.7 shows the completion time of the 20 deep learning jobs. For the Spread scheme, the

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1



(a) Worker-1 **(b)** Worker-2

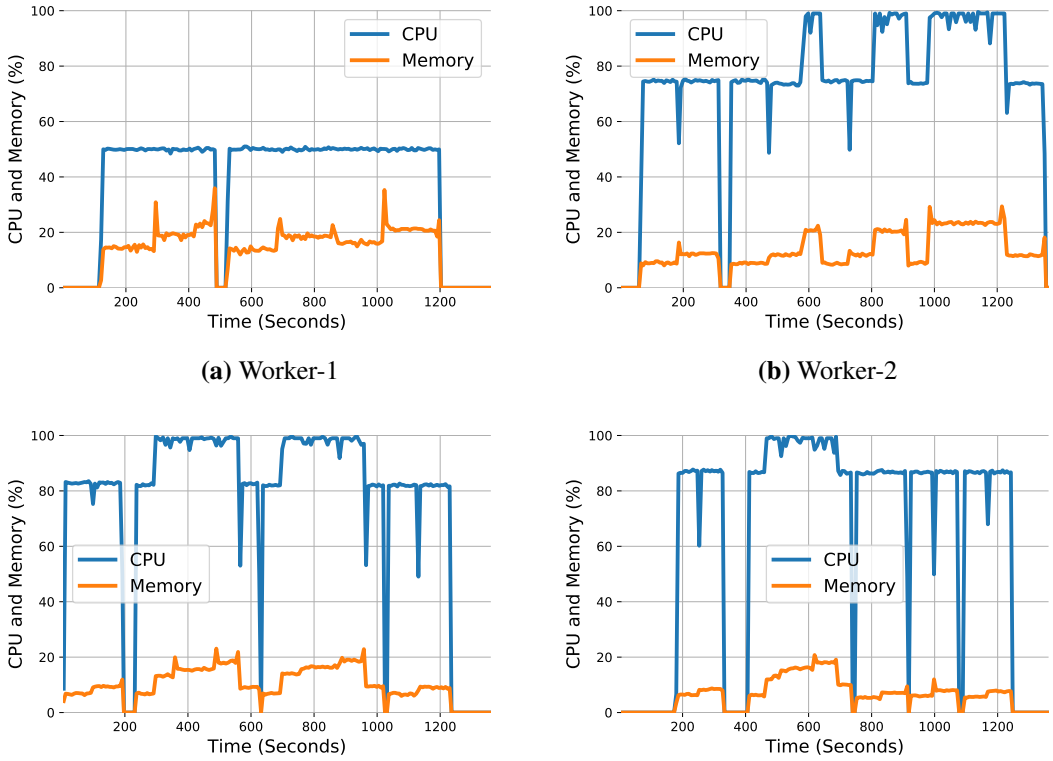
(c) Worker-3 **(d)** Worker-4

Fig. 7.4: Spread Scheme (Deep Learning): CPU and Memory Resource Usage

Job-1, Job-5, Job-9, Job-13 and Job-17 are the slowest ones. This is because that Spread scheme equally distributes the workload such that the worker with smallest resources is assigned 5 jobs. Both DRAPS and Default consider the resource configurations on each worker, however, Default fails to detect the dynamic resource availabilities. In our experiments, 13 out of 20 jobs record a shorter completion time, that is 65% of the jobs get improved. Additionally, DRAPS improves the average completion time. The average completion times for DRAPS, Default and Spread are 247.1s, 283.5 and 292.9s. The proposed system, DRAPS, achieves 12.7% and 15.7% improvement on completion time.

Next, we conduct the experiment with a database image, MongoDB. The clutser is fed with 10 database containers. Inside each container, it has a 450 MB database. Upon successfully starting, the container executes the built-in queries to simulate the workloads. Comparing with deep learning

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1



(a) Worker-1 **(b)** Worker-2

(c) Worker-3 **(d)** Worker-4

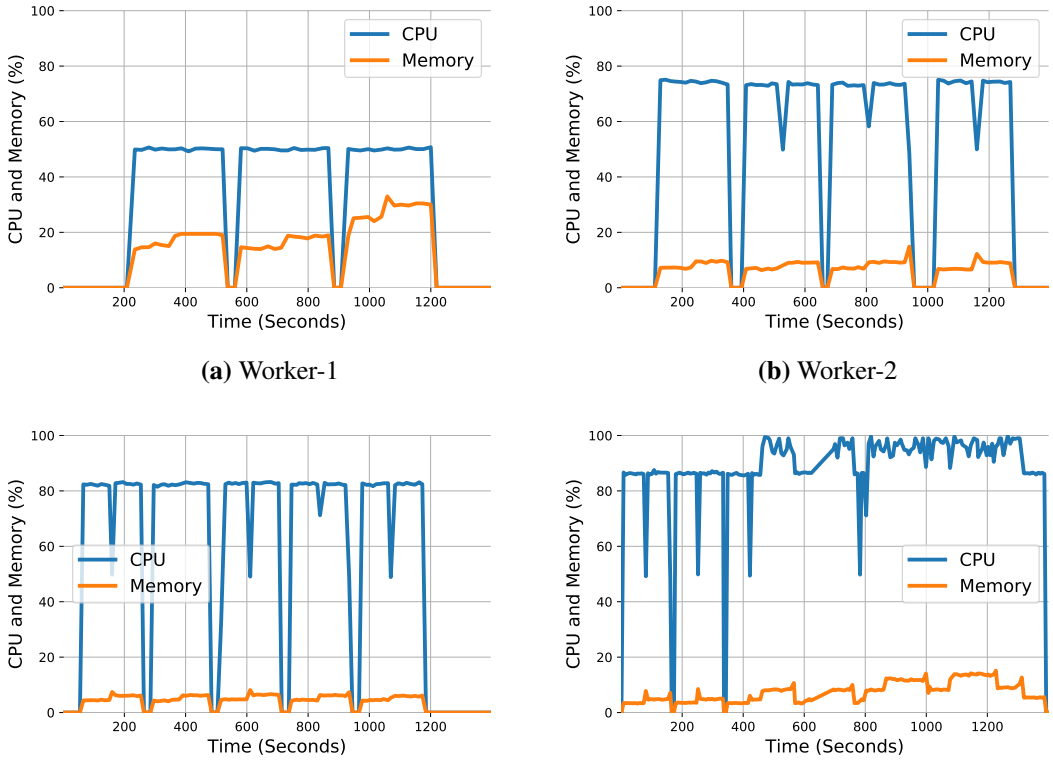
Fig. 7.5: Default Scheme (Deep Learning): CPU and Memory Resource Usage

applications, the database services are persistence and would not terminate themselves after the queries are done. In the experiment, we terminate the containers after the queries in the 10^{th} job complete.

Figures 7.8, 7.9, 7.10 present the CPU and memory usage of each worker. It is clearly noticed that, with Spread scheme, the worker 1 and worker 2 dead. This is due to the fact that the worker is out of memory when the third container is started on it at 150s and 200s. When the third container submitted, the memory usage jumps to 92%, which is a dangerous level for the Docker system itself. In this case, the system force stopped all the running containers and restart the Docker daemon to protect itself.

With Default scheme, the system is able to aware that worker 1 has minimum memory space. Therefore, it allocates less workload to worker 1, with only 2 containers are assigned to it. However,

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1



(a) Worker-1 **(b)** Worker-2

(c) Worker-3 **(d)** Worker-4

Fig. 7.6: DRAPS Scheme (Deep Learning): CPU and Memory Resource Usage

Default scheme fails to detect the differences between worker 2, worker 3 and worker 4. In the experiment, worker 2 crashes when t he third one is started and worker 3 crashes when the fourth database container tries to start the service. Since DRAPS is able to dynamically detect the resource availabilities, it distributes the containers with the respect of the latest workloads on the workers. As we can see that DRAPS protects all four workers from crashing by fully utilize worker 4, which has the most memory spaces in the cluster.

7.2.2 Multiple Images of Jobs

Next, we evaluate the performance of DRAPS with multiple image types, persistence (e.g. database and non-persistence (deep learning). In previous experiments, the containers are submitted to the system with a fixed interval, which controlled by the administrator. In the production environment,

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1

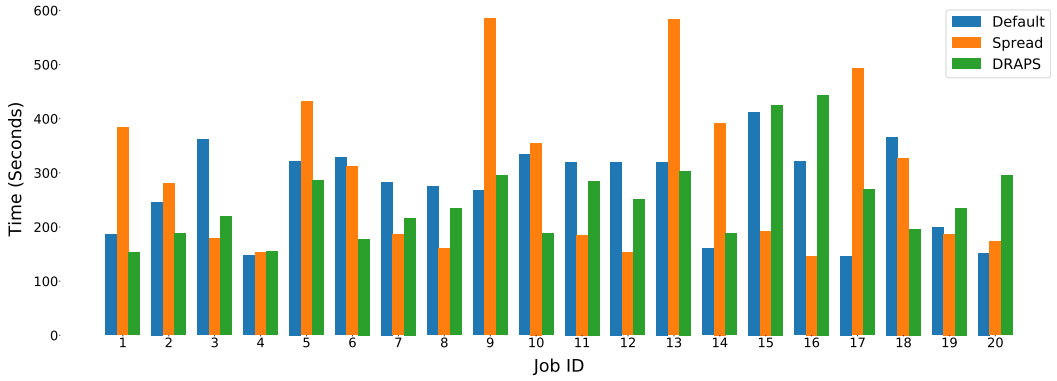
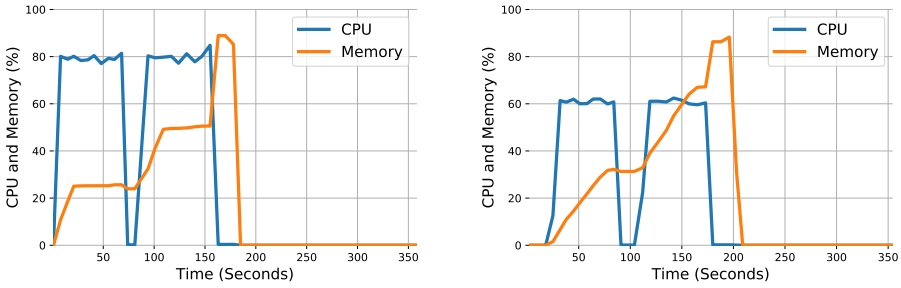


Fig. 7.7: Completion Time (Single Image Type)

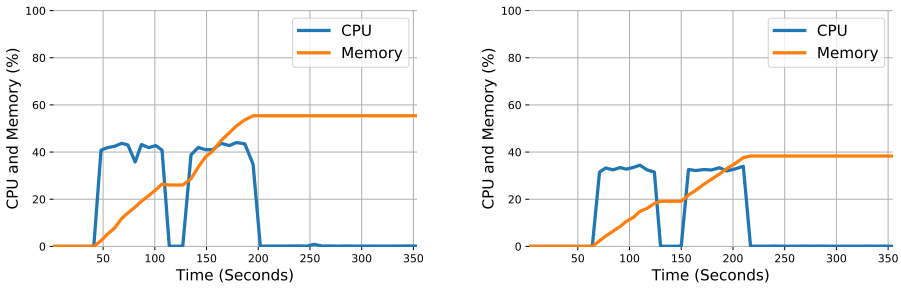
the submission interval is based on the workload and controlled by the clients. Therefore, in the experiment with multiple image types, we randomly generate the submission interval and create more challenges to the scheduling system. In this experiment, we randomly select the images for 15 jobs and submit them in the cluster. In these 15 jobs, 7 of them are deep learning applications and 8 of them are database services. The submission intervals are [0, 46, 36, 44, 66, 19, 7, 28, 2, 85, 15, 8, 52, 5, 56].

Figures 7.11 and 7.12 plot the resource usages on each worker. We terminate worker after the last job done on it. As we can see from Figures 7.11 that worker 1 crashes at 600s. This is because Default scheme fail to recognize the memory shortage on worker 1. While deep learning applications would not cause crashes since they are CPU-intensive, the database services crash the system with memory requests. With Default scheme, the system did not fully utilize worker 4, which has the largest amount of memory spaces. The highest memory usage rate on worker 4 is 38%, far from an alarming level. With DRAPS, however, the system is able to handle all 15 jobs without crashing any workers. For example, the memory usage of worker 1 is at 49% it only host 1 database and 2 deep learning containers. The Figure 7.13 plots the completion time of the deep learning jobs. It demonstrates that DRAPS is able to reduce the completion time for 6 of out 7 jobs.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1

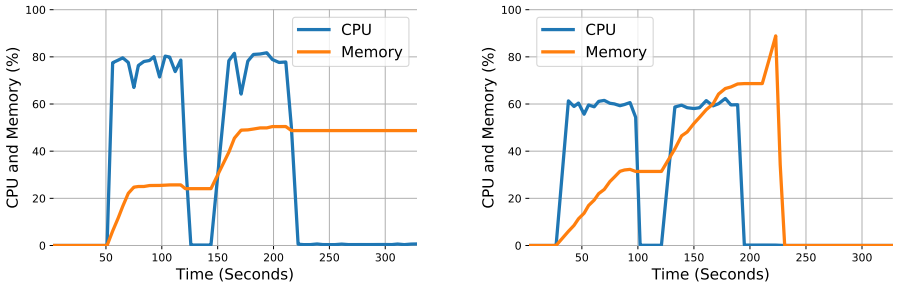


(a) Worker-1 **(b)** Worker-2

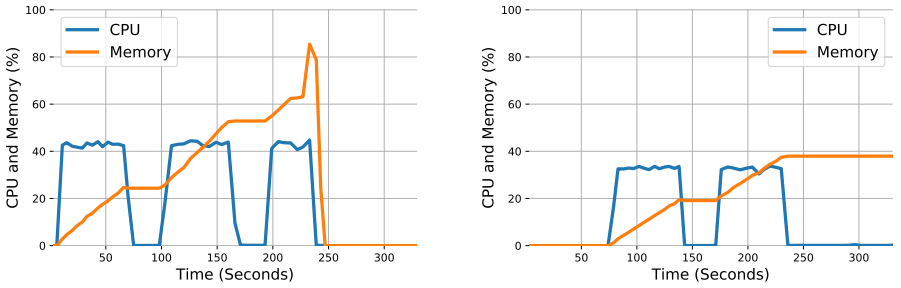


(c) Worker-3 **(d)** Worker-4

Fig. 7.8: Spread Scheme (Database): CPU and Memory Resource Usage



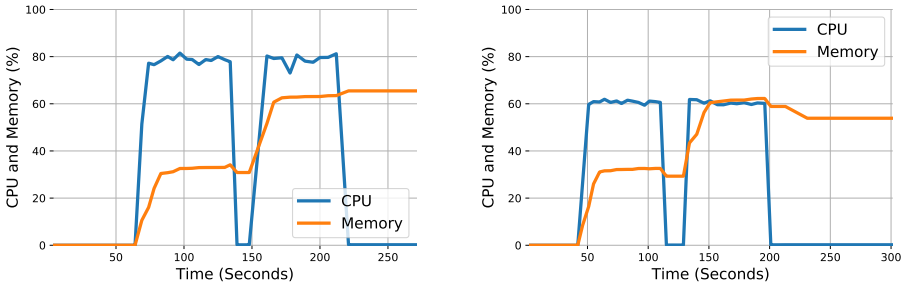
(a) Worker-1 **(b)** Worker-2



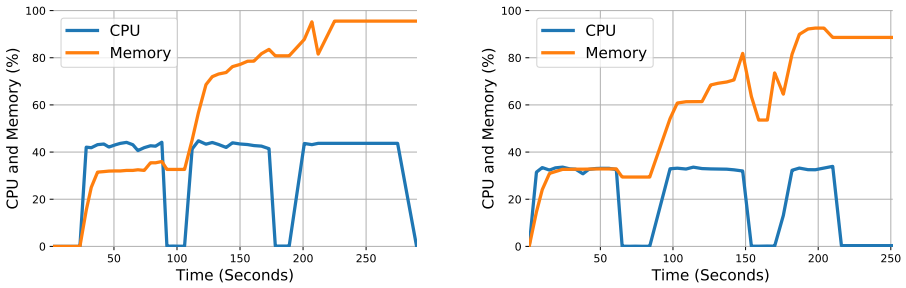
(c) Worker-3 **(d)** Worker-4

Fig. 7.9: Default Scheme (Database): CPU and Memory Resource Usage

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1

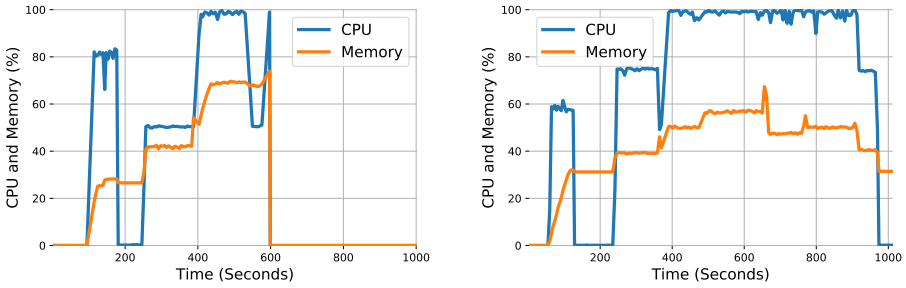


(a) Worker-1 **(b)** Worker-2

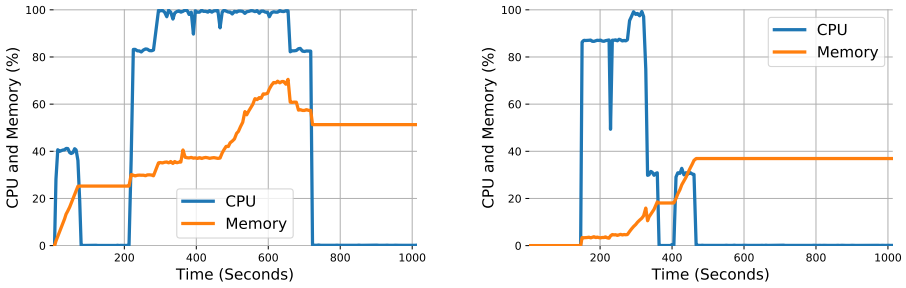


(c) Worker-3 **(d)** Worker-4

Fig. 7.10: DRAPS Scheme (Database): CPU and Memory Resource Usage



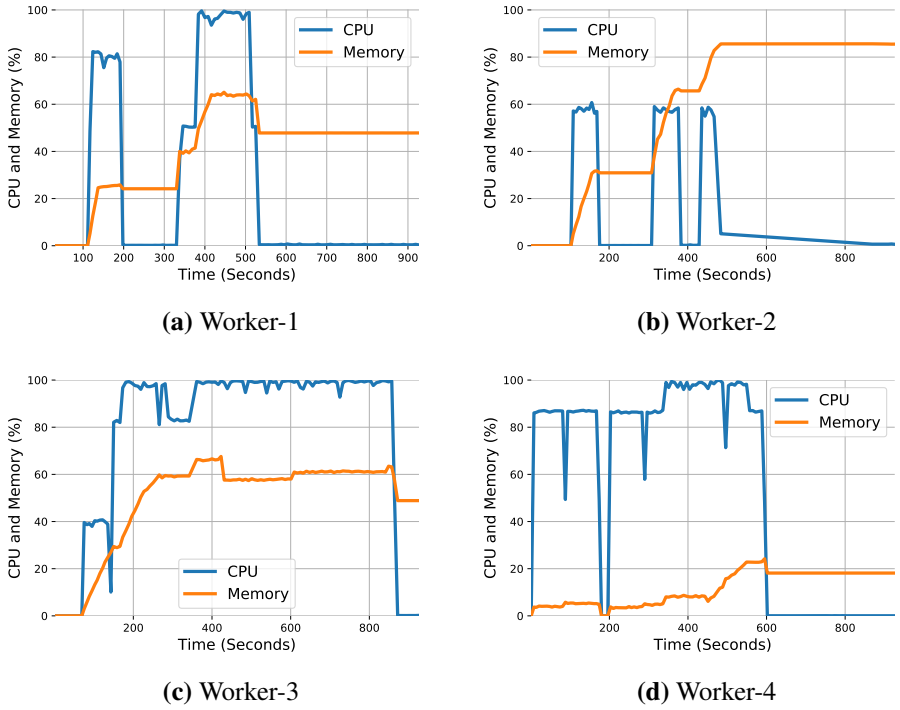
(a) Worker-1 **(b)** Worker-2



(c) Worker-3 **(d)** Worker-4

Fig. 7.11: Default Scheme (Multiple Image Types): CPU and Memory Resource Usage

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 May 2021 doi:10.20944/preprints202105.0230.v1



(a) Worker-1 **(b)** Worker-2

(c) Worker-3 **(d)** Worker-4

Fig. 7.12: DRAPS Scheme (Multiple Image Types): CPU and Memory Resource Usage

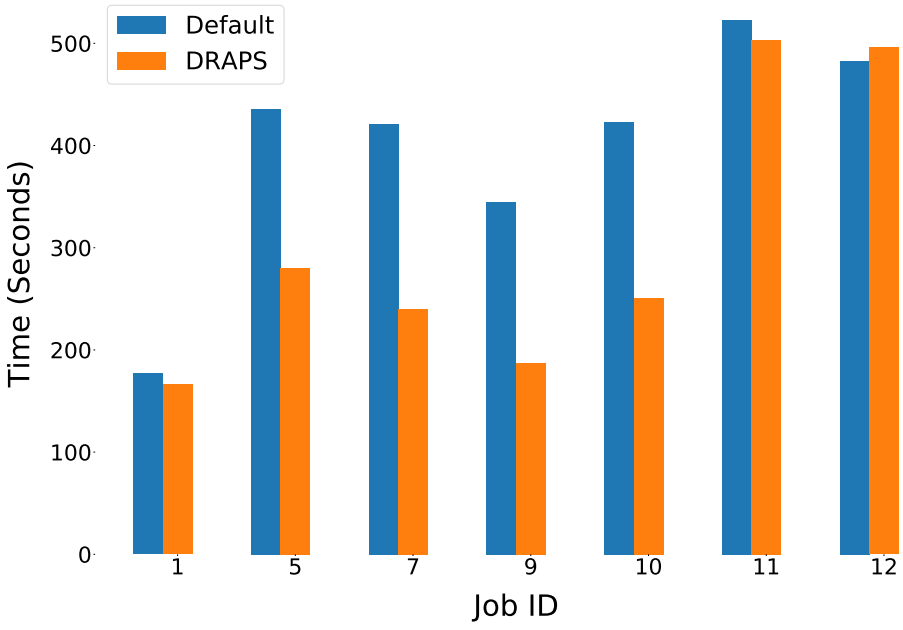


Fig. 7.13: Completion Time (Multiple Image Types)

 doi:10.20944/preprints202105.0230.v1

Conclusion

This thesis studies the container management strategy in a heterogeneous cluster. The current system only targets on homogeneous environment, which considers the computing nodes with the same resource availabilities. In our system, We aim to distribute containers to the worker nodes with the best available resources and consider the dynamic resource usage patterns from the running jobs.

We propose DRAPS that considers various resource demands from containers and current available resources on each node. DRAPS works with both persistence (e.g. database) and non-persistence (deep learning) workloads. It is implemented on Docker Swarm platform and evaluated with extensive experiments on Google Cloud Platform. The results show a significant improvement on the system stability and scalability when comparing with the Default and Spread strategies. Additionally, DRAPS reduces the completion time for deep learning containers.

 doi:10.20944/preprints202105.0230.v1

Bibliography

[1] *Amazon web service*, https://aws.amazon.com/.

[2] *Microsoft azure*, https://azure.microsoft.com/en-us/.

[3] V. Medina and J. M. García, "A survey of migration mechanisms of virtual machines," *ACM Computing Surveys (CSUR)*, vol. 46, no. 3, p. 30, 2014.

[4] *Virtual machine*, https://en.wikipedia.org/wiki/Information_explosion.

[5] J. Wang, Y. Yao, Y. Mao, B. Sheng, and N. Mi, "Fresh: Fair and efficient slot configuration and scheduling for hadoop clusters," in *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, IEEE, 2014, pp. 761–768.

[6] ——, "Omo: Optimize mapreduce overlap with a good start (reduce) and a good finish (map)," in *Computing and Communications Conference (IPCCC), 2015 IEEE 34th International Performance*, IEEE, 2015, pp. 1–8.

[7] J. Wang, T. Wang, Z. Yang, Y. Mao, N. Mi, and B. Sheng, "Seina: A stealthy and effective internal attack in hadoop systems," in *Computing, Networking and Communications (ICNC), 2017 International Conference on*, IEEE, 2017, pp. 525–530.

[8] Y. Mao, J. Wang, and B. Sheng, "Skyfiles: Efficient and secure cloud-assisted file management for mobile devices," in *2014 IEEE International Conference on Communications (ICC)*, IEEE, 2014, pp. 4202–4207.

[9] L. Cheng, Y. Wang, Q. Liu, D. H. Epema, C. Liu, Y. Mao, and J. Murphy, "Network-aware locality scheduling for distributed data operators in data centers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 6, pp. 1494–1510, 2021.

[10] X. Chen, L. Cheng, C. Liu, Q. Liu, J. Liu, Y. Mao, and J. Murphy, "A woa-based optimization approach for task scheduling in cloud computing systems," *IEEE Systems Journal*, vol. 14, no. 3, pp. 3117–3128, 2020.

[11] "Amazon.com. customer success," Powered by the AWS Cloud. [Online]. Available: [Online]. Available: `http://aws.amazon.com/solutions/case-studies/`.

[12] "Amazon.com," Amazon Elastic Compute Cloud (Amazon EC2). [Online]. Available: [Online]. Available: `http://aws.amazon.com/ec2/`.

[13] R. P. Goldberg, "- 128) * 64 + (' - 128) ‰'Survey of virtual machine research,'' Computer, vol. 7, no. 9," *p*, vol. 34, 1974.

[14] M. Rosenblum and T. Garfinkel, "- 128) * 64 + (' - 128) ‰'Virtual machine monitors: Current technology and future trends,'' computer, vol. 38, no. 5," *p*, vol. 39, 2005.

[15] F. Xu, F. Liu, H. Jin, and A. V. Vasilakos, "Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions," *Proceedings of the IEEE*, vol. 102, no. 1, pp. 11–31, 2014.

[16] X. Wang, Y. Chen, Z. Wang, Y. Qi, and Y. Zhou, "Secpod: A framework for virtualization-based security systems," in *Proceedings of the 2015 USENIX Annual Technical Conference*, 2015, pp. 347–360.

[17] A. Acharya, Y. Hou, Y. Mao, M. Xian, and J. Yuan, "Workload-aware task placement in edge-assisted human re-identification," in *2019 16th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, IEEE, 2019, pp. 1–9.

[18] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos, "Efficiently handling skew in outer joins on distributed systems," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, IEEE, 2014, pp. 295–304.

 doi:10.20944/preprints202105.0230.v1

[19] ——, "Qbdj: A novel framework for handling skew in parallel join processing on distributed memory," in *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*, IEEE, 2013, pp. 1519–1527.

[20] L. Cheng and T. Li, "Efficient data redistribution to speedup big data analytics in large systems," in *High Performance Computing (HiPC), 2016 IEEE 23rd International Conference on*, pp. 91–100.

[21] Y. Mao, V. Green, J. Wang, H. Xiong, and Z. Guo, "Dress: Dynamic resource-reservation scheme for congested data-intensive computing platforms," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, IEEE, 2018, pp. 694–701.

[22] H. H. Harvey, Y. Mao, Y. Hou, and B. Sheng, "Edos: Edge assisted offloading system for mobile devices," in *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, 2017.

[23] J. Bhimani, J. Yang, Z. Yang, N. Mi, Q. Xu, M. Awasthi, R. Pandurangan, and V. Balakrishnan, "Understanding performance of i/o intensive containerized applications for nvme ssds," in *Performance Computing and Communications Conference (IPCCC), 2016 IEEE 35th International*, IEEE, 2016, pp. 1–8.

[24] J. Bhimani, Z. Yang, M. Leeser, and N. Mi, "Accelerating big data applications using lightweight virtualization framework on enterprise cloud."

[25] M. Tang, J.-Y. Zhao, R.-f. Tong, and D. Manocha, "Gpu accelerated convex hull computation," *Computers & Graphics*, vol. 36, no. 5, pp. 498–506, 2012.

[26] P. Du, J.-Y. Zhao, W.-B. Pan, and Y.-G. Wang, "Gpu accelerated real-time collision handling in virtual disassembly," *Journal of Computer Science and Technology*, vol. 30, no. 3, pp. 511–518, 2015.

[27] A. Acharya, Y. Hou, Y. Mao, and J. Yuan, "Edge-assisted image processing with joint optimization of responding and placement strategy," in *2019 International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom)*

and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), IEEE, 2019, pp. 1241–1248.

[28] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "March)," *Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In ACM SIGOPS Operating Systems Review (Vol.*, vol. 41, no. 3, pp. 275–287,

[29] N. Regola and J. C. Ducom, "November). recommendations for virtualization technologies in high performance computing," in *Cloud Computing Technology and Science*, C. Second, Ed., International Conference on . IEEE, 2010, pp. 409–416.

[30] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers. technology," vol. 28, p. 32, 2014.

[31] Y. Zhou, B. Subramaniam, K. Keahey, and J. Lange, "Comparison of virtualization and containerization techniques for high-performance computing,"

[32] M. A. Rodriguez and R. Buyya, "Container-based cluster orchestration systems: A taxonomy and future directions," *Software: Practice and Experience*, vol. 49, no. 5, pp. 698–719, 2019.

[33] R. Morabito, J. Kjällman, and K. M. H. vs. lightweight virtualization: a performance comparison, in *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E) IEEE; p*, 386–393, 2015.

[34] C. Ruiz, E. Jeanvoine, and L. Nussbaum, "Performance evaluation of containers for hpc," in *European Conference on Parallel Processing Springer; p*, 813–824, 2015.

[35] S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros, and B. R. ContainerCloudSim, "An environment for modeling and simulation of containers in cloud data centers," *Software: Practice and Experience*, vol. 47, p. 4, 2017.

[36] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, IEEE, 2015, pp. 171–172.

[37] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast distribution with lazy docker containers.," in *FAST*, vol. 16, 2016, pp. 181–195.

[38] S. Nathan, R. Ghosh, T. Mukherjee, and K. Narayanan, "Comicon: A co-operative management system for docker container images," in *Cloud Engineering (IC2E), 2017 IEEE International Conference on*, IEEE, 2017, pp. 116–126.

[39] A. Hegde, R. Ghosh, T. Mukherjee, and V. Sharma, "Scope: A decision system for large scale container provisioning management," in *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on*, IEEE, 2016, pp. 220–227.

[40] W. Zheng, M. Tynes, H. Gorelick, Y. Mao, L. Cheng, and Y. Hou, "Flowcon: Elastic flow configuration for containerized deep learning applications," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–10.

[41] Y. Fu, S. Zhang, J. Terrero, Y. Mao, G. Liu, S. Li, and D. Tao, "Progress-based container scheduling for short-lived applications in a kubernetes cluster," in *2019 IEEE International Conference on Big Data (Big Data)*, IEEE, 2019, pp. 278–287.

[42] W. Zheng, Y. Song, Z. Guo, Y. Cui, S. Gu, Y. Mao, and L. Cheng, "Target-based resource allocation for deep learning applications in a multi-tenancy system," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, IEEE, 2019, pp. 1–7.

[43] Y. Mao, Y. Fu, S. Gu, S. Vhaduri, L. Cheng, and Q. Liu, "Resource management schemes for cloud-native platforms with computing containers of docker and kubernetes," *arXiv preprint arXiv:2010.10350*, 2020.

[44] Y. Mao, W. Yan, Y. Song, Y. Zeng, M. Chen, L. Cheng, and Q. Liu, "Differentiate quality of experience scheduling for deep learning applications with docker containers in the cloud," *arXiv preprint arXiv:2010.12728*, 2020.

[45] B. Jennings and R. Stadler, "Resource management in clouds: Survey and research challenges," *Journal of Network and Systems Management*, vol. 23, p. 3, 2015.

[46] S. S. Manvi and S. GK., "Resource management for infrastructure as a service (iaas) in cloud computing: A survey," *Journal of Network and Computer Applications*, vol. 41, 2014.

[47] M. ZÁ, "Allocation of virtual machines in cloud data centers - a survey of problem models and optimization algorithms," *ACMComputing Surveys (CSUR)*, vol. 48, p. 1, 2015.

 doi:10.20944/preprints202105.0230.v1

[48] *Docker swarmkit*, `https://github.com/docker/swarmkit`.

[49] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[50] I. Gog, M. Schwarzkopf, A. Gleave, R. N. Watson, and S. Hand, "Firmament: Fast, centralized cluster scheduling at scale," Usenix, 2016.

[51] C. Kaewkasi and K. Chuenmuneewong, "Improvement of container scheduling for docker using ant colony optimization," in *Knowledge and Smart Technology (KST), 2017 9th International Conference on*, IEEE, 2017, pp. 254–259.

[52] Y. Mao, J. Oak, A. Pompili, D. Beer, T. Han, and P. Hu, "Draps: Dynamic and resource-aware placement scheme for docker containers in a heterogeneous cluster," in *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, IEEE, 2017, pp. 1–8.

[53] N. Bansal, A. Caprara, and M. Sviridenko, "Improved approximation algorithms for multidimensional bin packing problems," in *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, 2006, pp. 697–708.

[54] A. Meyerson, A. Roytman, and B. Tagiku, "Online multidimensional load balancing," in *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques: 16th International Workshop, APPROX 2013, and 17th International Workshop, RANDOM 2013, Berkeley, CA, USA, August 21-23, 2013. Proceedings*, P. Raghavendra, S. Raskhodnikova, K. Jansen, and J. D. P. Rolim, Eds. Berlin, Heidelberg, 2013, pp. 287–302.

[55] S. Im, N. Kell, J. Kulkarni, and D. Panigrahi, "Tight bounds for online vector scheduling," in *Proceedings of the 2015 IEEE 56th Annual Symposium on Foundations of Computer Science (FOCS)*, ser. FOCS '15.

[56] *Nsf cloudlab*, `https://cloudlab.us/`.