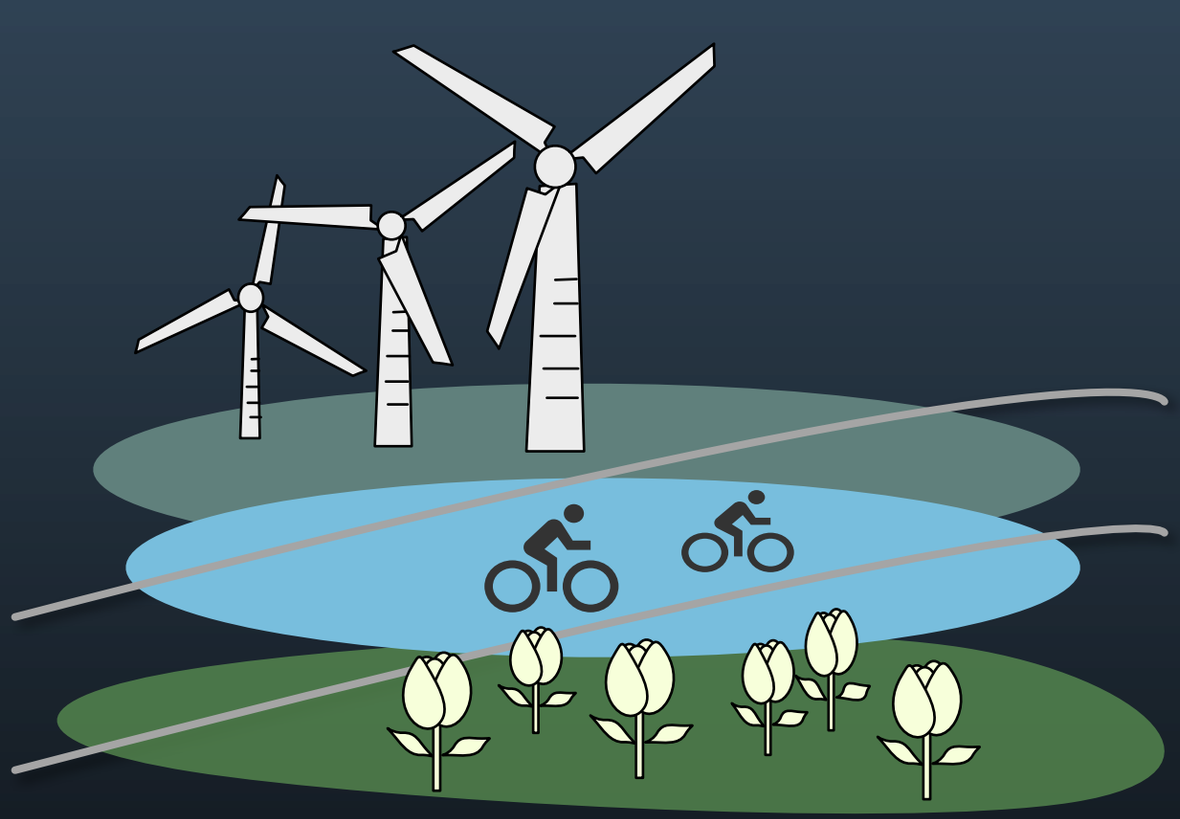


Resource Scheduling for Quality-Critical Applications on Cloud Infrastructure

Yang Hu

Resource Scheduling for Quality-Critical Applications on Cloud Infrastructure

Yang Hu

Resource Scheduling for Quality-Critical Applications on Cloud Infrastructure

Yang Hu

The research received funding from China Scholarship Council.

This research was also supported by the European projects of Horizon 2020 Programme under grant agreement No. 643963 (SWITCH), No. 825134 (ARTICONF), No. 654182 (ENVRIPLUS), No. 824068 (ENVRIFAIR), and No. 676247 (VRE4EIC).

The author also would like to thank ExoGENI and DAS5 for providing testbeds to conduct the experiments.

Cover design by Ludan Tan
Printed and bound by Ipskamp Printing, Enschede
ISBN: 978-94-028-1713-3

Resource Scheduling for Quality-Critical Applications on Cloud Infrastructure

Academisch Proefschrift

ter verkrijging van de graad van doctor aan de
Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. ir. K.I.J. Maex
ten overstaan van een door het College voor Promoties ingestelde
commissie, in het openbaar te verdedigen in
de Agnietenkapel
op woensdag 23 oktober 2019, te 12:00 uur

door

Yang Hu

geboren te Hunan

Promotiecommissie

Promotor:		
	Prof. dr. ir. C.T.A.M. de Laat	Universiteit van Amsterdam
Co-promotor:		
	Dr. Z. Zhao	Universiteit van Amsterdam
Overige leden:		
	Prof. dr. ir. A. Iosup	Vrij Universiteit Amsterdam
	Prof. dr. R. Prodan	University of Klagenfurt
	Prof. dr. R.V. van Nieuwpoort	Universiteit van Amsterdam
	Prof. dr. P.W. Adriaans	Universiteit van Amsterdam
	Dr. A.S.Z. Belloum	Universiteit van Amsterdam
	Prof. dr. D. Li	National University of Defense Technology

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

The words of truth are always paradoxical.

Laozi

Contents

1 Introduction

1.1 Motivation

As a major disruptive technology in the last decade, cloud computing enables an organization to effectively outsource its IT resource management to a third party at any level between application software and underlying infrastructure. The virtualized, elastic, and on-demand resource-as-a-service offered by the cloud have made a great impact on applications in both industry and academia. Instead of provisioning the maximum capacities in advance, cloud environments allow applications to start from the small and increase resources when the resource demand rises [153]. According to the state of the cloud report of RightScale in 2019[1], 79% of companies' workloads nowadays are in cloud environments. The cloud market is poised to grow by around 27.5% and expected to reach $1,250 billion by 2025[2].

The cloud virtualization technologies allow software developers to encapsulate software systems with their runtime system contexts as self-contained virtual machines or containers, which can significantly improve the efficiency of delivering software products to customers and reduce the cost of application deployment. Thus, the software components, especially for data-intensive and computing-intensive applications, can be flexibly scaled across distributed cloud environments in an elastic way. Due to these advantages, cloud computing has been widely used in various fields, e.g., enhancing the computing capacity of robotics [140], storing and streaming virtual and augmented reality (VR/AR) content [74], and accelerating machine learning and data mining [100]. Furthermore, the applications with high-performance requirements or critical timeliness constraints also start to migrate to clouds. Typical examples like disaster early warning, business collaborating or live event broadcasting are

[1] https://www.rightscale.com/lp/state-of-the-cloud

[2] https://www.researchandmarkets.com/reports/4039738/global-cloud-computing-market-analysis-and-trends

highlighted in the EU H2020 SWITCH project[3].

Moreover, cloud computing has also generated a significant impact on the software lifecycle. By automating the pipeline of software testing, integration, and deployment, cloud infrastructure potentially enables software development and operations (DevOps) to be seamlessly integrated. In this way, online applications or services can be continuously operated and accessible.

Besides the benefits that cloud computing as we can see above brings to the applications and software DevOps, we also have to face the challenges of achieving the desired Quality of Service (QoS) or Quality of Experience (QoE) required by cloud applications. For instance, in the environment and earth observations, near-realtime data has to be processed within a specific time window in order to be useful by a certain modeling framework. The response time of sensor data processing jobs is crucial for the cloud infrastructure to serve their user communities, as indicated in the EU H2020 ENVRIPLUS[4], ENVRI-FAIR[5] and VRE4EIC[6] projects. Such kind of cloud applications typically involve distributed and parallel components (e.g., geo-distributed sensors) to handle massive and complex tasks in acquiring and processing data. Depending on their performance characteristics, the application components often require specific resource configurations (e.g., a combination of CPU, memory, and storage) from the underlying cloud infrastructure to be running properly. Moreover, application-specific properties, such as network traffic among collaborative services and dependency of batch tasks, also require careful treatment by cloud resource schedulers for achieving the desired system performance. When the scale of cloud applications and the complexity of cloud infrastructure grow, as we can see from the use cases of the Internet of Things (IoT) and social applications in the EU H2020 ARTICONF project[7], effective cloud resource scheduling mechanisms become extremely important.

To efficiently schedule cloud resources, recent work has explored various directions. Google's Borg system [139] is a cluster manager that schedules tasks based on a scoring model across heterogeneous resources, which tries to reduce the amount of stranded resources. Microsoft's Apollo system [40], employing a distributed and loosely coordinated framework, schedules a task on the machine with the earliest estimated completion time. Alibaba's Fuxi [154] system, a resource management and job scheduling system, tries to schedule a task on the machine which meets both the multi-resource demands and the application's locality requirements. Fairness is another essential aspect in allocating multiple types of resources to users with heterogeneous demands [60, 143, 144]. For the scheduling speed, distributed schedulers [113] and hybrid schedulers [48, 84]

[3]EU H2020 SWITCH: https://www.switchproject.eu
[4]EU H2020 ENVRIPLUS: https://www.envriplus.eu
[5]EU H2020 ENVRI-FAIR: https://www.envri.eu/envri-fair
[6]EU H2020 VRE4EIC: https://www.vre4eic.eu
[7]EU H2020 ARTICONF: https://www.articonf.eu

have been proposed to schedule massive and diverse workloads on large-scale clusters efficiently. For the scheduling quality, resource efficiency [67, 31] and application performance [117, 68] are the two main factors considered when designing a scheduler.

While many research efforts have been devoted to the cloud resource scheduling problems, a number of important research questions still remain, particularly for the applications with quality-critical requirements. As cloud can provide elastic and on-demand computing resources for applications to handle rapidly changing requirements, many time-critical applications tend to be deployed in clouds [156]. However, support for deploying complex and distributed applications with critical time constraints is lacking in current cloud computing platforms. As the adoption of cloud services increases and the scale of cloud applications grows, modern cloud platforms have to deal with a large number of application requests. In many cases, they are concurrent requests at the same time. It becomes more challenging when those concurrent tasks have high performance requirements and diverse resource requirements on the cluster with heterogeneous machines. Furthermore, besides the performance and resource requirements, schedulers have to consider different application-specific properties, such as traffic demands between collaborative services and dependency constraints of batch tasks, in scheduling decisions for being able to handle diverse cloud applications.

We are thus motivated to tackle these scheduling challenges for handling the increasingly growing complexity in both cloud applications and cloud infrastructure. We formulate our main research question as:

RQ: How can we efficiently schedule resources to satisfy quality-critical requirements of diverse applications on cloud infrastructure?

1.2 Virtualization and Cloud Computing

The virtualization technologies provide the ability to partition a computer system as multiple isolated spaces at a specific level (e.g., hardware level, or operating system level). It essentially enables effective resource isolation, flexible resource management, and on-demand resource provisioning. Based on virtualization, cloud computing is able to deliver software, computing capacity, data storage, and other types of resources, to remote users via three typical service models [103]: Infrastructure as a service (IaaS), Platform as a service (PaaS) and Software as a service (SaaS).

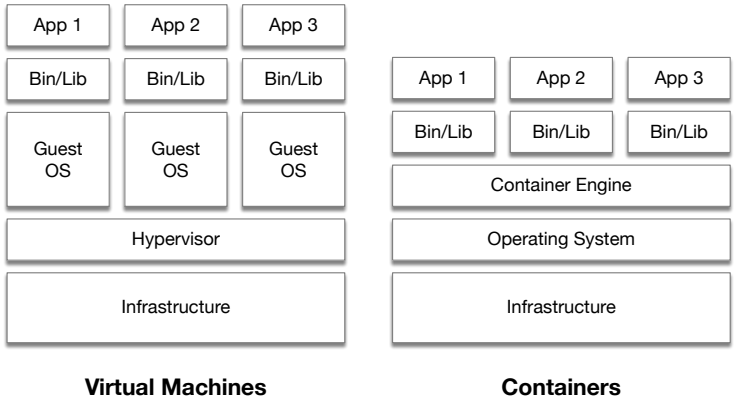


Figure 1.1: Comparison of the architecture between virtual machine and container

1.2.1 Virtual Machine

Virtual machine (VM) is one of the most important virtualization technologies for cloud computing. In IaaS model, cloud providers mainly rely on virtual machine technologies to provide virtual infrastructure to users. A virtual machine is an emulation of a computer system, which is backed by the physical resources of the underlying infrastructure. Figure 1.1 shows the architecture of virtual machines. Each virtual machine runs its own guest operating system (OS) and applications on top of the hypervisor. A hypervisor is a hardware virtualization technique that creates and runs virtual machines on the physical machines, which expands the hardware capability to do more simultaneous work, improves the security for each independent instance, and enhances the ability to run OS-dependent applications. The most commonly adopted hypervisor technologies in clouds are KVM [10], XEN [35] and VMware ESXi [18]. To efficiently configure and schedule virtual machines on a computer cluster, several management platforms, such as OpenStack [15] and OpenNebula [107], are leveraged in clouds.

1.2.2 Container

Compared to virtual machine, container is a lightweight virtualization technology for running software in isolated and portable virtual environments. Figure 1.1 shows the difference of the architecture between virtual machine and container. Container technology leverages OS-level virtualization to create virtual environments. Unlike hardware virtualization, OS-level virtualization partitions the operating system by using system kernel features. For instance,

namespaces [14] are one feature of the Linux kernel, which are used to provide the isolation environment for containers. Every aspect of a container that runs in a namespace can only see or use the resources in the same namespace. Another example is cgroups [4], which provides resource limitation for different containers. Depending on the OS-level virtualization, the containers that are running on the same machine share the host operating system kernel and can only contain the necessary libraries and binaries in their virtual environments.

Container engine is a software that helps users to build and containerize applications. There are many competing container engines including Docker [105], CoreOS rkt [16], Mesos Containerizer [12]. Docker has attracted the most attention and achieved great success recently. One of the main reasons for the success is that Docker provides an efficient way to package up an application with all its dependencies as a Docker image, which is widely adopted by many companies and tends to be the standard container image to deliver containerized applications. Therefore, the applications can be assured to run on any machine without considering the customized settings and dependent libraries. It thus makes it easier to migrate local applications to clouds.

As containers are emerging as a key technology for encapsulating applications and managing tasks in cloud computing, we assume all application components are containerized, and the cloud infrastructure is configured to support container deployment in our thesis. Although we conduct the research based on the containerized applications, we believe that the proposed solutions of our work can be easily extended to apply to other common resource scheduling problems in clouds.

1.3 Container Orchestration System

Efficiently managing and scheduling containerized applications on a computer cluster requires flexible and sophisticated systems. The container orchestration system, such as Docker Swarm [17] and Google Kubernetes [9], is commonly adopted to coordinate the placement and deployment of containers among multiple host machines. In this section, we first introduce the typical architecture of the container orchestration system, as shown in Figure 1.2, that we empirically used to build our work on.

For a common container orchestration system, there are two kinds of nodes: *master node* and *worker node*. The master node manages the overall state of the cluster, which includes handling requests from users, running a control loop that watches the state of the cluster and makes changes to move the current state towards the desired state, and scheduling the requested containers. The worker nodes manage and execute containers on a machine as dictated by the scheduling decisions from master nodes. In order to run applications on the

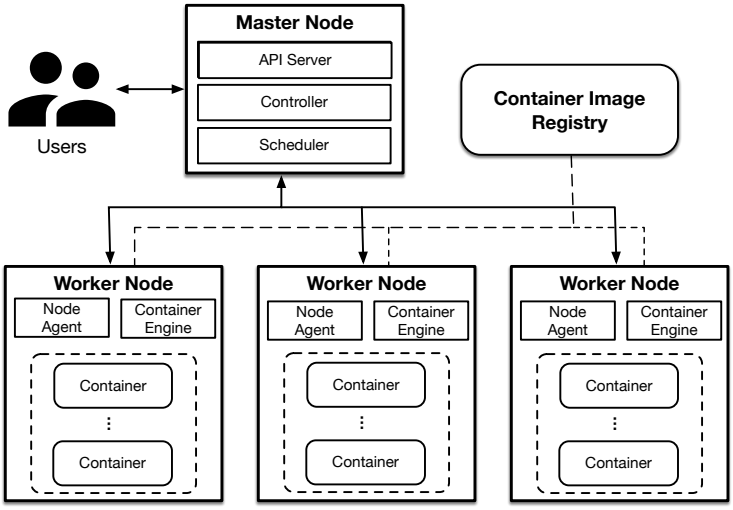


Figure 1.2: The typical architecture of the container orchestration system

cluster, users have to first upload their container images (a package of the applications) to the worker nodes or the container image registry, such as Docker Hub [19] and Google Container Registry [7]. A worker node would fetch the container image from the registry if it does not have the image of the container which is scheduled to run on it. When a user submits a deployment request to the master node, the API server receives the request and puts it into a queue. The scheduler continuously fetches a request from the queue and assigns the requested container to a worker node according to its scheduling strategies. The corresponding worker node receives the scheduling decision via the node agent and starts executing the requested container. At the runtime, the controller watches the state of the cluster. If some containers fail, it tries to restart or recreate the failed containers for maintaining the desired state.

In this thesis, we mainly focus on the scheduler, where we strive to place containers on the best possible machines. In general, a high-quality scheduling solution should consider following properties:

- **Application performance** is the primary requirement of users. To ensure the desired application performance, a scheduling solution has to meet different types of requirements. Multi-resource demand of the application, such as CPU and memory demand, is the fundamental requirement that has to be satisfied. Service-level agreements (SLAs) define the level of service users expect from the cloud provider. Thus, SLAs (e.g., deadlines) should be guaranteed for the cloud applications.

Moreover, application-specific properties also need to be well dealt with for effectively handling different applications.

- **Resource efficiency** is another essential aspect of the scheduling quality. Most computer clusters in cloud operate at very low resource utilization which greatly reduces cost-effectiveness ratio [36, 119]. This is the case even for the clusters that use resource management frameworks to enable cluster sharing across different applications. For instance, according to the analysis of two production cluster traces: Google Cluster [8] and Alibaba Cluster [1], only around half of the CPU and memory are utilized [123]. Therefore, the scheduler should be designed to improve the resource efficiency of computer clusters.

- **Scheduling time** indicates how fast a scheduler can generate a scheduling solution for the application. The scheduling time is critical for time-critical requests and low-latency interactive data processing, which are sensitive to the scheduling delay. Especially, when a large number of requests arrive simultaneously, the scheduling time would have a more severe impact on the overall performance.

- **Scalability** is defined as the ability of a scheduler to handle user requests for large-scale clusters. The scale of a modern cluster can range from several thousand machines to ten thousand or several hundred thousand machines because of a large number of applications simultaneously running in the system [139]. Hence, a scheduling solution with good scalability is quite necessary.

1.4 Quality-Critical Requirements and Cloud Applications

Quality-critical cloud applications refer to the cloud applications with quality-critical requirements. These applications typically demand a high standard of QoS (e.g., emergency response time of disaster early warning systems) or QoE (e.g., smooth delivery of ultra-high definition audio and video for live events) [156] to achieve their expected business value and outstanding social impact.

1.4.1 Quality-Critical Requirements

Quality-critical requirements are the requirements that the cloud system has to satisfy to achieve the desired application performance. Resource requirements, such as CPU demand and memory demand, are the fundamental requirements of the applications deployed in clouds, which have to be met by the underlying infrastructure for running the applications properly. Time constraints are

one of the most important quality requirements needed to achieve expected business value, particularly for the time critical applications. The common form of the time constraints is a deadline which is a time before which a particular task of the application must be finished or a particular request must be done. For instance, a cloud disaster early warning system has to report the emergent event within a certain time period to save lives and property in case of disaster [161]. Another important quality requirements are the speed requirements, which are specified to minimize the completion time (i.e., the sooner, the better). For instance, a stream processing platform needs to process the data streams as soon as possible to achieve high throughput. Besides, several other essential requirements are also imposed on the cloud platforms for maintaining high-quality performance, such as network traffic requirements and robustness requirements.

1.4.2 Cloud Applications

In this thesis, we generally classify cloud applications into two categories: Service-based Application and Batch Job.

Service-based Application

Service-based applications are the applications that are composed by a number of services made available over networks to offer diverse and flexible functionalities. A service is typically a long running or constantly running program. At runtime, a service is waiting for requests. When a request arrives, the service processes the request and gives a response. As the microservice architecture is emerging as a main architectural style choice in the service oriented software industry [27, 131], many developers are trying to divide a traditional monolithic system into small and modular services that work cohesively to provide the business function. Each service can run in its own process and communicates with other services through communication protocols, such as HTTP and TCP. For instance, an online shopping application could be basically divided into product service, cart service and order service. The microservice architecture is becoming increasingly popular because it can greatly improve the productivity, agility, scalability, and resilience of the application. However, it also brings challenges. When deploying a service-based application in clouds, the scheduler has to carefully schedule each service, which may have diverse resource demand, on distributed compute clusters. Furthermore, the network communication between different services needs to be handled well, as the communication conditions significantly influence the quality of service, especially the response time of service.

Batch Job

A batch job is a set of tasks processed in batch mode. A typical example of batch jobs is a data processing job which consists of a number of tasks that convert raw data to meaningful and usable form. A batch job usually can be modeled as a Directed-Acyclic Graph (DAG), where each vertex represents a task and edges encode precedence constraints. A task in a DAG relies on the outputs of the precedent tasks and cannot be started until all its required inputs are in place. Big data processing frameworks, such as Apache Hadoop [20], Apache Tez [120] and Apache Spark [152], and distributed scientific application frameworks [47] are developed to compile user scripts into DAG jobs. For a DAG job, the job completion time (the difference between the job arrival time and the completion time of the last task) is the main concern of users. To efficiently schedule batch jobs, the schedule has to ensure that independent tasks run in parallel as many as possible, and no tasks that are ready to run are blocked if there are available resources. However, existing cluster schedulers ignore this challenge. They fetch tasks as soon as the tasks become ready and schedule them in an arbitrary order without considering the job characteristic.

1.5 Research Questions

As mentioned in section 1.1, our main goal is to efficiently schedule resources to satisfy quality-critical requirements of diverse applications on cloud infrastructure. Different resource scheduling strategies lead to different application performance and resource efficiency on cloud infrastructure. In this thesis, we investigate application-centric scheduling approaches to improve the performance of diverse cloud applications. Specifically, we address the following research questions.

RQ1: How can we effectively deploy distributed applications with critical time constraints in clouds? Time critical applications tend to be deployed in clouds due to the rapid elasticity of cloud resources. A common deployment procedure is to transmit application packages from the application provider to the cloud and install the application there. Thus, users need to manually deploy their applications into clouds step by step with no guarantee for deployment time. Therefore, we first investigate scheduling mechanisms for meeting the deployment deadlines of time critical applications in clouds. This is also very important for adaptive applications that must automatically and seamlessly scale, migrate, or recover swiftly from failures.

RQ2: How can we efficiently handle concurrent container requests with multi-resource constraints on heterogeneous clusters? As more and more applications move to cloud computing, modern cloud platforms have

to handle a large number of concurrent container requests at the same time. However, existing queue-based container schedulers have crucial limitations to achieve high-quality schedules for concurrent requests, because they lack a global view on the waiting containers. For instance, a queue-based scheduler makes a decision early for a requested container (to be placed on a machine), which would restrict its choices for other waiting containers. The requested applications are often constrained by multiple resources, such as database applications that are compute-intensive and network-intensive. Moreover, the underlying cluster may consist of heterogeneous machines to support different applications. Therefore, it is essential to investigate scheduling approaches for efficiently handling concurrent container requests with multi-resource constraints on heterogeneous clusters.

RQ3: How can we optimize the placement of service-based applications in clouds? As microservice architecture is becoming more popular than ever, developers intend to transform traditional monolithic applications into service-based applications. To deploy a service-based application in clouds, besides the resource demands of each service, the traffic demands between collaborative services are crucial for the overall performance. Poor handling of the traffic demands can result in severe performance degradation, such as high response time and jitter. However, current cluster schedulers fail to place services at the best possible machine, since they only consider the resource constraints but ignore the traffic demands between services. In order to guarantee a desired performance, the cluster scheduler has to place each service with respect to service traffic demands carefully.

RQ4: How can we learn scheduling policies of DAG jobs with deep reinforcement learning on multi-resource clusters? DAG scheduling problems are pervasive in data-parallel clusters [68]. Efficiently scheduling DAG jobs on multi-resource clusters requires intricate algorithms, since the scheduler has to consider all the characteristics of the cluster and the DAG job, such as the cluster resource utilization, task resource demands (e.g., CPU, memory, network, etc.), task duration and inter-task dependencies, to make scheduling decisions. That is algorithmically hard. Due to recent advances in deep learning, applying deep neural networks in reinforcement learning can make it possible to deal with more complex problems which have high-dimensional states or actions [108, 126]. This breakthrough of deep reinforcement learning also provides a promising technique for dealing with DAG scheduling. Thus, we decided to investigate how to apply deep reinforcement learning to the scheduling problem of DAG jobs on multi-resource clusters.

1.6 Contributions and Thesis Outline

This thesis consists of 6 chapters. All the research questions are addressed in Chapter 2 through Chapter 5, respectively. The contributions of this thesis are listed below.

- **Deadline-aware Deployment for Time Critical Applications (Chapter 2)** To address research question RQ1, we propose a Deadline-aware Deployment System (DDS)[8] for time critical applications in clouds. DDS enables users to automatically deploy time critical applications and provide scheduling mechanisms to guarantee deployment deadlines. To be deadline-aware, DDS schedules deployment requests based on Earliest Deadline First (EDF) which is an effective real-time scheduling algorithm to minimize the number of missed deadlines. Furthermore, we design a bandwidth-aware approach to facilitate parallel transmission of multiple application packages and improve the utilization of network bandwidth.

- **Enhancing Scheduling for Concurrent Container Requests (Chapter 3)** To address research question RQ2, we propose an Enhanced Container Scheduler (ECSched)[9] for efficiently scheduling concurrent container requests with multi-resource constraints on heterogeneous clusters. To handle concurrent requests, we formulate the container scheduling problem as a minimum-cost flow problem (MCFP) and represent the container requirements using a specific graph data structure (flow network). In the flow network, we propose a novel approach to encode the multi-resource demands and affinity requirements of requested containers. By analyzing the properties of different classical MCFP algorithms, we choose an appropriate variant of the successive shortest path algorithms implemented in ECSched. At each scheduling event, ECSched first constructs a flow network based on a batch of concurrent requests and then performs the MCFP algorithm to schedule the batch of concurrent requests at the same time. In the experiments, we evaluate the container performance, resource efficiency, and scheduling overhead of ECSched with workloads derived from production workload traces.

- **Optimizing Placement for Service-based Applications (Chapter 4)** To address research question RQ3, we propose a new approach to optimize the placement of service-based applications in clouds. Considering the traffic demands among services, we aim to find a placement solution to minimize the overall traffic between the services that are placed on different machines (i.e., minimize inter-machine traffic), while

[8] https://github.com/huyang1022/Deployment-Agent
[9] https://github.com/huyang1022/ECSched

satisfying multi-resource demands of services. To address this problem, first, we partition a requested application into several parts while keeping overall traffic between different parts to a minimum. Meanwhile, we introduce a resource demand threshold to determine how many parts the application is partitioned into. Second, we try to pack all the parts of the partition into machines with multi-resource constraints. Finally, by adjusting the resource demand threshold, we combine the partition and packing to find an appropriate placement solution for the service-based application. We conduct extensive experiments to demonstrate the quality of our placement solution and the run time of the proposed algorithms.

- **Learning Scheduling Policies for DAG jobs (Chapter 5)** To address research question RQ4, we present GoTask[10], an approach that can learn to well schedule DAG jobs on multi-resource clusters. GoTask directly learns scheduling policies from experience through deep reinforcement learning. In order to deal with the complexity and scale of the DAG scheduling problem, we propose a two-stage approach to learn scheduling policies in GoTask. In the first stage, we leverage a deep reinforcement learning agent to learn policies for selecting a pending task of DAG jobs. In the second stage, we leverage another agent to learn policies for selecting a machine to run the selected task. In order to encode inter-task dependencies, we adopt an approach based on tasks' longest path and critical path in the state representation for task selection. For machine selection, we represent fitness scores of several packing heuristics in the state to facilitate the learning of scheduling policies. We implement a prototype of GoTask and a simulator for simulation of task execution on multi-resource clusters. In the experiments, we evaluate the performance and the convergence of our prototype.

In **Chapter 6**, we summarize the conclusions of this thesis and discuss future research directions.

[10] https://github.com/huyang1022/RLSched

2

Deadline-aware Deployment for Time Critical Applications

In this chapter, we investigate how to effectively deploy distributed applications with critical time constraints in clouds. We first analyze the procedure of deploying applications in clouds. We find that the transmission time is widely varying while the installation time is roughly stable in different locations. Regarding the transmission process of deployment, we propose a Deadline-aware Deployment System (DDS) for deploying time critical applications in clouds.

This chapter is based on:

- **Hu, Y.**, Wang, J., Zhou, H., Martin, P., Taal, A., De Laat, C. and Zhao, Z. Deadline-aware Deployment for Time Critical Applications in Clouds. In 2017 European Conference on Parallel Processing (EuroPar) (Pages 345-357). Springer, Cham.

2.1 Introduction

Cloud computing is the platform of choice for deploying and running many of today's businesses. When executing applications in clouds, deployment is an important step to make the required software and data of an application available before execution. In cloud environments, Software as a Service (SaaS), e.g., Google Apps, or Platform as a Service (PaaS), e.g., Amazon EMR [2], aims at hiding the deployment complexity by automating deployment during resource provisioning [136]. However, these solutions are not sufficient for applications that require infrastructure-level optimization under the given platform services or application-level customized environments, which are not included in predefined virtual machines or container images.

Time critical applications, such as disaster early warning systems, of-

ten have very high-performance requirements for data communication and processing[158]. To support time critical applications using cloud environments, developers often use Infrastructure as a Service (IaaS) to optimize overall system-level performance by selecting the most suitable virtual machines, customizing their network topology and optimizing the scheduling of execution on the virtual infrastructure [75, 159, 141]. During the execution, the virtual infrastructure often has to adapt to, e.g., virtual machines scaling out/in or up/down to handle dynamically changing workloads [156]. A deployment service is thus needed not only before the application execution for making the environment available but also at runtime. In particular, it is necessary to ensure that components can be deployed immediately whenever the application needs to rescale to handle increased workloads or migrate components to new VMs. Moving the repository of components closer to the application is necessary to ensure that such deployments can be handled as rapidly as possible for time critical applications. Furthermore, the deployment service also has to be aware of time constraints, e.g., deadlines, required for acceptable system performance. Deployments that fail to finish within certain deadlines harm user experience, affect application performance, and even incur penalties for application failure. However current cloud providers lack explicit support for deploying time critical applications where users need to manually deploy their applications step by step and have no guarantee regarding deployment deadlines.

In this chapter, we propose a Deadline-aware Deployment System (DDS) for time critical applications in clouds. DDS enables users to automatically deploy time critical applications and provides scheduling mechanisms to guarantee deployment deadlines. First, DDS helps users to create a local repository for application components instead of using a remote repository. It provides a guarantee of bandwidth for transmitting application packages as the transmission rate directly from the remote repository is widely varying. To be deadline-aware, DDS schedules deployment requests based on Earliest Deadline First (EDF) [98] which is a classical scheduling technique to minimize the number of missed deadlines. Furthermore, we design bandwidth-aware EDF to facilitate DDS to satisfy a greater number of deadline requirements and achieve sufficient utilization of network bandwidth. In the evaluation, we demonstrate that DDS significantly reduces the number of missed deployment deadlines and leverages network bandwidth sufficiently. We summarize our contributions as follows:

- We design and implement DDS, a deadline-aware deployment system which can support automatic deployment of time critical applications in clouds.
- We build on DDS to implement deployment scheduling algorithms that minimize the number of missed deployment deadlines and maximize the utilization of network bandwidth.

- We experimentally evaluate the benefits of DDS on the ExoGENI [34] test-bed and large-scale simulations by comparing it with three different scheduling schemes.

2.2 Problem Statement

A typical scenario for deploying distributed applications in clouds involves two basic steps: transmitting necessary application packages or software components from remote repositories to virtual machines (VMs) in the provisioned infrastructure, and installing the software once runnable. In this work, we assume containers, e.g., Docker [105], are the default way to wrap application components.

For a distributed application, the deployment service has to know the location of application components and the location to deploy in for each component. Those container images are often stored in a public registry, e.g., Docker hub, which stores a collection of repositories and is not a part of the provisioned virtual infrastructure. The deployment service should schedule the sequence of each component based on the application description for transmitting and installing each individual component. The time for deploying a single container (T_d) typically contains time cost for transmitting the component from its repository (T_f) and installing (extracting files from the Docker image) the component (T_i). The total deployment time of the whole application starts from the first component transmission until the last component finishes its installation. When an application contains more components, careless scheduling of the deployment sequence might lead to a high time cost, which can eventually influence the execution of the application if key application components are delayed during deployment.

T_f depends on the size of the container and the network bandwidth between a repository and a target. T_i mainly depends on the performance of the VM and the complexity of the container itself. In many cases, T_f is much bigger than T_i. Table 2.1 shows some observations in a private cloud environment (ExoGENI [34]). We created VMs which are "XOMedium" configuration in three different locations: Boston, Washington, and Houston. We found that T_f is widely varying because the internet connection between VMs and Docker hub is different in different locations, and T_i is stable for the same VM configurations. For meeting the deployment time constraints of time critical applications in a provisioned virtual infrastructure, the key challenge is how to minimize the transmission time T_f and predict the installation time T_i. Note that installation time prediction is not the focus in this work, as we assume that existing predictors [127] can achieve good estimations of installation time. In this work, we focus on the transmission process (i.e., T_f) of deployment.

Table 2.1: Comparison of transmission time and installation time in different locations

Docker Image	Image Size	Boston Rack	Washington Rack	Houston Rack
ubuntu	400Mb	$T_f : 40.8s(\pm 2.2s)$ $T_i : 6.3s(\pm 0.5s)$	$T_f : 27.0s(\pm 1.5s)$ $T_i : 6.4s(\pm 0.4s)$	$T_f : 20.3s(\pm 1.5s)$ $T_i : 6.3s(\pm 0.6s)$
nginx	576Mb	$T_f : 58.7s(\pm 2.5s)$ $T_i : 9.3s(\pm 0.7s)$	$T_f : 38.9s(\pm 2.6s)$ $T_i : 9.1s(\pm 0.5s)$	$T_f : 29.2s(\pm 1.8s)$ $T_i : 9.3s(\pm 0.6s)$
mongodb	1200Mb	$T_f : 122.4s(\pm 3.0s)$ $T_i : 15.4s(\pm 0.5s)$	$T_f : 81.0s(\pm 3.4s)$ $T_i : 15.7s(\pm 0.8s)$	$T_f : 60.9s(\pm 1.9s)$ $T_i : 15.5s(\pm 0.8s)$
cassandra	1296Mb	$T_f : 132.2s(\pm 3.1s)$ $T_i : 17.1s(\pm 0.9s)$	$T_f : 87.5s(\pm 3.4s)$ $T_i : 17.3s(\pm 0.7s)$	$T_f : 65.7s(\pm 2.3s)$ $T_i : 17.4s(\pm 0.6s)$

The deployment model in this work is a set of deployment requests. The deployment service has to optimize the time cost by scheduling application transmissions carefully and parallelize the data transfer based on the time constraints of the requested applications. We model the deployment request as a tuple $R_i = (v_i, s_i, d_i)$, where v_i is the target virtual machine to deploy request R_i, s_i is the application size (e.g., Mb), and d_i is its deadline. As we concentrate on transmission, we model bandwidth information for provisioned VMs as a set $B = \{b_1, b_2, b_3, \ldots, b_n\}$, where b_i denotes the bandwidth of virtual machine i. This means that the *throughput* of virtual machine i cannot exceed b_i during the transmission process, and the bandwidth is stable because of the service-level agreement (SLA) assurance [43] in the cloud. We denote the bandwidth of the target machine v_i as b_j, so that the transmission time of request R_i can be represented as $T_f = \frac{s_i}{b_j}$. Similarly, the deployment time can be represented as $T_d = \frac{s_i}{b_j} + T_i$. The problem of this chapter is thus to investigate the scheduling mechanisms for meeting the deployment deadlines (i.e., ensure that $T_d \leq d_i$) of time critical applications in clouds.

2.3 Deadline-aware Deployment System

This section highlights our approach in DDS. DDS aims to provide a deadline-aware, efficient and automatic deployment system that supports time critical applications on IaaS cloud systems. As we mainly consider the transmission part of the deployment procedure in this work, DDS focuses on the network of the underlying distributed system to provide the best guarantee for deployment within deadlines.

2.3.1 Design Principles

Repository location

The repository for applications is a shared storage from which application packages can be fetched to be installed on another machine. The repository can be located in a remote server or in the cloud already. The location of the repository can directly impact the deployment time because the network bandwidth between cloud VMs and the network bandwidth between a VM and a remote repository in a different location can be very different. Compared to a remote repository, a local repository within a cloud has some obvious advantages. First, the local repository has greater transmission capacity than the remote repository. Second, the bandwidth of the local repository inside a cloud is more stable, which provides a guarantee regarding the transmission time. Third, the local repository is more flexible due to the possibility of

personalized configuration. Thus, DDS would help users to create a local repository first for fast and stable network transmission.

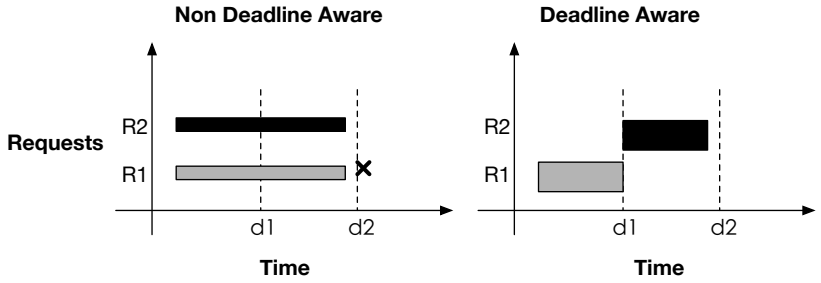


Figure 2.1: Awareness of deadlines can be used to meet two deadlines

Deadline-aware mechanism

As the goal of DDS to meet the deadline of requests, whether the system is aware of the deadline is important for deployment. Consider a common time critical application scenario involving two deployment requests sent to the same application repository simultaneously, where one request has a tighter deadline than the other. The resulting requests share a bottleneck via which to transmit application packages. As shown in Figure 2.1, with today's setup, the transport protocol (e.g., TCP) strives for fairness and the transmission finishes for both requests almost simultaneously. However, only one of the requests meets its deadline which makes another request invalidated or degrades its value. Alternatively, given explicit information about deployment deadlines, the system can arrange the transmission order to better meet the deployment deadline.

Bandwidth-aware mechanism

In addition to deadline-aware scheduling, to be aware of bandwidth is another significant attribute for deployment. Consider another scenario with two deployment requests, where the second request pulls a larger application package. The resulting requests also share a link to transmit their respective packages. As shown in Figure 2.2, the deployment system has information about the deadlines and schedules the transmission based on those deadlines. However, only one request meets its deadline. Because the transmission bottleneck is the bandwidth of the target machine, there is some spare bandwidth on the server

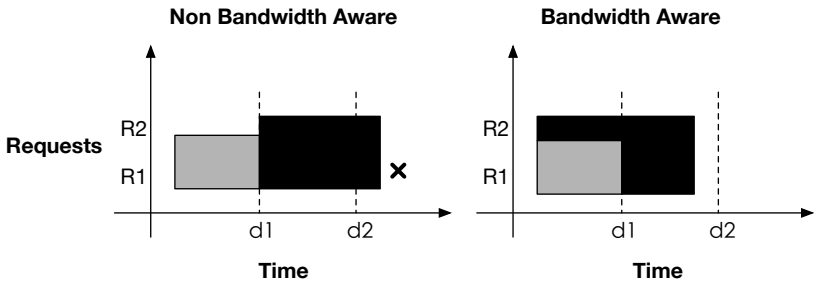


Figure 2.2: Awareness of bandwidth can be used to meet two deadlines

of the repository which is not used. Thus, given explicit information about the bandwidth capacity of each machine in the cloud, the system could schedule more deployment requests and leverage the bandwidth more efficiently.

2.3.2 Scheduling Algorithm

In this section, we zoom in on the design principles presented in Section 2.3.1 by providing an algorithmic description. The main goal of our algorithms is to minimize the deadline miss rate: the application packages should be transmitted to the target machine within the deadline wherever possible. In addition to minimizing the deadline miss rate, we should maximize the bandwidth utilization to reduce the total transmission time. To achieve both these goals, we employ EDF to prioritize requests and design bandwidth-aware EDF to support parallel transmission and realize dynamic rate control.

EDF scheduling

The key insight guiding the design of deadline-aware scheduling is derived from the classic real-time scheduling algorithm Earliest Deadline First (EDF) [98], which prioritizes tasks based on their deadline. EDF is an optimal scheduling algorithm in that if a set of deadlines can be satisfied under some schedule, then EDF can satisfy them too.

We adopt EDF to schedule deployment requests. When a deployment request comes, DDS compares the deadline of new request with previous requests and then sets the corresponding priority relative to the other deadlines. DDS then puts the new request into the request queue where the requests are sorted by priority. The algorithm is described in Algorithm 1. Consequently, DDS obtains the request from the queue and starts to transmit application

packages to the target machine.

Algorithm 1: EDF scheduling

Input: The new deployment request Ri
Output: The request queue RQ where requests sorted by the deadline

```
1: for each Rj ∈ RQ do
2:   if Ri.deadline < Rj.deadline then
3:     RQ.insert(Ri)
4:     return RQ
5:   end if
6: end for
7: RQ.append(Ri)
8: return RQ
```

Bandwidth-aware EDF scheduling

In addition to EDF scheduling, we design bandwidth-aware scheduling in cooperation with EDF scheduling. The key idea of bandwidth-aware scheduling is to make use of the spare bandwidth available between the local repository and the target as much as possible for parallelizing multiple requests. Thus, DDS needs the bandwidth information for each machine in the cloud. DDS would collect the bandwidth information before the whole deployment procedure begins.

EDF is optimal when the deadlines can be satisfied. However, without bandwidth information, EDF would schedule requests in a sequential way which leads to insufficient utilization of bandwidth or even missing deployment deadlines. However, if we directly schedule requests in a parallel way, the bandwidth contention among different requests can also cause deployment deadlines to be missed. Therefore, the challenge of bandwidth-aware scheduling is how to dynamically allocate transmission rates for deployment requests in order to avoid unnecessary contentions. For this purpose, we design bandwidth-aware EDF algorithm as described in Algorithm 2.

As per the description of bandwidth-aware EDF, if there is spare bandwidth in the local repository, DDS will continue to obtain requests from the request queue until the required bandwidth is equal or greater than the local repository bandwidth. DDS then sets the specific rate for the last deployment request to make sure the total required bandwidth is equal to the bandwidth of local repository. Consequently, it avoids bandwidth contention with previous deployment requests and makes full use of spare bandwidth to transmit. Once a new deployment request arrives, DDS performs bandwidth-aware EDF scheduling after the request is put into the request queue. When one deployment

Algorithm 2: Bandwidth-aware EDF scheduling

Input: *throughput* and *bandwidth* of the local repository

```
1:  while throughput < bandwidth do
2:    if RQ ∉ ∅ then
3:      R_i = RQ.pop()
4:      b_j = GetBandwidth(v_i)
5:      if throughput + b_j < bandwidth then
6:        throughput = throughput + b_j
7:      else
8:        SetTransmissionRate(R_i, bandwidth − throughput)
9:        throughput = bandwidth
10:     end if
11:     StartTransmission(R_i)
12:   end if
13: end while
14: return
```

request finishes, DDS will allocate the released bandwidth for the running requests first, and then perform bandwidth-aware EDF scheduling again.

2.4 Evaluation

In this section, we describe experiments for quantitative evaluation of the deadline-aware deployment system. We perform three kinds of experiments. First, we evaluate the transmission time using a DDS local repository versus a remote repository. Second, we evaluate DDS in comparison with three typical scheduling algorithms by running experiments on our cloud test-bed. Third, we evaluate DDS with large-scale simulations.

2.4.1 Repository Evaluation

In this section, we compare the transmission time to a target machine from a DDS local repository and a remote repository based on Docker. In most common cases, the application provider only has the repository outside clouds. Thus, DDS would help users to create local repository within their cloud first. We provision two virtual machines with 50Mbps bandwidth in the ExoGENI Boston rack and create a local repository in one of them. Then, we use the other machine to fetch the image from the local repository and also the original remote repository (Docker Hub). The comparative results are shown in the Table 2.2. We observe that the transmission time (T_f) from the local repository is much

less than from the remote repository. The reason is the network bandwidth between cloud VMs is much better than the network bandwidth between VMs and remote repositories.

Table 2.2: Comparison of transmission time from different repository

Docker Image	Image Size	Local Repository	Remote Repository
ubuntu	400Mb	$T_f : 8.1s(\pm 1.1s)$	$T_f : 40.8s(\pm 2.2s)$
nginx	576Mb	$T_f : 11.7s(\pm 1.3s)$	$T_f : 58.7s(\pm 2.5s)$
mongodb	1200Mb	$T_f : 24.4s(\pm 1.2s)$	$T_f : 122.4s(\pm 3.0s)$
cassandra	1296Mb	$T_f : 26.4s(\pm 1.5s)$	$T_f : 132.2s(\pm 3.1s)$

2.4.2 Testbed Experiments

In this section, we evaluate DDS alongside three typical scheduling algorithms in ExoGENI [34] test-bed. ExoGENI is a networked infrastructure-as-a-service (NIaaS) platform where researchers can define the network topology and bandwidth of virtual infrastructures. In our experimental setup, we chose the "XOXLarge" type of machine as our local repository, and all other application nodes we chose "XOMedium" type machines. The guest OS in VMs which are provisioned for evaluation is Ubuntu 14.04. In the experiment, we use *iPerf* [133] to simulate the application package transmission, therefore the size of the application package can be customized via *iPerf* in the evaluation. For transmission rate control, we leverage Linux Traffic Control (TC) to perform the rate limiting. We use two-level Hierarchical Token Bucket (HTB) in TC: the root node classifies requests to their corresponding leaf nodes based on IP address and the leaf nodes enforce the rate limiting.

Schemes to compare: we compare the following schemes with DDS.

- **FIFO:** all the deployment requests are scheduled by the arrival time of the request in a sequential way.
- **EDF:** all the deployment requests are scheduled by the EDF algorithm in a sequential way.
- **PARALLEL:** all the deployment requests are scheduled immediately after arrival in a parallel way.

Through comparison with these three schemes, we can inspect the benefits of DDS from different aspects. FIFO is the most common scheduling algorithm in application distribution. EDF is optimal in sequential scheduling when the deadline can be satisfied, but it is not bandwidth-aware. PARALLEL can achieve high utilization of the bandwidth, but it is not deadline-aware.

Metrics. In this section, we compare the number of schedulable requests (requests that meet the deadline) and the total deployment time among different schemes. The number of schedulable requests can indicate the satisfaction of deadline requirements. The total deployment time can indicate the utilization of the network bandwidth.

In this experiment, we provide two kinds of bandwidth configuration to evaluate DDS as described in Table 2.3. We instantiate four nodes to deploy time critical applications in ExoGENI. For these four nodes, we generate six deployment requests which include the target machine, application size, arrival time, and the deadline as described in Table 2.4. To understand the scheduling mechanisms in DDS better, we assume that the installation time T_i of each application is $1s$ in this experiment.

Table 2.3: Bandwidth Configuration

(a) Configuration A (Mbps)

Repository	Node1	Node2	Node3	Node4
100	20	50	70	100

(b) Configuration B (Mbps)

Repository	Node1	Node2	Node3	Node4
100	70	70	70	70

Table 2.4: Deployment Request

Machine	Size	Deadline	Arrival Time
Node1	200Mb	14s	0s
Node1	160Mb	20s	10s
Node2	320Mb	9s	11s
Node2	560Mb	15s	30s
Node3	960Mb	20s	30s
Node4	640Mb	25s	30s

In Figure 2.3, we inspect the number of schedulable requests on different schemes. We observe that DDS can schedule more requests in two different bandwidth configurations, because sequential scheduling (EDF, FIFO) cannot meet all the deadlines when multiple requests emerge simultaneously, and direct parallel scheduling suffers from the bandwidth contention. Figure 2.4 shows the total deployment time of various schemes. We note that the total deployment time of DDS is less than EDF and FIFO, and comparable to PARALLEL.

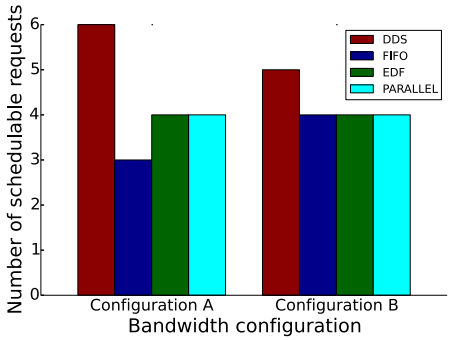


Figure 2.3: Comparison of the number of schedulable requests in various schemes

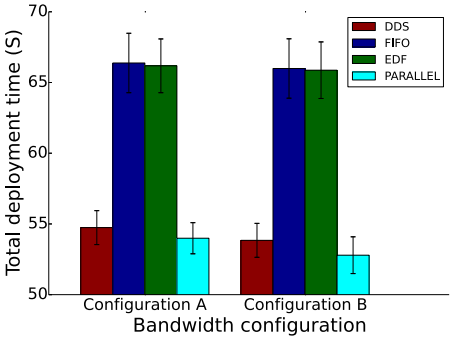


Figure 2.4: Comparison of the total deployment time in various schemes

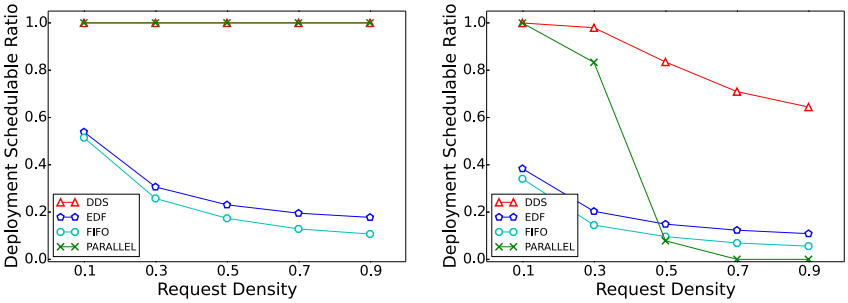
This indicates that DDS makes full use of network bandwidth to process the deployment requests.

2.4.3 Large-scale Simulations

We evaluate DDS with large-scale simulations in this section. We compare the deployment schedulable ratio which is the number of schedulable requests to the number of total requests in different schemes.

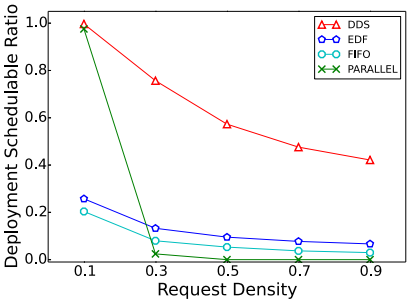
VMs configuration. We equip the repository server with 10Gbps bandwidth connection and the nodes for deploying applications with 1Gbps bandwidth connection which are the typical configurations in public clouds. In the simulation, the number of nodes ranges over 10, 20, 40 and 80 which are sufficient to account for most distributed cloud applications.

Deployment requests. We simulate the deployment service running 10 days $T_{running}$ in the experiment. During this period, we generate deployment requests in different densities to simulate deploying various applications on each node. We denote S^i_{total} as the total application size of all deployment requests on node i. The request density of node i is equal to $\frac{S^i_{total}}{T_{running} * 1Gbps}$, and the request density of whole system is the average for each node. The overall request density varies from 0.1 to 0.9. In the experiment, the deadline d_i of each request ranges from $10s$ to $100s$, and the application size is equal to $(d_i - 1) * 1Gbps$. We assume the installation time T_i is $1s$ in the simulation.

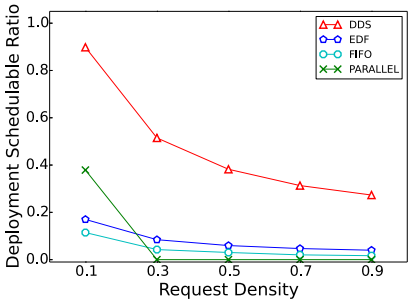


(a) 10 nodes

(b) 20 nodes



(c) 40 nodes



(d) 80 nodes

Figure 2.5: Comparison of the deployment schedulable ratio in different scenarios

Figure 2.5 shows the deployment schedulable ratio in different scenarios. We observe that: compared to EDF, DDS can reduce the deployment deadline miss ratio by 24% to 83%; compared to FIFO, DDS can reduce the ratio by 26% to 89%; compared to PARALLEL, DDS can reduce the ratio by up to 86%. Because EDF and FIFO schedule deployment requests in a sequential

way, DDS can benefit from parallelized deployments. The PARALLEL scheme parallelizes deployments but suffers severe bandwidth contention as request density increases. In contrast, DDS is bandwidth-aware and provides dynamic transmission rate control to avoid bandwidth contention for different deployment requests. In summary, due to the EDF and bandwidth-aware scheduling, DDS significantly reduces the number of deadline missing requests for deploying cloud applications.

2.5 Related Work

In recent years, deployment has been an important topic in a distributed environment, service-oriented systems, and cloud computing. The techniques in DDS are related to the following areas of research:

Automatic cloud application deployment. To enable automatic deployment has been the focus of several recent works [79, 81]. SO-MVDS [58] allows users to design and create virtual machines with specific services running in them and define a service deployment request to enhance the efficiency of service deployment. Li et al. [97] propose a general approach to application deployment. They adopt the contextualization process which is to embed various scripts in VM images to initiate applications. These approaches mainly focus on automating service deployment on virtual machines to improve deployment performance. DDS, on the other hand, leverages emerging container technologies, such as Docker, to achieve automatic application deployment with critical time constraints.

On-demand image distribution. The idea of efficiently distributing images in clouds has been explored in recent works [155, 118]. Vaquero et al. [138] proposes a solution based on combining hierarchical and Peer to Peer (P2P) data distribution techniques. VDN [115], a new VM image distribution network on the top of chunk-level, enables collaborative sharing in cloud data centers. These approaches focus on fast network transmission to reduce the VM instance provisioning time. In contrast, DDS is not only transmitting images efficiently but is also aware of deadlines via EDF scheduling mechanisms.

Deadline-aware scheduling techniques. D^3[145] and D^2TCP [137] are transport protocols designed for deadline-aware transmission inside data centers. These protocols add the deadline information to TCP and provide control mechanisms based on the deadline information. Techniques like Karuna [45] and pFabric [26] prioritize network flows to transmit. All these approaches schedule transmission at the flow level. Different to these efforts, DDS exploits the information of bandwidth to schedule transmission at application level which is more relevant to users' requirements.

2.6 Conclusion

It is challenging to deploy time critical applications in clouds while meeting the critical time constraints of deployment. This is an important and practical problem, but has been neglected by prior work in this field. In this chapter, we propose a Deadline-aware Deployment System (DDS) which helps users to create a local repository and automatically deploy applications into clouds. We investigate the scheduling mechanisms in cloud deployment system and implement EDF and bandwidth-aware scheduling algorithms in DDS. DDS schedules deployment requests based on the deadline and provides dynamic transmission rate control to avoid bandwidth contention for different deployment requests. In the evaluation, we show that DDS leverages network resources sufficiently and significantly reduces the number of missed deployment deadlines.

3

Enhancing Scheduling for Concurrent Container Requests

In this chapter, we investigate how to efficiently handle concurrent container requests with multi-resource constraints on heterogeneous clusters. We first expose the new features of container-based infrastructure comparing with VM-based infrastructure. We then point out the limitations of using existing container schedulers to schedule concurrent requests. With respect to the limitations, we propose an Enhanced Container Scheduler (ECSched) for efficiently scheduling concurrent container requests with multi-resource constraints on heterogeneous clusters.

This chapter is based on:

- **Hu, Y.**, Zhou, H., de Laat, C. and Zhao, Z. Ecsched: Efficient Container Scheduling on Heterogeneous Clusters. In 2018 European Conference on Parallel Processing (EuroPar) (Pages 365-377). Springer, Cham.
- **Hu, Y.**, Zhou, H., de Laat, C. and Zhao, Z. Concurrent Container Scheduling on Heterogeneous Clusters with Multi-Resource Constraints. Future Generation Computer Systems, Volume 102, Pages 562-573. 2020. Elsevier.

3.1 Introduction

Container technologies effectively virtualize the operating system and are becoming increasingly popular in cloud computing. By encapsulating runtime contexts of software components and services, containers significantly improve portability and efficiency for cloud application deployment. Major cloud providers have recently announced container-based cloud services to cater for this popularity [3, 5]. Meanwhile, container orchestration platforms, such as Docker Swarm [17], Mesosphere Marathon [72], and Google Kubernetes [42], are

emerging to provide container-based infrastructure for automating deployment, scaling, and management of containers on underlying clusters.

Typically, Infrastructure as a Service (IaaS) offered by the cloud providers (e.g., Amazon EC2, Microsoft Azure [3, 5]) relies on the underlying Virtual Machines (VMs). Compared with VM-based infrastructure, container-based infrastructure has some new features.

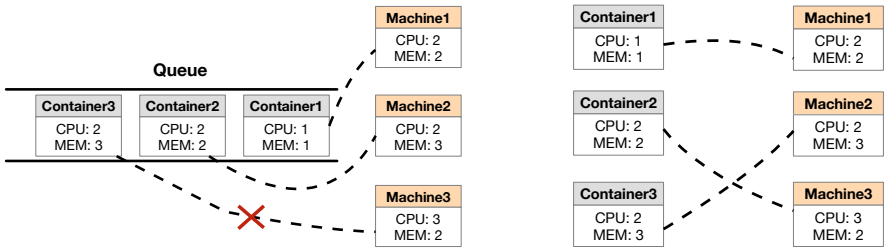
- It can be deployed on both physical and virtual machines, and the highly diverse configuration of VMs makes the clustered machines more heterogeneous.
- It can provide fine-grained resource allocation due to the operating-system-level virtualization techniques of containers, which is much more flexible than predefined VM types in VM-based infrastructure.
- It can support users specifying affinities among containers (e.g., Affinity in Kubernetes) for a distributed application, which facilitates container orchestration over the cluster.

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

1. The resources demanded by a container are often a combination of multiple resources (CPU, memory, network, etc.), which have to be satisfied by the underlying container cluster; it becomes extremely challenging when the nodes have diverse capacity and capability.
2. Containers of a distributed application often have strong affinity with other containers (due to frequent data communication) or specific machines (due to data locality). Placing containers on the appropriate node can significantly reduce the latency of container communication or decrease the volume of data transferred. Hence, the affinity has to be taken into account when scheduling containers.
3. The high scheduling overhead in large clusters may hurt the performance of applications with high-quality constraints [76, 141, 157, 159], e.g., real-time analytics [113, 130]. Moreover, the scheduling algorithms are frequently invoked during the application execution, in particular when scaling out or recovering from failure, which often have critical time constraints. Thus, the scheduling overhead should be small so that the scheduler is able to scale to large clusters.

Meanwhile, with the adoption of cloud services and the scale of applications increase, modern cloud platforms have to deal with a large number of concurrent

requests at the same time. By analyzing the Google cluster trace [119], the scheduler needs to make hundreds of task placement decisions per second in peak hours. When considering the above requirements, it inevitably introduces new challenges to the container schedulers. In recent years, container management and scheduling have attracted quite a lot of research attentions. A queue-based scheduler is widely used in the orchestration platforms, such as Marathon [11], Swarm [17] and Kubernetes [9]. All deployment requests first enter a queue; the scheduler fetches requests from the queue and processes one container (one pod in Kubernetes) at a time. Regarding the scheduling algorithms to the queue-based schedulers, variants of heuristic packing algorithms, such as Best-Fit Decreasing (BFD) and First-Fit Decreasing (FFD) [99, 24], are often adopted to achieve practical solutions.



(a) Container-by-container Scheduling (b) Concurrent Scheduling

Figure 3.1: An example of container-by-container scheduling and concurrent scheduling for three concurrent requests

Container-by-container scheduling has the advantage for making parallel decisions on distributed deployment [49, 113], but it also has crucial limitations for achieving high-quality placements of the entire workloads (concurrent tasks), due to its lack of global view on the waiting containers. For instance, the scheduler makes a decision early for a requested container, which would restricts its choices for the waiting containers. Figure 3.1 shows an example of container-by-container scheduling and concurrent scheduling for three concurrent requests, where the resource demands of containers and the available resources of machines are depicted. For the container-by-container scheduling, if we apply a simple scheduling algorithm (i.e., First come, first served), Container3 cannot be scheduled at this moment. It is because Machine3 does not have enough resources to run Container3. For the concurrent scheduling, as the scheduler has a global view of the entire workloads, it could schedule all the containers to the machines. The problem of scheduling a batch of concurrent requests can be usually formulated as an integer programming problem [41, 134] or a mixed integer programming problem [124]. However, those are NP-hard [122].

In this chapter, we propose an Enhanced Container Scheduler (ECSched) for efficiently scheduling concurrent container requests with multi-resource constraints on heterogeneous clusters. We formulate the container scheduling problem as a minimum-cost flow problem (MCFP) and represent the container requirements using a specific graph data structure (flow network). In the flow network, we propose a novel approach to encode the multi-resource demands and affinity requirements of requested containers. By analyzing the properties of different classical MCFP algorithms, we choose an appropriate variant of successive shortest path algorithm implemented in ECSched. In our implementation, ECSched first constructs the flow network based on a batch of concurrent requests, and then performs the MCFP algorithm to schedule the concurrent requests at the same time. To evaluate the scheduling quality, we compare the container performance and the resource efficiency of ECSched and state-of-the-art container schedulers in different testbed clusters. To understand the scheduling overhead, we measure the algorithm runtime of ECSched and state-of-the-art container schedulers by performing large-scale simulations.

we summarize our contributions as follows:

- We identify the specific challenges in effectively handling concurrent container requests on heterogeneous clusters with multi-resource constraints.
- We propose a novel and efficient approach to address concurrent multi-resource container scheduling problem using minimum-cost flow model.
- We show that ECSched outperforms state-of-the-art container schedulers in container performance and resource efficiency, and only introduces a small and acceptable scheduling overhead in large-scale clusters.

3.2 Problem Formulation

In this section, we first present the formulation of the containers scheduling problem. Then, we analyze different deployment requirements of container requests.

3.2.1 Model Description

In container-based infrastructure, the cluster is typically composed of a set of networked heterogeneous machines $\mathcal{M} = \{m_1, m_2, ..., m_M\}$, where $M = |\mathcal{M}|$ is the number of machines. We consider R types of resources $\mathcal{R} = \{r_1, r_2, ..., r_R\}$ (e.g., CPU, memory, or network bandwidth) in each machine. For machine m_i, let $V_i = (v_i^1, v_i^2, ..., v_i^R)$ be the vector of its resource capacities where the element v_i^j denotes the total amount of resource r_j available on machine m_i.

We consider the container requests continuously arrive over time. At one moment, we assume there is a set of concurrent requests $\mathcal{C} = \{c_1, c_2, ..., c_N\}$ that are to be deployed on the cluster, and $N = |\mathcal{C}|$ is the number of requests. For container c_i, let $D_i = (d_i^1, d_i^2, ..., d_i^R)$ be the vector of its resource demands, where the element d_i^j denotes the amount of resource r_j that the container c_i demands. To affinity specification, let 0-1 matrix $\mathbb{A} = [a_{ij}]_{N \times N}$ denote the container affinity. If $a_{ij} = 1$, it means that the container c_i has an affinity with container c_j. Let 0-1 matrix $\mathbb{B} = [b_{ij}]_{N \times M}$ denote the machine affinity. If $b_{ij} = 1$, it means that the container c_i has an affinity with machine m_j.

Next, we model a placement solution of the scheduler. Note that a placement solution means a mapping of containers to machines on the cluster in this work. Let 0-1 matrix $\mathbb{X} = [x_{ij}]_{N \times M}$ denote a solution, where x_{ij} is 1 if container c_i is to be deployed on machine m_j.

3.2.2 Deployment Requirements

By analyzing the features of container-based infrastructure, a placement solution is desired to satisfy the following requirements.

Multi-resource Guarantee

Providing multi-resource guarantee for each container on the heterogeneous cluster is the primary requirement to the scheduler. Container-based infrastructure, which has the advantages and benefits of container techniques inherently, can allocate resources in a more fine-grained way than VM-based infrastructure; it facilitates the flexibility of resource allocation for applications. Given the constraints of Service Level Agreements (SLAs) with users, different types of resource demands should be at least guaranteed with a placement solution so that SLAs are not violated. Thus, the resource demands of the containers in one machine should not exceed its capacity.

$$\begin{aligned} &\sum_{c_i \in \mathbb{C}} x_{ij} \cdot d_i^k \leq v_j^k \\ &\forall m_j \in \mathcal{M},\ \forall r_k \in \mathcal{R} \end{aligned} \tag{3.1}$$

Affinity Awareness

In container-based infrastructure, users can specify the affinity of containers in a deployment request, which represents the demands of data communication or the location of data input. As distributed applications transfer data frequently, especially data-intensive applications, the network condition would directly affect the overall performance. Considering the influence of the network, the

scheduler should be aware of the affinity requirements so that it can make effective use of this information to adjust container placements. The intuitive and effective solution is to co-locate the containers which have affinity to others on the same machine,

$$\sum_{m_k \in \mathcal{M}} x_{ik} \cdot x_{jk} \geq a_{ij} \quad \forall c_i, \forall c_j \in \mathcal{C} \tag{3.2}$$

and place the container on the affinity machine.

$$x_{ik} \geq b_{ik} \quad \forall c_i \in \mathcal{C}, \forall m_k \in \mathcal{M} \tag{3.3}$$

Accordingly, the challenge for a scheduler is how to efficiently schedule the concurrent requests while satisfying all the deployment requirements of requested containers.

3.3 Minimum Cost Flow Problem

As existing queue-based schedulers process one container at a time, the other waiting requests cannot be considered in the decision-making phase. Consequently, it is hard for a scheduler to achieve high-quality placements, since it makes a separate decision for each container. In this work, we choose a graph-based approach to model the container scheduling problem as minimum cost flow problem (MCFP) [23], which can perform the container scheduling of concurrent requests at the same time.

The minimum cost flow problem is an optimization and decision problem to find the minimum-cost way of sending a certain amount of flow through a flow network. A flow network is a directed graph $G = (V, E)$ with a source node $s \in V$ and a sink node $t \in V$, where each edge $e_{u,v} \in E$ has a capacity $c_{u,v} > 0$ and a unit transportation cost $w_{u,v}$. Figure 3.2 shows an example of a flow network.

In the flow network, the cost of sending a flow of $f_{u,v} \geq 0$ units along the edge $e_{u,v}$ is $f_{u,v} \cdot w_{u,v}$. The problem requires K units of flow (source node with a supply of K units) to be sent from source s to sink t, and the goal is to minimize the total cost of the flow over all edges:

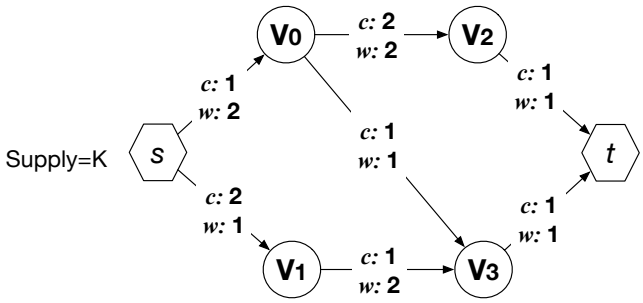


Figure 3.2: An example of a flow network

$$\text{Minimize} \quad \sum_{e_{u,v} \in E} f_{u,v} \cdot w_{u,v} \tag{3.4}$$

$$\text{subject to:} \quad f_{u,v} \leq c_{u,v} \tag{3.5}$$

$$\sum_{x \in V} f_{x,u} = \sum_{x \in V} f_{u,x} \ (u \neq s, t) \tag{3.6}$$

$$\sum_{x \in V} f_{s,x} = \sum_{x \in V} f_{x,t} = K \tag{3.7}$$

Equation 3.5 guarantees that the amount of the flow that goes through an edge cannot exceed its capacity. Equation 3.6 guarantees that the amount of the flow that goes into a node is equal to the amount of the flow that comes out of the node, except source node and sink node. Equation 3.7 guarantees that both the amount of the flow that comes out of source node and the amount of the flow that goes into sink node are equal to K. Equation 3.4 expresses the goal of the minimum cost flow problem.

MCFP is a well-studied problem in the past years [23]. A solution of MCFP can be extracted as a mapping between the nodes in the flow network. If we can convert the container scheduling problem to the MCFP by representing the status of requested containers and clustered machines with a flow network, we could apply effective MCFP algorithms to the concurrent scheduling problem and obtain a feasible placement solution (mapping) from the algorithms. Therefore, the question is how to convert the container scheduling problem to the MCFP, and what MCFP algorithm is used to solve the problem.

3.4 ECSched Approach

In this section, we describe how to construct the specific graph data structure (flow network) of MCFP to solve the container scheduling problem, what MCFP algorithms to use, and how to build ECSched scheduler.

3.4.1 Flow Network Structure

To map the container scheduling problem to the MCFP, we formulate the problem using a specific structure of flow networks. Figure 3.3 shows a case flow network of tackling the container scheduling problem, but we only annotate the capacity on the edges in the figure. The flow network corresponds to an instantaneous status of the container cluster, while encoding a set of requested containers and clustered machines. The overall structure of the flow network can be described as follows.

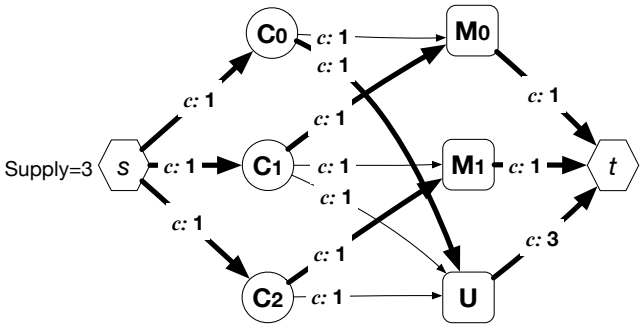


Figure 3.3: A case flow network of tackling the container scheduling problem

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

- **Unscheduled Node:** Inspired by previous works [61, 80], we add a new node in the flow network, which called unscheduled node U. All container nodes have an outgoing edge to the node U with a capacity of 1 unit.
- **Sink Node:** The sink node t on the right hand is the place to drain off the flow. All machine nodes have an edge to the sink node with a capacity of 1 unit, and the unscheduled node has an edge to the sink node with a capacity of N units.

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

For instance, all bold edges can be one possible solution returned by MCFP algorithms as shown in Figure 3.3. The solution corresponds to a placement solution: Container0 is not scheduled at this moment; Container1 is assigned to Machine0; Container2 is assigned to Machine1. Accordingly, the scheduler can successively perform MCFP algorithms to continuously schedule containers.

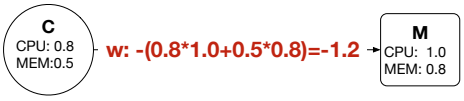
3.4.2 Encoding Deployment Requirements

As the goal of the MCFP problem is to minimize the total cost of the flow over all edges, flexible assignment of the costs on the edges can make the MCFP algorithms return a placement solution which we desire for container scheduling. Considering two deployment requirements as described in the previous section, we propose following methods to encode them in the flow network.

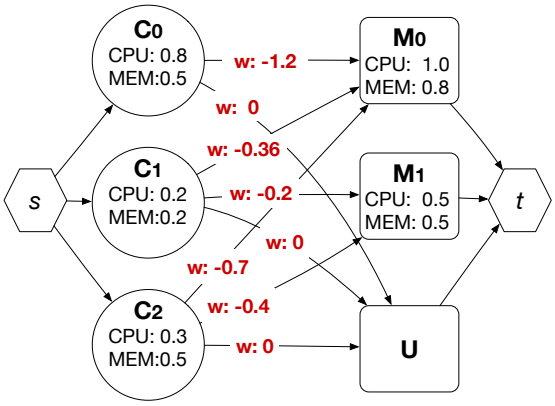
Multi-resource Guarantee

Providing multi-resource guarantee for each requested container is the primary objective of the container scheduling. In order to make the values of different resources comparable to each other and easy to handle, we first normalize the resource number to be the fraction of the maximum capacity in the cluster independently. For instance, there are two requested containers with resource demands: (CPU: 1 core, MEM: 2 GB) and (CPU: 2 cores, MEM: 1 GB), and there are two machines in the cluster with resource capacities: (CPU: 4 cores, MEM: 2 GB) and (CPU: 2 cores, MEM: 4 GB). After normalization, the vector of container resource demands becomes (CPU: 0.25, MEM: 0.5) and (CPU: 0.5, MEM: 0.25); the vector of machine resource capacities becomes (CPU: 1.0, MEM: 0.5) and (CPU: 0.5, MEM: 1.0), since the maximum number of CPU cores is 4 and the maximum capacity of memory is 4 GB in the cluster.

Next, the scheduler would construct the flow network as mentioned earlier. To construct the flow network, the scheduler checks whether the machines in the cluster have sufficient resources to place the requested containers. If a machine is eligible for a container, it adds an edge from the container node to the machine node with a capacity of 1 unit. The key challenge here is how to assign the costs on the edges to distinguish the quality of different mappings between containers and machines. In this work, we introduce two strategies which are inspired by vector bin packing algorithms: dot-product heuristic [114] and most-loaded heuristic.



(a) An example of dot-product heuristic

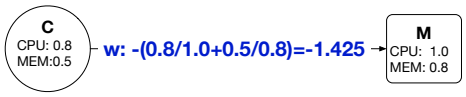


(b) A flow network example of dot-product heuristic

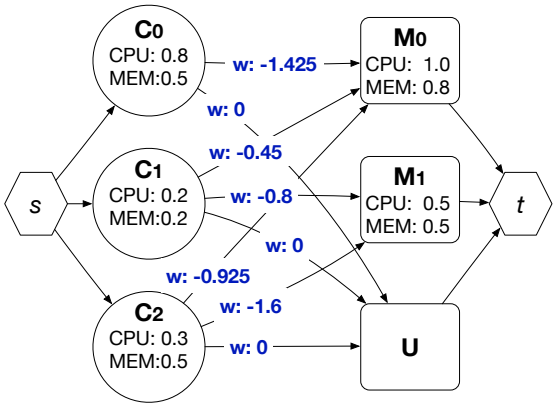
Figure 3.4: An example for encoding the multi-resource requirements based on dot-product heuristic

- **Dot-product:** In this heuristic, we prioritize different placements based on the dot product. Here, the dot product between the demand vector of container c_i and the capacity vector of machine m_j is defined as $dp_{ij} = \sum_{r_k \in \mathcal{R}} d_i^k v_j^k$. Figure 3.4(a) shows an example. The dot product between the container and the machine in the figure can be calculated as: $0.8 \times 1.0 + 0.5 \times 0.8 = 1.2$. The idea of this heuristic is that it takes into account not only the resource demands of containers but also how well its demands align with the resource capacities of machines; the dot product implies the degree of the alignment. Thus, for this heuristic, the higher dp_{ij} is, the better the placement is. In MCFP, the cost on the edge is inversely

related, which is a flow is better if the cost of the flow is lower. Therefore, the cost on the edge between the container node and the machine node is assigned to $-dp_{ij}$ in the flow network. For the edge from container node to unscheduled node, the cost is assigned to 0 which is the highest. A flow network example is shown in Figure 3.4(b).



(a) An example of most-loaded heuristic



(b) A flow network example of most-loaded heuristic

Figure 3.5: An example for encoding the multi-resource requirements based on most-loaded heuristic

- **Most-loaded:** In this heuristic, we prioritize different placements based on the load situations of the machines. The container tends to be placed on the most loaded machine in the cluster. In this cost model, it is also based on a scalar value which is defined as $ml_{ij} = \sum_{r_k \in \mathcal{R}} \frac{d_i^k}{v_j^k}$, which implies the mapping quality between the container c_i and the machine m_j. Figure 3.5(a) shows an example. The value between the container and the machine in the figure can be calculated as: $\frac{0.8}{1.0} + \frac{0.5}{0.8} = 1.425$. Thus, the higher ml_{ij} is, the more loaded the machine is in this heuristic. Similar to dot-product heuristic, the cost on the edge is assigned to $-ml_{ij}$. For the edge from container node to unscheduled node, the cost is also assigned to 0. A flow network example is shown in Figure 3.5(b).

Affinity Awareness

When submitting a deployment request, users can specify the affinities among the containers. It represents the demands of data communication or the location of data input. An appropriate placement of containers can lead to lower network latency and better network utilization. Thus, the location of containers is crucial for the overall performance. In the flow network, it is flexible to handle container affinity (co-located on the same machine) and machine affinity (located on a specific machine) by dynamically adjusting the edges in the flow network.

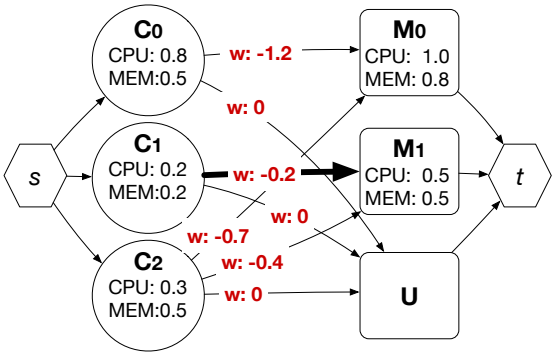


Figure 3.6: A flow network for encoding the machine affinity requirements based on dot-product heuristic

- **Machine affinity:** Considering the location of input data or container image, users would specify the machine affinity to indicate the preferred machine. Placing the container on the specified machine can reduce network transmission time significantly. Thus, we adjust the flow network to only connect the container with the preferred machine, which can limit placement options of the requested container. Figure 3.6 shows an example with machine affinity, where container c_1 has a machine affinity to machine m_1. In the example, container c_1 has only one edge to machine m_1 but no edge to machine m_0 that is also eligible for container c_1. Accordingly, container c_1 can only be scheduled to machine m_1.

- **Container affinity:** Considering the network latency within the containers, users would specify the container affinity among certain containers. Placing these containers on the same machine can reduce network latency significantly. In the flow network, we add a new node, called aggregator node A_i, to merge affinity containers. Figure 3.7 shows an example with container affinity, where container c_0 and container c_1 have an affinity. In the example, we add an aggregator node A_0 in the flow network. Both

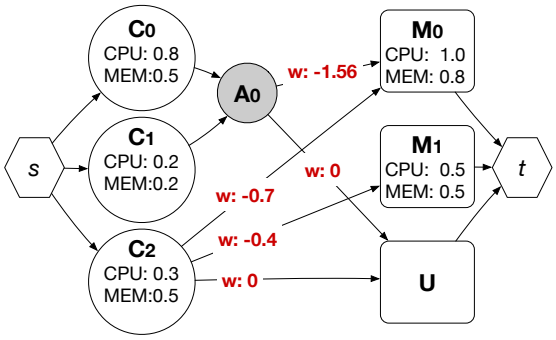


Figure 3.7: A flow network for encoding the container affinity requirements based on dot-product heuristic

container c_0 and container c_1 have an edge to aggregator node A_0. Hence, the scheduler would treat these two container nodes as one node to perform scheduling.

3.4.3 MCFP Algorithms

After constructing the flow network, the scheduler will perform MCFP algorithms to find the optimal routing solution with respect to the costs we have assigned. In this section, we discuss two kinds of classical MCFP algorithms: **cycle canceling algorithm** and **successive shortest path algorithm**. We then explain the MCFP algorithm that we implemented in ECSched.

The simplest MCFP algorithm is cycle canceling algorithm [87]. This algorithm maintains a feasible solution meeting Equation (3.5)-(3.7) and at every iteration attempts to improve its optimality. The algorithm first establishes a feasible flow in the flow network by solving a maximum flow problem [23]. Then it iteratively finds negative cost-directed cycles in the residual network and augments flows on these cycles. The residual network is defined as follows. In this flow network, each edge $e_{u,v} \in E$ with a capacity $c_{u,v} > 0$ and a unit transportation cost $w_{u,v}$ is replaced by two edges: $e_{u,v}$ and $e_{v,u}$. Then edge $e_{u,v}$ has a residual capacity $r_{u,v} = c_{u,v} - f_{u,v}$ and a unit transportation cost $w_{u,v}$, while edge $e_{v,u}$ has a residual capacity $r_{v,u} = f_{u,v}$ and a unit transportation cost $-w_{u,v}$. All constraints of MCFP can also be applied in the residual network. The cycle canceling algorithm terminates when the residual network contains no negative cost-directed cycle.

The cycle canceling algorithm maintains feasibility of the solution at every step and attempts to achieve optimality. In contrast, the successive shortest path algorithm [54] maintains optimality of the solution at every step and strives

to attain feasibility. At each step, the algorithm sends flow from the source node s to the sink node t along the shortest path in the residual network. The algorithm terminates when the current solution meets Equation (3.5)-(3.7) of MCFP.

Based on these two classical MCFP algorithms, many optimization methods and scaling algorithms have been proposed [64, 63, 62, 51, 112]. Known worst-case complexity bounds on the MCFP are $O(N^2C\log(N))$ [112] for the successive shortest path based algorithm, and $O(N^2M\log(NW))$ [64] for the cycle canceling based algorithm. N is the number of nodes; M is the number of edges; C is the number of the largest edge capacity; W is the number of the largest edge cost. In our container scheduling problem, it is the case as $M > N > C > W$. Thus, the successive shortest path based algorithm would perform better due to the complexities ($C\log(N) < M\log(NW)$). On the other hand, the cost on the edges is not integer in the flow network as we defined in above sections; it can be easier to solve through the successive shortest path based algorithm. Therefore, we choose to implement a variant of successive shortest path algorithm in our ECSched.

3.4.4 Implemention

The Implemention of ECSched is based on a heartbeat mechanism. On a heartbeat, ECSched first fetches a set of container requests to construct a flow network, and then performs the MCFP algorithm to place the requested containers. The details are explained as follows.

Constructing flow network

First, ECSched would fetch a certain number of concurrent container requests from the queue system. In our implementation, users can customize the maximum number of requests that ECSsched can fetch. Considering the tradeoff between scheduling quality and scheduling overhead, selecting a proper number is crucial for the overall performance. We discuss the tradeoff in Section 3.5.5. Then, ECSched constructs the flow network according to the strategies we described earlier.

Placing requested containers

Next, ECSched would perform the MCFP algorithm (a variant of successive shortest path algorithm) to find an optimal flow solution over the flow network. Then, ECSched extract the container placements out of the optimal flow solution. According to the placements, ECSched places the scheduled containers on the corresponding machines and puts the unscheduled containers back to the queue system for the next scheduling.

3.5 Evaluation

We implement ECSched with a container manager and a variant of MCFP algorithm in Python. In this section, we evaluate our ECSched in testbed clusters of ExoGENI [34] experimental environment to compare the scheduling quality with state-of-the-art container schedulers. In order to understand the scheduling overhead of ECSched, we measure the algorithm runtime by performing large-scale simulations.

3.5.1 Experimental Setup

Cluster. We create two different container clusters with 30 virtual machines (VM) in ExoGENI [34] which is a multi-domain Infrastructure-as-a-Service testbed. For the first cluster, we use 30 homogeneous VMs of "XOLarge" type (2-core CPU, 6 GB of memory) in the testbed. Considering the heterogeneity, we choose three types of VM configurations for the second cluster. The cluster is composed of 10 VMs of "XOMedium" type (1-core CPU, 3 GB of memory), 10 VMs of "XOLarge" type (2-core CPU, 6 GB of memory) and 10 VMs of "XOXLarge" type (4-core CPU, 12 GB of memory). After normalization, the capacity vectors of the machines in the homogeneous cluster are all: (CPU: 0.5, MEM: 0.5); the capacity vectors of the machines in the heterogeneous cluster are: (CPU: 0.25, MEM: 0.25), (CPU: 0.5, MEM: 0.5), and (CPU: 1, MEM: 1) respectively.

Workloads. To test our prototype, we yield container requests based on the Google cluster trace [119], which provides data from a 12,500-machine cluster over a month-long period. As we choose to spend 6 hours at each experiment, we analyzed the trace of the first 6 hours. There are around 100,000 tasks completed within the first 6 hours, and the average duration of the tasks is around 740 seconds. Considering the scale of our testbed cluster, we randomly sample 2,500 tasks (2.5%) from them at each experiment. The generator yields container requests according to following aspects from the trace: task submission times, task durations and task resource requirements. The resource requirements have been normalized in the trace. Additionally, we add the requirements of container affinity and machine affinity with a probability according to the task constraints in the trace [119].

ECSched. We implement two strategies in our scheduler. ECSched-dp is based on dot-product heuristic; ECSched-ml is based on most-loaded heuristic. For the heartbeat mechanism, the scheduling interval is set to be 100 ms in the scheduler. Unless otherwise specified, the maximum number of container requests that ECSched can fetch from the queue system on a heartbeat is 100.

Baselines. We compare ECSched to the state-of-the-art scheduling algorithms implemented in two container orchestration systems: Google Ku-

bernetes [9] and Docker Swarm [17]. Under multi-resource requirements, the default scheduler of Kubernetes tends to distribute pods (smallest deployable units in Kubernetes) evenly across the cluster to balance the overall resource usage, while the scheduler of Swarm tends to place containers on the most loaded machines to improve resource utilization over the cluster. Both are queue-based schedulers, which process one unit at a time.

Metrics. The primary metric to quantify container performance is the improvement in the average container completion time. We define the *Factor of Improvement* as follows:

$$\text{Factor of Improvement} = \frac{\text{Duration of a Baseline}}{\text{Duration of ECSched}} \tag{3.8}$$

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

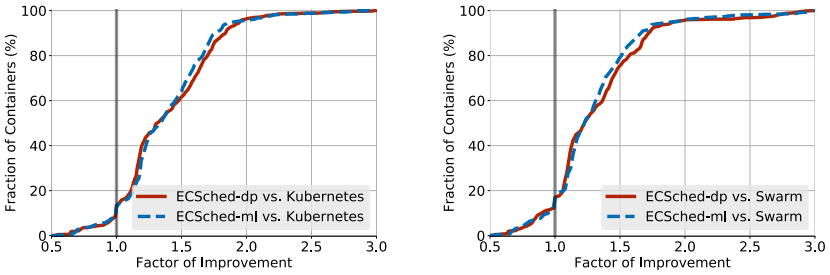
Additionally, we use *Average Resource Utilization* over the cluster to measure resource efficiency during experiments.

Finally, to evaluate scheduling overhead, we compute *Algorithm Runtime* of schedulers in different scenarios.

3.5.2 Comparison of Container Performance

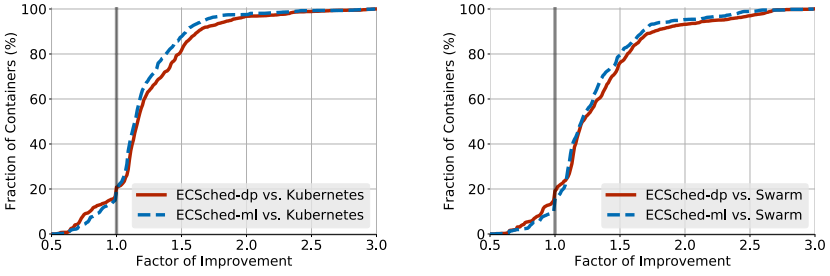
We first compare the container performance with baseline schemes to handle 2,500 container requests on both clusters. Figure 3.8 shows the results on the cluster with homogeneous machines. Overall, ECSched speeds up 83% to 86% of the containers and only slows down 8% to 12% of the containers. The reason is that ECSched schedules a batch of requests at the same time to find a more compact placement for the overall performance, which may hurt a small part of the requests due to the contention. We observe that ECSched-dp slightly outperforms ECSched-ml in the evaluation, as dot-product heuristic is that it takes into account not only the resource demands of containers but also how well its demands align with the resource capacities of machines. Compared to the scheduler of Kubernetes, ECSched-dp speeds up containers by 1.32× at the median and 1.68× at the 80th percentile. Compared to the scheduler of Swarm, ECSched-dp speeds up containers by 1.23× at the median and 1.57× at the 80th percentile.

Figure 3.9 shows the results on the cluster with heterogeneous machines. In this cluster, ECSched speeds up 79% to 81% of the containers and slows down 10% to 15% of the containers. Similar to the homogeneous cluster, ECSched-dp performs better than ECSched-ml. Compared to the scheduler of Kubernetes, ECSched-dp speeds up containers by 1.20× at the median and 1.50× at the 80th percentile. Compared to the scheduler of Swarm, ECSched-dp speeds up containers by 1.28× at the median and 1.63× at the 80th percentile.



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

Figure 3.8: CDFs of *Factor of Improvement* on the cluster with homogeneous machines



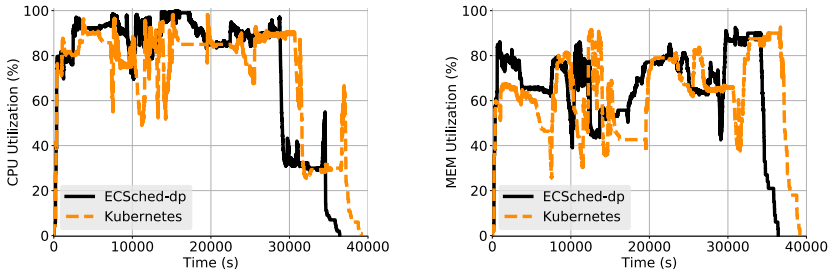
(a) Comparing with Kubernetes (b) Comparing with Swarm

Figure 3.9: CDFs of *Factor of Improvement* on the cluster with heterogeneous machines

Relative to the baselines, the performance of containers is improved in both clusters. ECSched lowers the average container completion time by up to 1.3× throughout the experiments. The improvements accrue from the increase in the number of simultaneously running containers on the cluster (less waiting time in the queue), as ECSched takes a batch of concurrent requests into consideration to make placement decisions.

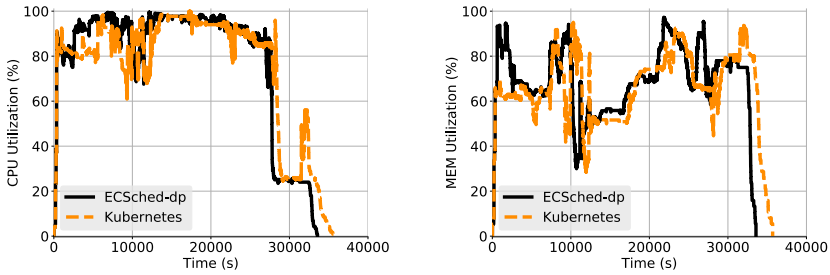
3.5.3 Comparison of Resource Efficiency

Next, we compare the average resource utilization over the cluster to evaluate the resource efficiency of different schedulers. During the experiments, we monitor the resource utilization across the cluster in every second. The



(a) CPU utilization (b) Memory utilization

Figure 3.10: Comparing the resource utilization on the cluster with homogeneous machines



(a) CPU utilization (b) Memory utilization

Figure 3.11: Comparing the resource utilization on the cluster with heterogeneous machines

resource utilization here is the ratio of the utilized resources to the total resources in the cluster. Table 3.1 and Table 3.2 show the average resource utilization throughout the entire experiment. We observe that ECSched sustains higher resource utilization than the baselines. Consistent with the container performance, ECSched-dp achieves the highest average resource utilization. For the cluster with homogeneous machines, ECSched-dp increases resource utilization by 4.1% to 5.3% compared to the two baselines. For the cluster with heterogeneous machines, ECSched-dp increases resource utilization by 3.7% to 4.6% compared to the two baselines. In order to better understand the resource efficiency, we choose to plot the exact resource utilization of ECSched-dp and Kubernetes during the experiment.

Figure 3.10 and Figure 3.11 show the details of ECSched-dp and Kubernetes

Table 3.1: Average resource utilization on the cluster with homogeneous machines

Scheduler	CPU utilization	Memory utilization
ECSched-dp	**76.57**%	**67.03**%
ECSched-ml	76.10%	66.61%
Kubernetes	71.21%	62.33%
Swarm	71.88%	62.92%

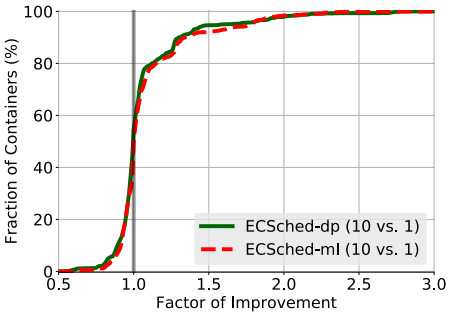
Table 3.2: Average resource utilization on the cluster with heterogeneous machines

Scheduler	CPU utilization	Memory utilization
ECSched-dp	**79.81**%	**69.86**%
ECSched-ml	79.22%	69.34%
Kubernetes	75.53%	66.11%
Swarm	75.18%	65.80%

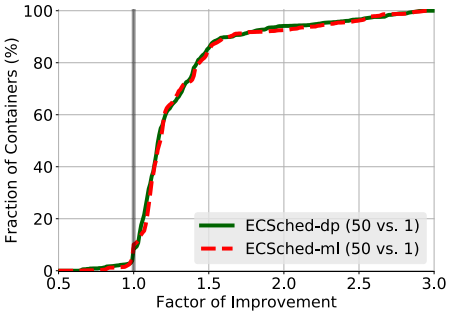
for both clusters. We observe that the requests we yielded are more CPU intensive. The CPU resources are highly competitive, and the utilization of CPU remains high during the experiments. Nevertheless, ECSched-dp still achieves higher resource utilization than Kubernetes in the peak hours. These improvements are because ECSched leverages the MCFP algorithm to find a better placement solution for the concurrent container requests, which can lead to less resource fragmentation of machines. Overall, it demonstrates that the ECSched outperforms existing container schedulers in terms of resource efficiency on different cluster.

3.5.4 Impact of Concurrent Scheduling

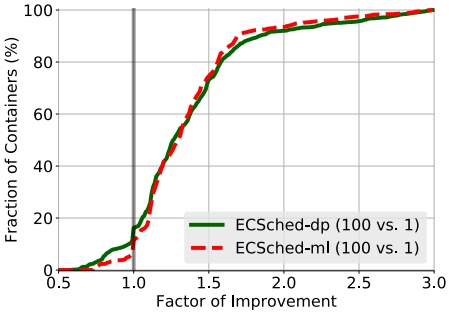
Compared to state-of-the-art schedulers, scheduling a set of concurrent requests at the same time is an innovative advantage of ECSched. As described earlier, ECSched would fetch a certain number of container requests from the queue system to construct a flow network for scheduling. For the above experiments, we set the maximum number that ECSched can fetch to 100. Thus, ECSched can schedule up to 100 container requests at a time. In this section, we configure the ECSched with the maximum fetch number of 1, 10, 50 and 100 to evaluate the container performance in different configurations. In order to understand how much influence does it have on the container completion time, we compare the configuration with the number of 10, 50, 100 to the configuration with the number of 1 in this experiment.



(a) 10 vs. 1



(b) 50 vs. 1



(c) 100 vs. 1

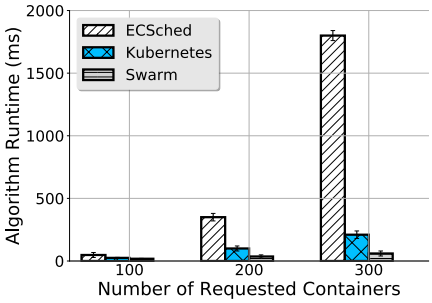
Figure 3.12: Comparing the container performance in the configurations with maximum fetch number of 1, 10, 50 and 100

Figure 3.12 shows the results of *Factor of Improvement* on the cluster with heterogeneous machines. We observe that along with the increase of maximum fetch number, the improvement of the container performance also increases. The impact to these two heuristics is quite similar in the experiment. Compared to the configuration with maximum fetch number of 1, ECSched lowers the average container completion time by 1.08× with the number of 10, 1.21× with the number of 50, and 1.30× with the number of 100. Consequently, it demonstrates that ECSched can efficiently handle concurrent container requests, and significantly benefits from concurrent scheduling.

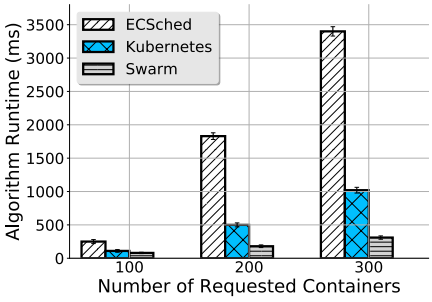
3.5.5 Overhead Evaluation

As we model the scheduling problem as a MCFP, the scheduling algorithm in our scheduler is more complex than existing schedulers. It would cause the overhead of ECSched to be higher than the others. To estimate the scheduling overhead, we perform large-scale simulations to measure the algorithm runtime of different schedulers. We consider two cluster sizes in the simulation: 1000-machine cluster and 5000-machine cluster (largest cluster which Kubernetes can support currently). In order to make the simulated cluster more heterogeneous, the configuration of each machine is chosen uniformly at random from 4 types of VM instances of ExoGENI experimental environment. As hundreds of requests need to be processed per second in peak hours according to the Google cluster trace [119], we choose to submit 100, 200 and 300 concurrent container requests to the scheduler for testing at the same time. Accordingly, we configure the ECSched with maximum fetch number of 100, 200 and 300. In order to fairly compare the algorithm runtime, we also implement the scheduling algorithm of Kubernetes and Swarm in Python, which is the same with ECSched. We conduct this experiment on a dedicated server with Intel Xeon E5-2630 2.4GHz CPU and 64GB memory.

Figure 3.13 shows the results of the average algorithm runtime which we repeated one hundred times. We observe that the algorithm runtime of ECSched is highest while Swarm is lowest. The algorithm of Swarm is a simple greedy search to place requested containers on the most loaded machines. Compared to Swarm, the algorithm of Kubernetes is a bit complex, which has multiple predicated policies and priorities policies to filter and score machines. Obviously, our algorithm is the most complicated one, and has higher overhead. Nevertheless, ECSched can respond in sub-second time when the number of concurrent requests is less than 100. When processing 300 containers concurrently, the ECSched responds in about 1.8 seconds for 1000-machine cluster and about 3.4 seconds for 5000-machine cluster. Actually, compared to the average duration (740 seconds in our experiments) of the containers in the cluster [119], this overhead is relatively small and acceptable. Considering the



(a) 1000-machine cluster



(b) 5000-machine cluster

Figure 3.13: Comparing algorithm runtime with large-scale simulations

container performance we discussed in previous section, there thus is a tradeoff between the quality and the overhead when scheduling containers. Users can dynamically adjust the maximum fetch number of ECSched to seek a best tradeoff for their workloads. Overall, we believe that ECSched is effective and usable in practice.

3.6 Related Work

The problem investigated in this chapter - container scheduling on heterogeneous clusters with multi-resource constraints - is related to a variety of research topics as follows.

Bin packing The problem of VM placement or consolidation which is similar to our problem is often formulated as vector bin packing problem, and various heuristics have been proposed for this problem [99, 93, 116, 57, 94].

Stillwell et al. [128] studied the resource allocation problem in shared hosting platforms for static workloads with machines which provide multiple types of resources. They proposed several kinds of vector bin packing algorithms and evaluated them over a wide range of simulations. They concluded that the first fit decreasing (FFD) heuristic that reasons on the sum of the resource demands of the tasks are the most effective. Furthermore, Panigrahy et al. [114] systematically studied variants of the FFD algorithm that have been proposed for VM placement problems, and presented a different generalization of the classical FFD heuristic. In their empirical evaluations, it showed that the Dot-Product heuristic often outperforms FFD-based heuristics. These studies focus on the packing problem with identical bins (i.e., machines), and consider each request independently. Different from them, we tackle the problem of scheduling concurrent requests on heterogeneous cluster and consider the requirements of container affinity and machine affinity at the same time.

Metaheuristics In recent years, many metaheuristic techniques have become prevalent for the approximate solution of multi-objective optimization problems [149, 59, 55, 101]. Mi et al. [106] proposed a genetic algorithm based approach, namely GABA, to adaptively self-reconfigure the virtual machines on large-scale clusters which is composed of heterogeneous machines. Xu et al. [149] presented a modified genetic algorithm to find global optimal solutions of virtual machine placement problem. Their approach leverages a fuzzy-logic based evaluation for incorporating different objectives. Gao et al. [59] proposed a multi-objective ant colony system algorithm to find a set of Pareto solutions for the virtual machine placement problem. However, these approaches often take minutes or even hours, particularly for large-scale clusters, to generate a placement solution, which would face difficulties for a online response. In contrast, we formulate the scheduling problem as a minimum cost flow problem, which can be solved in a polynomial time.

Cluster schedulers Many cluster schedulers have been proposed for different purposes [77, 68, 67, 82]. Sparrow [113] and Tarcil [49] are distributed schedulers developed for clusters that achieve a high throughput for short tasks. Quincy [80], a fair cluster scheduler, models the fair scheduling problem as a minimum cost flow problem to schedule jobs into slots. In their flow network, the edge capacities and weights encode the demands of starvation-freedom, data locality, and fairness. And then, they used a standard solver to compute the optimal solution based on a cost model. In contrast, we focus on the concurrent container requests with multi-resource demands and implement an appropriate MCFP algorithm for our problem. Firmament [61], a centralized scheduler that can scale to over ten thousand machines at sub-second placement latency via a min-cost max-flow (MCMF) optimization. They proposed some problem-specific optimizations for MCMF algorithms and can achieve low latency by solving the problem incrementally. However, they cannot handle the requests

with multi-resource demands. ECSched shows that how to encode multi-resource constraints and affinity requirements in minimum cost flow problem.

3.7 Conclusion

In this chapter, we have presented ECSched, an efficient solution for handling concurrent container requests on heterogeneous clusters with multi-resource constraints. ECSched is a graph-based scheduler, which can leverage the minimum-cost flow model to effectively process concurrent container requests. In the testbed experiments, we demonstrate that ECSched can achieve better scheduling quality than state-of-the-art container schedulers, which can lower the average container completion time by up to 1.3× and noticeably improve resource utilization. The large-scale simulations show that there is relatively small overhead of ECSched, but it is acceptable in practice.

4

Optimizing Placement for Service-based Applications

In this chapter, we investigate how to optimize the placement of service-based applications in clouds. For deploying a service-based application in clouds, besides the resource demands of each service, the traffic demands between collaborative services are crucial for the overall performance. To tackle this problem, we propose a new approach to optimize the placement of service-based applications in clouds.

This chapter is based on:

- **Hu, Y.**, de Laat, C. and Zhao, Z. Multi-objective Container Deployment on Heterogeneous Clusters. International Workshop on Network-Aware Big Data Computing, In Proceedings of 2019 IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID) (Pages 592-599). IEEE. (**Best paper award**)
- **Hu, Y.**, de Laat, C. and Zhao, Z. Optimizing Service Placement for Microservice Architecture in Clouds. Applied Sciences. (Under review)

4.1 Introduction

Microservices architecture is a new trend rising fast for application development, as it enhances flexibility to incorporate different technologies, reduces complexity by using lightweight and modular services, and improves overall scalability and resilience of the system. In the definition [13], the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. The application then is composed by a number of services (service-based application) that work cohesively to provide

complex functionalities. Due to the advantages of microservices architecture, many developers intend to transform traditional monolithic applications into service-based applications. Ensuring the desired performance of service-based applications, especially the network performance between the involved services, becomes increasingly important and also brings new challenges.

In general, service-based applications involve numerous distributed and complex services which usually require more computing resources beyond single machine capability. Therefore, a cluster of networked machines or cloud computing platforms (e.g., Amazon EC2 [5], Microsoft Azure [3], or Google Cloud Platform [6]) are generally leveraged to run service-based applications. More importantly, containers are emerging as the disruptive technology for effectively encapsulating runtime contexts of software components and services, which significantly improves portability and efficiency of deploying applications in clouds. When deploying a service-based application in clouds, several essential aspects have to be taken into account. First, services involved in the application often have diverse resource demands, such as CPU, memory and disk. The underlying machines has to ensure sufficient resources to run each service at the same time to provide cohesive functionalities. Efficient resource allocation to each service is difficult, while it becomes more challenging when the cluster consists of heterogeneous machines. Second, services involved in the application often have traffic demands among them due to data communication, which require meticulous treatment. Poor handling of the traffic demands can result in severe performance degradation, as the response time of a service is directly affected by its traffic situation. Considering the traffic demands, an intuitive solution is to place the services that have large traffic demands among them on the same machine, which can achieve intra-machine communication and reduce inter-machine traffic. However, such services cannot all be co-located on one machine due to the limited resource capacities. Hence, placement of service-based applications is quite complicated in clouds. In order to achieve a desired performance of a service-based application, cluster schedulers have to carefully place each service of the application with respect to the resource demands and traffic demands.

Recent cluster scheduling methods mainly focus on the cluster resource efficiency or job completion time of batch workloads. For instance, Tetris [66], a multi-resource cluster scheduler, adapts heuristics for the multi-dimensional bin packing problem to efficiently pack tasks on multi-resource cluster. Firmament [61], a centralized cluster scheduler, can make high-quality placement decisions on large-scale clusters via a min-cost max-flow optimization. Unfortunately, these solutions would face difficulties for handling service-based applications, as they ignore the traffic demands when making placement decisions. Some other research works [151, 69] concentrate on composite Software as a service (SaaS) placement problem, which try to minimize the

total execution time for composite SaaS. However, they only focus on a set of predefined service components for the application placement. For traffic-aware scheduling, relevant research solutions [104, 142] are proposed to handle virtual machine (VM) placement problem, which aims to optimize network resource usage over the cluster. However, these solutions rely on a certain network topology, while most of existing cluster schedulers are agnostic to network topology. In particular, it is hard to get the network topology information when deploying a service-based application on a virtual infrastructure.

In this chapter, we propose a new approach to optimize the placement of service-based applications in clouds. The objective is to minimize inter-machine traffic while satisfying the multi-resource demands of service-based applications. Our approach involves two key steps: 1) The requested application is partitioned into several parts while keeping overall traffic between different parts to a minimum. 2) The parts in the partition are packed into machines with multi-resource constraints. Typically, the partition can be abstracted as a minimum k-cut problem; the packing can be abstracted as a multi-dimensional bin packing problem. However, both are NP-hard problems [65, 146]. To address these problems, we first propose two partition algorithms: *Binary Partition* and *K Partition*, which are based on a well designed randomized contraction algorithm [85], for finding a high quality application partition. Then, we propose a packing algorithm, which adopts an effective packing heuristic with traffic awareness, for efficiently packing each part of an application partition into machines. Finally, we combine the partition and packing algorithm with a resource demand threshold to find an appropriate placement solution. We implement a prototype scheduler based on our proposed algorithms, and evaluate it in testbed clusters. The results show that our scheduler outperforms existing container cluster schedulers and representative heuristics, leading to much less overall inter-machine traffic.

4.2 Problem Formulation

In this section, we formulate the placement problem of service-based application, and introduce the objective of this work. The notation used in the work is presented in Table 4.1.

4.2.1 Model Description

We consider a cloud computer cluster is composed of a set of heterogeneous machines $\mathcal{M} = \{m_1, m_2, ..., m_M\}$, where $M = |\mathcal{M}|$ is the number of machines. We consider R types of resources $\mathcal{R} = \{r_1, r_2, ..., r_R\}$ (e.g., CPU, memory, disk, etc.) in each machine. For machine m_i, let $V_i = (v_i^1, v_i^2, ..., v_i^R)$ be the vector of

Table 4.1: Notation and Description

Notation	Description
$\mathcal{M}$	Set of heterogeneous machines in the cluster: $\mathcal{M} = \{m_1, m_2, ..., m_M\}$
M	Number of the machines: $M = \|\mathcal{M}\|$
$\mathcal{R}$	Set of resource types: $\mathcal{R} = \{r_1, r_2, ..., r_R\}$
R	Number of the resource types: $R = \|\mathcal{R}\|$
V_i	Vector of available resources on machine m_i: $V_i = (v_i^1, v_i^2, ..., v_i^R)$
v_i^j	Amount of resource r_j available on machine m_i
$\mathcal{S}$	A service-based application which is composed by a set of services: $\mathcal{S} = \{s_1, s_2, ..., s_N\}$
N	Number of services in the application: $N = \|\mathcal{S}\|$
D_i	Vector of resource demands of service s_i: $D_i = (d_i^1, d_i^2, ..., d_i^R)$
d_i^j	Amount of resource r_j that service s_i demands
$\mathbb{T}$	Matrix of communication traffic between services: $\mathbb{T} = [t_{ij}]_{N \times N}$
t_{ij}	Traffic rate from service s_i to service s_j
$\mathbb{X}$	A placement solution: $\mathbb{X} = [x_{ij}]_{N \times M}$, where $x_{ij} = 1$ if service s_i is to be placed on machine m_j, otherwise $x_{ij} = 0$

its available resources, where the element v_i^j denotes the amount of resource r_j available on machine m_i.

We consider a service-based application is composed of a set of services $\mathcal{S} = \{s_1, s_2, ..., s_N\}$ that are to be deployed on the cluster, and $N = |\mathcal{S}|$ is the number of services. For service s_i, let $D_i = (d_i^1, d_i^2, ..., d_i^R)$ be the vector of its resource demands, where the element d_i^j denotes the amount of resource r_j that the service s_i demands. Let matrix $\mathbb{T} = [t_{ij}]_{N \times N}$ denote the traffic between services, where t_{ij} denotes the traffic rate from service s_i to service s_j.

We model a placement solution as a 0-1 matrix $\mathbb{X} = [x_{ij}]_{N \times M}$. if service s_i is to be deployed on machine m_j, it is $x_{ij} = 1$. Otherwise, it is $x_{ij} = 0$.

4.2.2 Objective

To achieve a desired performance of service-based applications, a scheduler should not only consider the multi-resource demands of services, but also the traffic situation between services. As services, especially data-intensive services, often need to transfer data frequently, the network performance would

directly influence the overall performance. Considering the network dynamics, the placement of different services of an application is crucial for maintaining the overall performance, particularly when unexpected network latency or congestion occurs in the cluster. Given the traffic situation, the most intuitive solution is to place the services that have high traffic rate among them on the same machine, so that the co-located services can leverage the loopback interface to get a high network performance without consuming actual network resources of the cluster. However, such services cannot all be co-located on one machine due to the limited resource capacities. Thus, with the resource constraints, we try to find a placement solution to minimize the overall traffic between services that are placed on different machines (inter-machine traffic) while satisfying multi-resource demands of services, so that the objective of this work can be formulated as:

$$\textbf{Minimize} \quad \sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{p=1}^{M}\sum_{\substack{q=1 \\ q\neq p}}^{M} t_{ij} \cdot x_{ip} \cdot x_{jq} \tag{4.1}$$

$$\textbf{Subject to:} \quad \sum_{j=1}^{M} x_{ij} = 1 \ (\forall i \in \{1, 2, ..., N\}) \tag{4.2}$$

$$\sum_{i=1}^{N} x_{ij} \cdot d_i^k \leq v_j^k \ (\forall j \in \{1, 2, ..., M\}, \forall k \in \{1, 2, ..., R\}) \tag{4.3}$$

$$x_{ij} \in \{0, 1\} \ (\forall i \in \{1, 2, ..., N\}, \forall j \in \{1, 2, ..., M\}) \tag{4.4}$$

Equation 4.2 guarantees that each service is placed on a machine. Equation 4.3 guarantees that resource demands on a machine do not exceed its resource capacities. Equation 4.1 expresses the goal of this work.

4.3 Minimum K-Cut Problem

As a service-based application typically cannot be placed on one machine, an effective partition of the set of services involved in the application is necessary during the deployment. After partition, each subset of the services should be able to be packed into a machine, which means the machine has sufficient resources to run all the services in the subset. Considering the traffic rate between different services, the quality of the partition is crucial for the application performance. To tackle this problem, we first discuss the minimum k-cut problem to understand the problem complexity.

Let $G = (V, E)$ be an undirected graph, where V is the node set and E is the edge set. In the graph, each edge $e_{u,v} \in E$ has an non-negative weight $w_{u,v}$. A

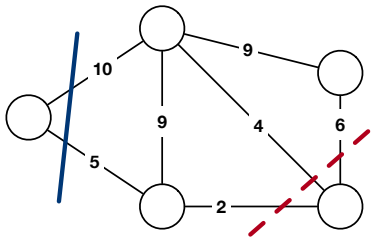


Figure 4.1: An example of a minimum cut (dash line)

k-cut in graph G is a set of edges, which when removed, partition the graph into k disjoint nonempty components $G' = \{G_1, G_2, ..., G_k\}$. The minimum k-cut problem is to find a k-cut of minimum total weight of edges whose two ends are in different components, which can be computed as:

$$\sum_{i=1}^{k-1} \sum_{j=i+1}^{k-1} \sum_{\substack{u \in G_i \\ v \in G_j}} w_{u,v} \tag{4.5}$$

A minimum cut is a simply minimum k-cut when $k = 2$. Figure 4.1 shows an example of a minimum cut of a graph. There are 2 cuts shown in the figure, and the dash line is a minimum cut of the graph, as the total weight of edges cut by the dash line is the minimum of all cuts. Given a service-based application, we can represent it as a graph, where the nodes represent services and the weights of edges represent the traffic rate. Specifically, the traffic rate from service s_i to service s_j and the rate from service s_j to service s_i are represented as two edges respectively in the graph. Hence, finding a minimum k-cut of the graph is equivalent to partitioning the application into k parts while keeping overall traffic between different parts to a minimum. However, for arbitrary k, the minimum k-cut problem is NP-hard [65].

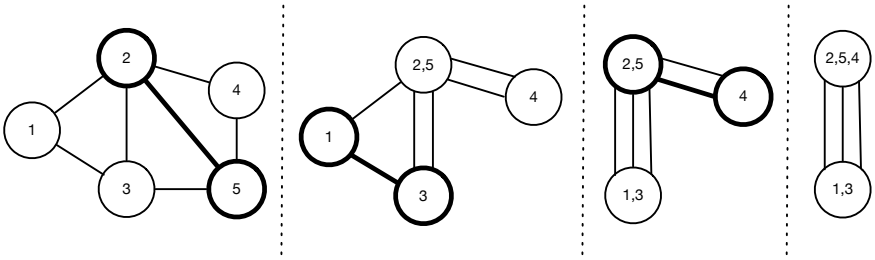


Figure 4.2: An example process of the contraction algorithm (k=2)

Different from developing a deterministic algorithm, Karger's algorithm [85] provides an efficient randomized approach to find a minimum cut of a graph. The basic idea of the Karger's algorithm is to randomly choose an edge $e_{u,v}$ from the graph with probability proportional to the weight of edge $e_{u,v}$, and merge the node u and node v into one (called edge contraction). In order to find a minimum cut, the algorithm iteratively contracts the edge which are randomly chosen until two nodes remain. The edges that remain at last are then output by the algorithm. The pseudocode is shown in Algorithm 3.

Algorithm 3: Contraction Algorithm (k=2)

Input: $G = (V, E)$
Output: A cut of G
while $|V| > 2$ **do**
    choose an edge $e_{u,v}$ with probability proportional to its weight;
    $G \leftarrow G/e_{u,v}$; `// contract edge` $e_{u,v}$
end
return the cut in G;

Figure 4.2 shows an example process of the contraction algorithm ($k = 2$). The algorithm iteratively merges two nodes of the chosen edge, and all other edges are reconnected to the merged node. For a graph $G = (V, E)$ with $n = |V|$ nodes and $m = |E|$ edges, Karger [85] argues that the contraction algorithm returns a minimum cut of the graph with probability $\Omega(1/n^2)$. Therefore, if we perform the contraction algorithm independently $n^2 \log n$ times, we can find a minimum cut with high probability, as the probability we do not get a minimum cut is less than $\Omega(1/n)$. For minimum k-cut, the contraction algorithm is basically the same, except that it terminates when k nodes remain (change $|V| > 2$ to $|V| > k$ in Algorithm 3), and returns all the edges left in the graph G. Similarly, the contraction algorithm returns a minimum k-cut of the graph with probability $\Omega(1/n^{2k-2})$. If we perform the algorithm independently $n^{2k-2} \log n$ times, we can obtain a minimum k-cut with high probability. Regarding the time complexity, the contraction algorithm can be implemented to run in strongly polynomial $O(m\log^2 n)$ time [85].

4.4 Placement Algorithm

In this section, we describe the algorithms we proposed in this work. The goal of our algorithms is to find a placement solution to minimize inter-machine traffic while satisfying multi-resource demands. The key design of our approach includes: 1) application partition based on contraction algorithms, 2)

heuristic packing with traffic awareness, and 3) placement finding with threshold adjustment.

4.4.1 Application Partition

In order to make the values of different resources comparable to each other and easy to handle, we first normalize the amount of available resources on machines and the resources that services demands to be the fraction of the maximum ones. We define the term v_{max-j} to be the maximum amount of available resource r_j on a machine.

$$v_{max-j} = \max_{i \in \{1,2,...,M\}} \left(v_i^j \right) \tag{4.6}$$

Then the vector V_i of available resources on machine m_i and the vector D_i of resource demands of service s_i are normalized as:

$$V_i = (\frac{v_i^1}{v_{max-1}}, \frac{v_i^2}{v_{max-2}}, ..., \frac{v_i^R}{v_{max-R}}) \tag{4.7}$$

$$D_i = (\frac{d_i^1}{v_{max-1}}, \frac{d_i^2}{v_{max-2}}, ..., \frac{d_i^R}{v_{max-R}}) \tag{4.8}$$

After normalization, we start partitioning the service-based application. The key question we ask first is how many parts the application is partitioned into. Considering multi-resource demands of different services, we introduce a threshold α to determine the number of parts when performing partition algorithms. The threshold α denotes the upper bound of the resource demands of partitioned parts, which means we perform partition algorithms continuously until the total resource demands from each part do not exceed α or no part contains more than one service. With a threshold $\alpha \in [0, 1]$ (as the resource demands have been normalized), it assures that each part after partition can be packed into a machine. Figure 4.3 shows an example of an application partition with threshold $\alpha = 0.5$. In the figure, the total CPU demands and memory demands from each part do not exceed 0.5. Given a threshold α, we propose two partition algorithms: *binary partition* and *k partition*, which are based on the contraction algorithm.

Binary Partition

The idea of the binary partition algorithm is to continuously perform binary partition on the application until the resource demands from each part do not exceed α or no part contains more than one service. The pseudocode is shown

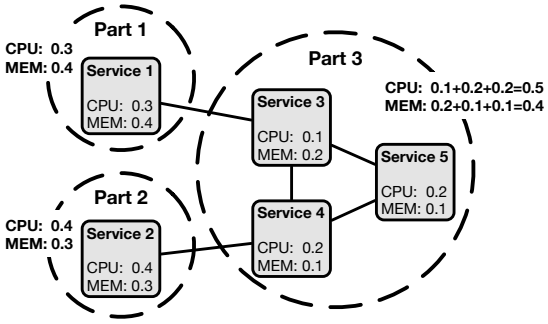


Figure 4.3: An example of an application partition with threshold $\alpha = 0.5$

Algorithm 4: Binary Partition

Input: service-based application $\mathcal{S}$, threshold α
Output: a partition of the application $P = \{\mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_{N'}\}$, N' is number of parts after partition

$P \leftarrow \{\mathcal{S}\}$;
while *exists part* $\mathcal{S}_i$ *in* P *that the total resource demands exceed* α *and part* $\mathcal{S}_i$ *contains more than one service* **do**
  $P \leftarrow P - \{\mathcal{S}_i\}$;
  Construct a graph $G = (V, E)$ based on $\mathcal{S}_i$;
  $n \leftarrow |V|$;
  $G_{min} \leftarrow G$;
  $t \leftarrow 0$;
  repeat
    Perform the contraction algorithm ($k = 2$) to get a cut G';
    $G_{min} \leftarrow \min(G_{min}, G')$; `// Store the smaller cut in` G_{min}
    $t \leftarrow t + 1$;
  until $t > n$;
  Get a partition $\{\mathcal{S}_x, \mathcal{S}_y\}$ of part $\mathcal{S}_i$ according to G_{min};
  $P \leftarrow P \cup \{\mathcal{S}_x, \mathcal{S}_y\}$;
end
return P;

in Algorithm 4. The basic process can be described as follows. The algorithm continuously checks the resource demands of each part in current application partition P. The initial partition is $P = \{\mathcal{S}\}$ where the entire application is treated as one part. If the total resource demands of a part $\mathcal{S}_i$ in P exceeds the threshold α and part $\mathcal{S}_i$ contains more than one service, the part is selected

to be partitioned into 2 parts (binary partition). It first constructs a graph $G = (V, E)$ based on $\mathcal{S}_i$, where the nodes represent services and the weights of edges represent the traffic rate. As mentioned in section 4.3, if we repeatedly perform the contraction algorithm many times we can obtain a minimum cut with high probability. Considering both the partition quality and the partition speed, we choose to perform the contraction algorithm n times in our algorithm (In offline manner, it can be set to run $n^2 \log n$ times to get a minimum cut with high probability). Then, according to the minimum cut G_{min} we get from the contraction algorithm, it partitions the $\mathcal{S}_i$ into two parts $\{\mathcal{S}_x, \mathcal{S}_y\}$. This process would be repeatedly performed until the resource demands from each part do not exceed threshold α or no part contains more than one service.

K Partition

Algorithm 5: K Partition

Input: service-based application $\mathcal{S}$, threshold α
Output: a partition of the application $P = \{\mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_{N'}\}$, N' is number of parts after partition

```
1  P ← {S};
2  Construct a graph G = (V, E) based on S;
3  n ← |V|;
4  k ← 1;
5  while exists part S_i in P that the total resource demands exceed α and
     part S_i contains more than one service do
6      G_min ← G;
7      k ← k + 1;
8      t ← 0;
9      repeat
10         Perform the contraction algorithm until k nodes remain to get a
             k-cut G';
11         G_min ← min(G_min, G') ;        // Store the smaller k-cut in
             G_min
12         t ← t + 1;
13     until t > n;
14     Get a partition {S_1, S_2, ..., S_k} of the application S according to
         G_min;
15     P ← {S_1, S_2, ..., S_k};
16 end
17 return P;
```

The idea of the k partition algorithm is to directly partition the application

into k parts. By iteratively increasing k, it terminates when the resource demands from each part do not exceed α or no part contains more than one service. The pseudocode is shown in Algorithm 5. The basic process can be described as follows. The algorithm first constructs a graph $G = (V, E)$ based on the application $\mathcal{S}$, and then continuously checks the resource demands of each part in current application partition P where $P = \{\mathcal{S}\}$ initially. If the total resource demands of a part $\mathcal{S}_i$ in P exceeds the threshold α and part $\mathcal{S}_i$ contains more than one service, it increases k which is the number of partitioned parts. As mentioned in section 4.3, in order to obtain a minimum k-cut with high probability, we have to perform the contraction algorithm independently $n^{2k-2} \log n$ times. However, the time complexity increases exponentially with k, which is prohibitively high. Thus, we make the time complexity consistent with the binary partition algorithm by sacrificing some probability of finding a minimum k-cut. It also performs the contraction algorithm n times. Then, according to the minimum k-cut G_{min} we get from the contraction algorithm, it partitions the application into k parts $P = \{\mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_k\}$. Similarly, this process would be repeatedly performed until the resource demands from each part do not exceed threshold α or no part contains more than one service.

4.4.2 Heuristic Packing

Given a partition of the application, the algorithm here is to pack each part into the heterogeneous machines. Without considering the traffic rate, the problem can be formulated as a classical multi-dimensional bin packing problem, which is known to be NP-hard [146]. When there are a large amount of services involved in the application, it is infeasible to find the optimal solution in polynomial time. Considering the time complexity and packing quality, we adopt two greedy heuristics in our packing algorithm: *Traffic Awareness* and *Most-Loaded Heuristic*. The algorithm is shown in Algorithm 6.

In order to find a best possible machine for part $\mathcal{S}_i$, the algorithm calculates two matching factors: tf and ml. For machine m_j, the factor tf is the sum of the traffic rate between the services in part $\mathcal{S}_i$ and the services have been determined to be packed into machine m_j before. The factor ml is a scalar value of the load situation between the vector of resource demands from part $\mathcal{S}_i$ and the vector of available resources on machine m_j. Assuming D'_i is the resource demand vector of part $\mathcal{S}_i$ and ${d'}_i^k$ is the amount of resource r_k part $\mathcal{S}_i$ demands, it is $ml = \sum_{k=1}^{R} \frac{{d'}_i^k}{v_j^k}$. The higher ml is, the more loaded the machine. The idea of this heuristic is improve the resource efficiency by packing the part to the most loaded machine. As our main goal is to minimize the inter-machine traffic, the algorithm is designed to first prioritize the machines based on the factors of tf. If the factors of tf are the same, it then prioritizes the machines based on the factors of ml. Consequently, if all parts in the partition can be

Algorithm 6: Heuristic Packing

Input: partition of the application $P = \{\mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_{N'}\}$, vectors of available resources on each machine $\{V_1, V_2, ..., V_M\}$

Output: a placement solution X

```
1  Calculate vectors of resource demands of each part as: {D'_1, D'_2, ..., D'_N'};
2  X ← [x_ij = 0]_{N'×M};
3  for i ← 1; i ≤ N'; i ← i + 1 do
4      tf ← 0; ml ← 0; y ← 0;
5      for j ← 1; j ≤ M; j ← j + 1 do
6          if part S_i can be packed into machine m_j then
7              tf_j ← Σ t_uv;
               /* Calculate the total traffic rates between part S_i
                  and machine m_j, for any service s_u in S_i and any
                  service s_v packed into machine m_j before          */
8              ml_j ← Σ_{k=1}^{R} d'^k_i / v^k_j ;
               /* Calculate the load situation between the vector
                  of resource demands from part S_i and the vector
                  of available resources on machine m_j               */
9              if tf_j > tf then
10                 tf ← tf_j; ml ← ml_j; y ← j;
11             end
12             else if tf_j == tf and ml_j > ml then
13                 tf ← tf_j; ml ← ml_j; y ← j;
14             end
15         end
16     end
17     if y == 0 then
18         return null;
19     end
20     else
21         V_y ← V_y − D'_i;
22         x_iy ← 1;
23     end
24 end
25 return X;
```

packed into machines, the algorithm returns the placement solution. Otherwise, it returns *null*.

4.4.3 Placement Finding

Algorithm 7: Placement Finding

Input: service-based application $\mathcal{S}$, vectors of available resources on each machine $\{V_1, V_2, ..., V_M\}$
Output: a placement solution X

```
1  X ← [x_ij = 0]_{N×M};
2  α ← 1.0;
3  Δ ← 0.1;
4  while α ≥ 0.0 do
5  |   P ← Binary_Partition(S, α);
   |   /* Or  P ← K_Partition(S, α);                    */
6  |   X' ← Heuristic_Packing(P, {V_1, V_2, ..., V_M});
7  |   if X' ≠ null then
8  |   |   Calculate X according to X' and P;
9  |   |   return X;
10 |   end
11 |   α ← α − Δ;
12 end
13 return null;
```

As we discussed before, in order to partition the application, the threshold α is required by the algorithm. However, giving an appropriate deterministic threshold α is difficult, as it cannot guarantee that the algorithm can find a placement solution through the randomized partition and the heuristic packing under a certain threshold α. Intuitively, the higher threshold α results in less parts in the partition, which leads to less traffic rate between different parts. Thus, we introduce a simple algorithm to find a better threshold α by enumerating from large to small. The algorithm is shown in Algorithm 7. At the beginning, the value of α is 1.0. To adjust the thresholds, we set a step value Δ, and the default value is 0.1, which can be customized by users. In each iteration, with the threshold α, the algorithm first partitions the given application $\mathcal{S}$ based on the binary partition algorithm or k partition algorithm. Note that the algorithm records the latest partition results to avoid multiple repeated partition. It then tries to pack all parts in the partition into machines based on the heuristic packing algorithm to find a placement solution for the application.

Next, we discuss the time complexity of the algorithm we proposed. We assume the number of services is n; the number of edges in the service graph is m (i.e., the number of the traffic rates $t_{ij} > 0$); the number of machines is M. For a service-based application, it can be partitioned up to n parts. For each partition, we perform the contraction algorithm n times, and the time complexity of the contraction algorithm is $O(m\log^2 n)$. As we record the latest partition results to avoid multiple repeated partition, the time complexity of the overall partition is $O(n^2 m\log^2 n)$. To the heuristic packing, the time complexity is $O(nM + n^2)$ as the overall time complexity of calculating the factor tf is $O(n^2)$. Let $C = \frac{1}{\Delta}$ denote the number of iterations. The overall time complexity of the proposed algorithm is $(n^2 m\log^2 n + CnM + Cn^2)$.

4.5 Evaluation

We implement a prototype scheduler using python, which is based on our proposed algorithms, for deploying service-based applications on container clusters. In the experiments, we evaluate our scheduler in testbed clusters of ExoGENI [34] experimental environment.

4.5.1 Experimental Methodology

Cluster. We create two different testbed clusters in ExoGENI for experiments. For the first cluster, we use 30 homogeneous VMs with 2 CPU cores and 6 GB RAM. Considering the heterogeneity, we use 10 VMs with 2 CPU cores and 6 GB RAM, and 10 VMs with 4 CPU cores and 12 GB RAM for the second cluster. The homogeneous cluster has 30 VMs and the heterogeneous cluster has 20 VMs, but the total resource capacity is the same.

Workloads. In order to evaluate the proposed algorithms in different scenarios, we use synthetic applications in the experiments. Considering the scale of the testbed cluster, we yield service-based applications which are composed by 64, 96, and 128 services. For the size of 64, the CPU demand of each service is uniformly picked at random from [30,100] where 100 represents 1 CPU core, and the memory demand is picked at random from [100,300] where 100 represents 1 GB RAM. For the size of 96, the CPU demand is picked at random from [20,67], and the memory demand is picked at random from [67,200]. For the size of 128, the CPU demand is picked at random from [15,50], and the memory demand is picked at random from [50,150]. According to these ranges, the total resource demands of different application sizes are roughly the same. For each application size, we generate 10,000 instances for testing. As the work [38] shows that the log-normal distribution produces the best fit to the data center traffic, we choose to generate the traffic demands between services with the probability 0.05 (ensure that application graph is connected),

and the traffic rate follows a log-normal distribution (mean = 5 Mbps, standard deviation = 1 Mbps).

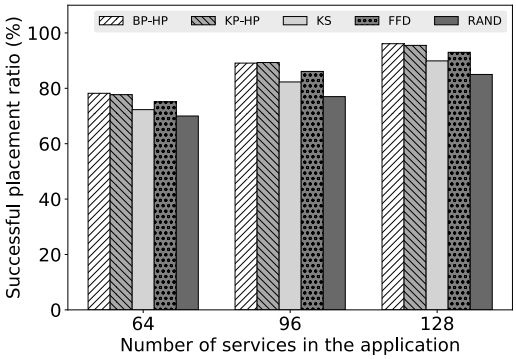
Implementation. We implement all proposed algorithms in our prototype scheduler, where the contraction algorithm is based on the parallel implementation [85]. As we proposed two algorithms for application partition, there are two kinds of configuration. BP-HP is based on binary partition (BP) and heuristic packing (HP). KP-HP is based on k partition (KP) and heuristic packing.

Baselines. We compare our scheduler with the following schemes:

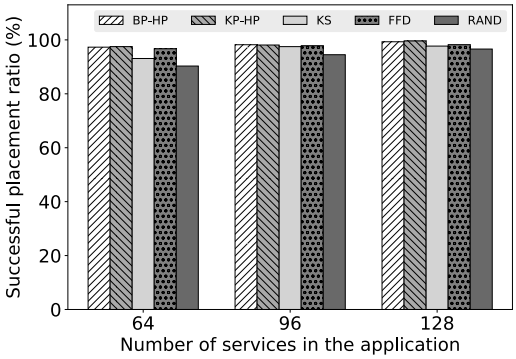
- Kubernetes Scheduler (KS): the default scheduler in Kubernetes [71] container cluster tends to distribute containers evenly across the cluster to balance the overall cluster resource usage. Specifically, we add a soft affinity (i.e., pod affinity in Kubernetes) to the services that have traffic between them, as the scheduler would try to place the services which have affinity between them on the same machine.
- First-Fit Decreasing (FFD): it is a simple and commonly adopted algorithm for the multi-dimensional bin packing problem [24]. FFD operates by first sorting the services in decreasing order according to a certain resource demand, and then packs each service into the first machine with sufficient resources.
- Random (RAND): it randomly picks a service in the application, and then packs it into the first machine with sufficient resources.

4.5.2 Comparison with Baselines

Figure 4.4 shows the successful placement ratio of different schemes over two clusters. The successful placement of a application is that the algorithm can find a placement solution to place all the involved services, so the ratio is the number of successfully placed applications to the number of all requested applications. We observe that RAND performs worst, as it has no heuristic to pack the services. FFD performs better than KS, because KS mainly focuses on balancing the resource utilization over the cluster while FFD has been demonstrated as an effective algorithm for multi-dimensional bin packing problems [128]. BP-HP performs comparably to KP-HP, and they both slightly outperform other schemes in this evaluation. This is mainly because the iterative partition and packing with different thresholds improve the probability of finding a placement solution. Moreover, the packing algorithm can pack services tightly due to the most-loaded heuristic. The results of the homogeneous cluster also show that the successful placement ratio increases when the number of services increases. As the total resource demands of the applications in different sizes (different number of services) are roughly the same, the less number of services results in larger resource demands of each individual service, which



(a) Cluster with Homogeneous Machines

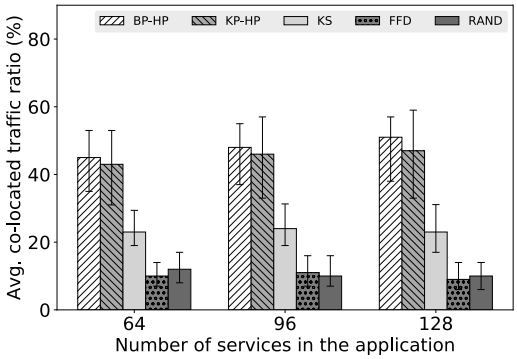


(b) Cluster with Heterogeneous Machines

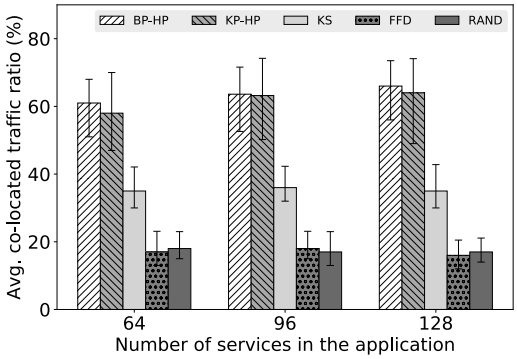
Figure 4.4: Comparison of successful placement ratio of different schemes

easily causes the resource fragmentation problem in the placement. Compared to the homogeneous cluster, the successful placement ratio is much higher in the heterogeneous cluster. As the machines have larger resource capacity in the heterogeneous cluster, it is easier to pack services constrained by multiple resources.

Next, we evaluate the traffic situation of different schemes. In the evaluation, we only compare the applications whose all services are placed on the cluster by different algorithms. Figure 4.5 shows the average co-located traffic ratio of different schemes, and the error bars represent the maximum and minimum ratio. The co-located traffic is the traffic between the services that are placed on the same machine, so the ratio is the amount of co-located traffic to the



(a) Cluster with Homogeneous Machines



(b) Cluster with Heterogeneous Machines

Figure 4.5: Comparison of average co-located traffic ratio of different schemes

amount of all traffic. For minimizing inter-machine traffic, the higher the co-located traffic ratio is, the better the placement solution is. In the figure, we observe that BP-HP and KP-HP significantly outperform the baselines. For the cluster with homogeneous machines, BP-HP improves average co-located traffic ratio by 22% to 42%; KP-HP improves the ratio by 20% to 38%. For the cluster with heterogeneous machines, BP-HP improves average co-located traffic ratio by 26% to 50%; KP-HP improves the ratio by 23% to 48%. FFD and RAND perform worst as they only focus on packing the services, without considering the traffic rate. As we set the affinity to the services that have traffic between them in KS, KS tries to put the affinity services on the same machine. However, KS ignores the concrete traffic rate when making placement decisions.

Regarding BP-HP and KP-HP, we find that BP-HP performs slightly better and more stable than KP-HP, but KP-HP may find a better solution in some cases (according to the error bars). In contrast, KP-HP also easily returns a worse solution. This is mainly because BP-HP performs the contraction algorithm to find a minimum cut with probability $\Omega(1/n^2)$; KP-HP performs the contraction algorithm to find a minimum k-cut with probability $\Omega(1/n^{2k-2})$ which is much less than the BP-HP. Thus, the performance of KP-HP varies widely in the experiments. Nevertheless, benefiting from the partition that strives to co-locate the large traffic demands and the traffic-aware packing, BP-HP and KP-HP both can effectively reduce inter-machine traffic for deploying service-based applications on computer clusters.

4.5.3 Impact of Threshold α

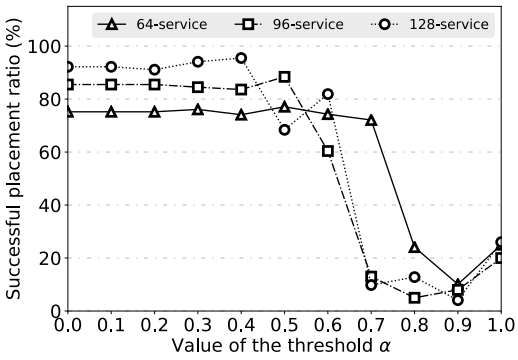


Figure 4.6: Successful placement ratio on the homogeneous cluster by using BP-HP with different values of threshold α

In this section, we discuss the impact of threshold α on the service-based application placement. To illustrate, we fix the threshold α by using BP-HP on the cluster with homogeneous machines. Figure 4.6 shows the successful placement ratio with different values of threshold α. For instance, BP-HP can find a placement solution for 77% of the applications with 64 services when $\alpha = 0.5$. We observe that the successful placement ratio decreases when the value of threshold α increases in general, and few applications can be successfully placed when $\alpha > 0.7$. Higher threshold α leads to less parts and larger average resource demands of parts in the partition, so it becomes harder to pack them into machines with multi-resource constraints. To understand the impact on the network traffic, Figure 4.7 shows the results of average co-located traffic

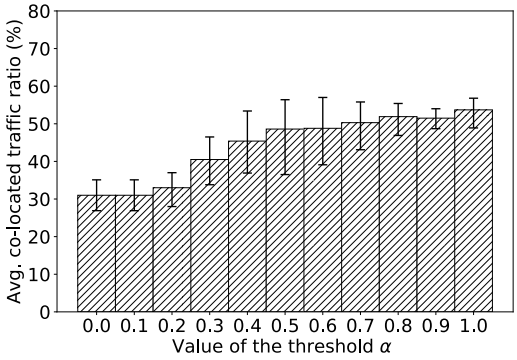


Figure 4.7: Average co-located traffic ratio on the homogeneous cluster by using BP-HP with different values of threshold α

ratio for each value of threshold α, and the error bars represent the maximum and minimum ratio. It explicitly demonstrates that the co-located traffic ratio increases more when α is larger. However, larger threshold α increases the difficulty of packing the applications. Thus, we try to find an appropriate threshold α by enumerating from large to small in the proposed algorithms.

4.5.4 Overhead Evaluation

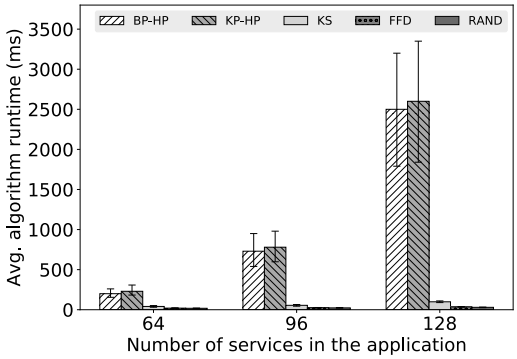


Figure 4.8: Average algorithm runtime of different schemes for the heterogeneous cluster

In this section, we evaluate the overhead by measuring the algorithm runtime

of different schemes. In order to fairly compare the algorithm runtime, we also implement the scheduling algorithm of KS in Python, which is the same with other schemes. We conduct this experiment on a dedicated server with Intel Xeon E5-2630 2.4GHz CPU and 64GB memory. Figure 4.8 shows the results of the average algorithm runtime of different schemes for the heterogeneous cluster (the homogeneous cluster is similar), and the error bars represent the maximum and minimum algorithm runtime. FFD and RAND incur little overhead, as they are simple packing algorithms. Compared with them, KS is a bit complex, as KS has multiple predicated policies and priorities policies to filter and score machines, such as handling the affinity between services. BP-HP and KP-HP are more complicated than the baselines, and have obviously higher overhead. We also observe that the difference between the maximum and minimum algorithm runtime is quite large, as the algorithm runtime heavily depends on the value of threshold α. In the algorithm, higher threshold α results in less iterations, and lower threshold α causes more iterations. Nevertheless, BP-HP and KP-HP can respond in seconds for different application sizes. Especially for the application with less than 100 services, BP-HP and KP-HP can respond in sub-second time, which is acceptable for online scheduling. Moreover, the most time consuming part of the proposed algorithms is application partition, which implies there would be no big difference of the algorithm runtime for large-scale clusters with the same number of services. We believe that the proposed algorithms could also effectively handle the placement problem on large-scale clusters.

4.6 Related Work

As the microservice architecture is emerging as a primary architectural style choice in the service oriented software industry [131], many research efforts have been devoted to the analysis and modeling of microservice architecture [44, 70, 53]. Leitner et al. [95] proposed a graph-based cost model for deploying microservice-based applications on a public cloud. Balalaie et al. [33] presented their experience and lessons on migrating a monolithic software architecture to microservices. Amaral et al. [28] evaluated the performance of microservices architectures using containers. However, the performance of service placement schemes received little attention in these works.

Software as a Service (SaaS) is one of the most important services offered by cloud providers, and many works have been proposed for optimizing composite SaaS placement in cloud environments [78]. Yusoh et al. [151] propose a genetic algorithm for the composite SaaS placement problem, which considers both the placement of the software components of a SaaS and the placement of data of the SaaS. It tries to minimize the total execution time of a composite SaaS. Hajji et al. [69] adopt a new variation of PSO called Particle Swarm Optimization with

Composite Particle (PSO-CP) to solve the composite SaaS placement problem. It considers not only the total execution time of the composite SaaS, but also the performance of the underlying machines. Unfortunately, they target at the placement for a certain set of predefined service components, which has limitations to handle a large number of different services.

In recent years, a number of research works have been proposed in the area of VM placement with traffic awareness for cloud data centers [25, 88]. Meng et al. [104] analyze the impact of data center network architectures and traffic patterns, and propose a heuristic approach to reduce the aggregate traffic when placing VM into the data center. Wang et al. [142] formulate the VM placement problem with dynamic bandwidth demands as a stochastic bin packing problem, and propose an online packing algorithm to minimize the number of machines required. However, they only focus on optimizing the network traffic in the data center, without considering the highly diverse resources requirements of the virtual machines. Biran et al. [39] proposed a placement scheme to satisfy the traffic demands of the VMs while meeting the CPU and memory requirements. Dong et al. [52] introduced a placement solution to improve network resource utilization in addition to meeting multiple resource constraints. They both rely on a certain network topology to make placement decisions. Different from them, our work is agnostic to the network topology, which aims to minimize the overall inter-machine traffic on the cluster.

4.7 Conclusion

In this chapter, we investigated the placement problem of service-based applications in clouds. In order to find a high quality partition, we propose two partition algorithms: *Binary Partition* and *K Partition*, which are based on a well designed randomized contraction algorithm. For efficiently packing the application, we adopt most-loaded heuristic and traffic awareness in the packing algorithm. By adjusting the threshold α which denotes the upper bound of the resource demands, we can find a better placement solution for service-based applications. We implement a prototype scheduler based on our proposed algorithms, and evaluate it in testbed clusters. In the evaluation, we show that our algorithms can improve the ratio of successfully placing applications on the cluster while significantly increasing the ratio of co-located traffic (i.e., reducing the inter-machine traffic). In the overhead evaluation, the results show that our algorithms incur some overhead, but in an acceptable time. We believe that the proposed algorithms are practical for realistic use cases.

5 Learning Scheduling Policies for DAG jobs

In this chapter, we investigate how to learn scheduling policies of DAG jobs with deep reinforcement learning on multi-resource clusters. Efficiently scheduling DAG jobs on distributed computer clusters requires intricate algorithms, since the scheduler has to consider all the characteristics of cluster and DAG jobs to make scheduling decisions. To address this problem, we present GoTask, a deep reinforcement learning based approach that can learn to well schedule DAG jobs on multi-resource clusters.

This chapter is based on:

- **Hu, Y.**, de Laat, C. and Zhao, Z. Learning Workflow Scheduling on Multi-Resource Clusters. In 2019 IEEE International Conference on Networking, Architecture and Storage (NAS) (Pages 1-8). IEEE.
- **Hu, Y.**, de Laat, C. and Zhao, Z. Learning DAG Scheduling with Multi-Resource Constraints on Heterogeneous Clusters. Concurrency and Computation: Practice and Experience. (Under review).

5.1 Introduction

DAG scheduling problems are pervasive in data-parallel clusters. In parallel frameworks, applications usually can be modeled as a Directed-Acyclic Graph (DAG), where each vertex represents a task and edges encode precedence constraints. A task in a DAG relies on the outputs of the precedent tasks and cannot be started until all its required inputs are in place. Big data processing frameworks, such as Apache Hadoop [20], Apache Hive [132] and Spark SQL [29], and distributed scientific application frameworks [47] typically compile user scripts into DAG jobs. An ideal scheduler for DAG jobs is a scheduler that

can ensure that independent tasks run in parallel as many as possible, and no tasks in the waiting queue are blocked if there are available resources. To achieve this, the scheduler has to consider all the characteristics of cluster and DAG jobs, such as the cluster resource utilization, task resource demands (e.g., CPU, memory, network, etc.), task duration, and inter-task dependencies, which requires intricate algorithms. Figure 5.1 shows two schedules for a DAG job with 6 tasks. In this example, we only consider the CPU demand of each task, and try to run the DAG job on a cluster with single machine. We observe that the critical path schedule which schedules the task in the path with longest duration first completes in 27 timesteps, while the optimal schedule can complete in 17 timesteps. In order to achieve the optimal schedule, all the aspects of the cluster and the DAG job have to be taken into account. When the DAG job has multiple resource requirements and the cluster is composed of a set of heterogeneous machines, the problem becomes extremely complicated.

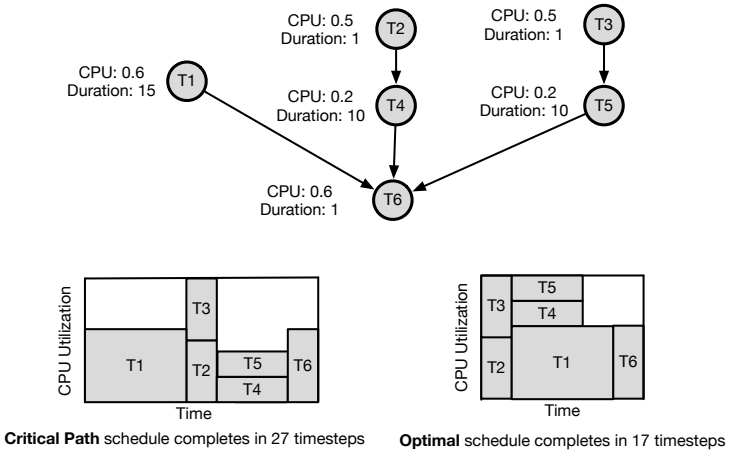


Figure 5.1: Critical path schedule and optimal schedule for a DAG job

In recent years, DAG scheduling problems have attracted quite a lot of research attention. As scheduling DAG jobs in a cluster is an NP-complete problem in general [91], many schedulers strive to provide better greedy or heuristic algorithms for better application performance or resource efficiency [32, 66]. For example, Graphene scheduler [68], aiming at jobs that have a complex dependency structure and heterogeneous resource demands, schedules troublesome tasks first and then schedule the remaining tasks to improve job completion time. However, they need to compute a offline schedule first, which is enforced by online scheduling. As another example, a workflow scheduler [22] recursively schedules the critical path ending at a recently

scheduled node to minimize the cost of workflow execution, but the resource requirements of the tasks are not considered in the scheduling algorithm. Moreover, many of the existing works are designed for some specific workloads or some specific metrics. As the workloads of request or the metrics of interest changes, researchers have to come up with new heuristics to adapt the new situation.

With these challenges, we investigate how to apply machine learning techniques, specifically deep reinforcement learning [129], to handle the DAG scheduling problem. That is to say how to make the system learn to schedule DAG jobs on their own. Reinforcement learning is to produce agents that interact with their environments to learn optimal behaviors. The agents will improve over time through trials and errors. At the beginning, the agent is not told which actions to take for a task. Then, the agent tries to interact with the environments to learn which actions yield the most reward that it receives based on how well it is doing on the task, which gradually helps the agent to make better decisions. Due to recent advances in deep learning, applying deep neural networks in reinforcement learning can make it possible to deal with more complex problems which have high-dimensional states or actions, such as playing Atari game [108], mastering the game of Go [126], etc. Thus, the breakthrough of deep reinforcement learning also provides a promising technique for dealing with DAG scheduling.

In this chapter, we present GoTask, an approach that can learn to well schedule DAG jobs with multi-resource constraints on heterogeneous clusters. GoTask directly learns the scheduling policy from experience through deep reinforcement learning, and the objective is to minimize the average job completion time. In order to handle the complexity and scale of the DAG scheduling problem, we propose a two-stage approach in GoTask, where the first stage leverages a deep reinforcement learning agent to learn policies for selecting a pending task of a DAG job, and the second stage leverages another agent to learn policies for selecting a machine to run the selected task. Moreover, to facilitate the learning of scheduling policy, we adopt a longest/critical path based approach for state encoding in task selection stage, and a fitness score based approach for packing heuristic encoding in machine selection stage. We implement a GoTask prototype and a simulator for simulation of task execution on multi-resource clusters. In the evaluation, the experiment results show that GoTask can effectively learn scheduling policies of DAG jobs from experience and outperforms commonly adopted scheduling heuristics.

5.2 Problem Formulation

In this section, we present the model of the DAG scheduling problem and the objective of this work. The notation used in the work is presented in Table 5.1.

Table 5.1: Notation and Description

Notation	Description
$\mathcal{M}$	Set of heterogeneous machines in the cluster: $\mathcal{M} = \{m_1, m_2, ..., m_M\}$
M	Number of the machines: $M = \|\mathcal{M}\|$
$\mathcal{R}$	Set of resource types: $\mathcal{R} = \{r_1, r_2, ..., r_R\}$
R	Number of the resource types: $R = \|\mathcal{R}\|$
V_i	Vector of available resources on machine m_i: $V_i = (v_i^1, v_i^2, ..., v_i^R)$
v_i^j	Amount of resource r_j available on machine m_i
$\mathcal{J}$	A DAG job is composed by a set of tasks $\mathcal{J} = \{t_1, t_2, ..., t_N\}$
N	Number of tasks in the DAG job: $N = \|\mathcal{J}\|$
D_i	Vector of resource demands of task t_i: $D_i = (d_i^1, d_i^2, ..., d_i^R)$
d_i^j	Amount of resource r_j that task t_i demands
s_i	Duration of task t_i
$\mathbb{P}$	0-1 Matrix of inter-task dependencies: $\mathbb{P} = [p_{ij}]_{N \times N}$, where $p_{ij} = 1$ if task t_i is a preceding task of task t_j.

5.2.1 Model Description

A cluster is typically composed of a set of heterogeneous machines $\mathcal{M} = \{m_1, m_2, ..., m_M\}$, where $M = |\mathcal{M}|$ is the number of machines. We consider R types of resources $\mathcal{R} = \{r_1, r_2, ..., r_R\}$ (e.g., CPU, memory, or network bandwidth) in each machine. For machine m_i, let $V_i = (v_i^1, v_i^2, ..., v_i^R)$ be the vector of its resource capacities where the element v_i^j denotes the total amount of resource r_j available on machine m_i.

We model a DAG job as a set of tasks $\mathcal{J} = \{t_1, t_2, ..., t_N\}$ that are to be executed on the cluster, and $N = |\mathcal{J}|$ is the number of tasks. For task t_i, let $D_i = (d_i^1, d_i^2, ..., d_i^R)$ be the vector of its resource demands, where the element d_i^j denotes the amount of resource r_j that the task t_i demands. We assume that the tasks' durations and the inter-task dependencies are known when a job request arrives, as it is found as the fact that many jobs are recurring and compute

on similar input data in compute clusters [56]. Therefore, tasks' durations and inter-task dependencies from a previous execution of a job can infer a future run of the same job [68]. Hence, let s_i be the duration of task t_i in our model. For dependency specification, let 0-1 matrix $\mathbb{P} = [p_{ij}]_{N \times N}$ denote the inter-task dependencies. If $p_{ij} = 1$, it means that the task t_i is a preceding task of task t_j. During the execution, a task can only be started when all its preceding tasks are completed.

5.2.2 Objective

For simplicity, preemption is not allowed in the cluster, which means the resources must be allocated continuously from the time that the task starts until it is completed. When the scheduler schedules a task to a machine, it must make sure that the machine has sufficient resources to execute the task. The main objective in this work is to minimize the job completion time (JCT) (the difference between the job arrival time and the completion time of the last task).

5.3 Deep Reinforcement Learning

In the section, we briefly introduce deep reinforcement learning techniques [83, 129, 30] which we used in this work.

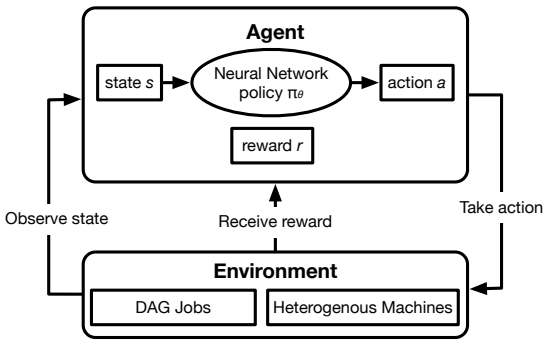


Figure 5.2: An example of reinforcement learning

5.3.1 Reinforcement Learning

We consider the standard reinforcement learning setting shown in Figure 5.2 where an agent interacts with an environment over a number of discrete

timesteps. At each timestep t, the agent receives a state $s_t \in \mathcal{S}$ through observation of the environment. The $\mathcal{S}$ is the state space which is a set of states. The agent then selects an action $a_t \in \mathcal{A}$ according to its policy π. The $\mathcal{A}$ is the action space which is a set of possible actions. The π is a mapping from states s to actions a; it denotes the probability of choosing different actions based on the states. Following the action, the environment transitions to the next state s_{t+1}, and the agent receives a scalar reward r_t. The state transitions and rewards are stochastic, which are assumed to have the Markov property. It means that the state transition probabilities and rewards depend only on the current state s_t of the environment and the action a_t taken by the agent. The process continues until the agent reaches a terminal state, and then the process restarts. The return of a policy is the cumulative reward that the agent receives in every timestep, which is represented as $R = \sum_{t=0}^{\infty} \gamma^t r_t$ with discount factor $\gamma \in [0, 1]$. The goal of the agent is to find an optimal policy π^*, which achieves the maximum expected return from all states:

$$\pi^* = \underset{\pi}{\operatorname{argmax}}\, \mathbb{E}[R|\pi] \tag{5.1}$$

5.3.2 Value Functions

The value function $V^\pi(s)$ is defined as the expected return when starting in state s and following the policy π:

$$V^\pi(s) = \mathbb{E}[R|s, \pi] \tag{5.2}$$

We represent the optimal policy as π^* and the corresponding value function as $V^*(s)$. The value function of the optimal policy can be defined as:

$$V^*(s) = \max_{\pi} V^\pi(s) \tag{5.3}$$

If the optimal value function $V^*(s)$ is known, the optimal policy π^* can be easily obtained by picking the action a_t (among all actions available at state s_t) that maximizes $r_t + V^*(s_{t+1})$.

In small cases, tabular methods or non-parametric methods [129] can be used to compute the $V^*(s)$. However, there are too many possible states in most practical problems [109, 125], including the DAG scheduling problem in this work. It is impossible to store the policy in a tabular form. Hence, the value function is commonly represented using a function approximator [108], such as neural networks. In neural networks, there are a certain number of adjustable parameters θ. Let $\pi_\theta(s, a)$ be the probability of taking action a in the state s given the parameters θ. Thus, different kinds of policies can be derived from adjusting the parameters θ of the neural network. Consequently, we obtain

deep reinforcement learning methods when we use deep neural networks to approximate the value function and the policy with different parameters.

5.3.3 Actor-Critic Method

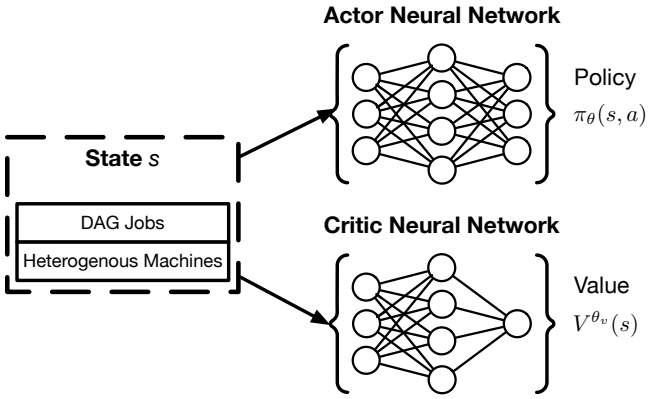


Figure 5.3: The architecture of actor-critic method

In this work, we focus on policy gradient methods [129], which are widely adopted in deep reinforcement learning. Specifically, we use the *actor-critic method* [89]. This method requires the agent to train two neural networks at the same time: *Actor neural network* and *Critic neural network*. The architecture is shown in Figure 5.3. Actor neural network is trained to be an estimate of the optimal policy. Critic neural network is trained to be an estimate of the optimal value function. As actor-critic method is one of the policy gradient methods, it is to learn the policy by performing gradient descent on the parameters. The basic idea of policy gradient methods is to estimate the gradient of the expected total rewards, which is derived from the trajectories of executions that are obtained by following a policy. As mentioned earlier, the objective of reinforcement learning is to maximize the expected cumulative reward. The gradient of this objective can be computed as [110]:

$$\nabla_\theta \mathbb{E}_{\pi_\theta}[R] = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) A^{\pi_\theta}(s,a)] \tag{5.4}$$

$A^{\pi_\theta}(s,a)$ is the advantage function. It represents the difference between the expected total reward when we choose to pick action a in state s and the expected reward for actions taken according to the policy π_θ. In our work, the agent samples a trajectory of scheduling decisions and uses the advantage $A(s_t, a_t)$ which is computed in the executions as an unbiased estimate

of $A^{\pi_\theta}(s_t, a_t)$. Thus, the update of the parameters θ of the policy neural network (actor neural network) can be represented as follows, where α is the learning rate.

$$\theta \leftarrow \theta + \alpha \sum_t \nabla_\theta \log \pi_\theta(s_t, a_t) A(s_t, a_t) \tag{5.5}$$

The idea behind this equation can be intuitively explained as follows. The $\nabla_\theta \log \pi_\theta(s_t, a_t)$ indicates the direction of the update for the parameters of the policy neural network. Based on this direction, it can increase $\pi_\theta(s_t, a_t)$, which is the probability of taking action a_t at state s_t. The advantage function $A(s_t, a_t)$ indicates the size of step of the update. Equation 5.5 takes a step of $A(s_t, a_t)$ towards the direction of $\nabla_\theta \log \pi_\theta(s_t, a_t)$. Consequently, it reinforces the actions drawn from policy π_θ, which leads to better returns.

In order to compute the advantage $A(s_t, a_t)$ during the executions, an estimate of the value function $V^{\pi_\theta}(s)$ is needed. The $V^{\pi_\theta}(s)$ is the expected total return when starting a process at state s and taking actions by following the policy π_θ. Hence, we need another neural network, critic neural network. The goal of the critic neural network is to learn an estimate of $V^{\pi_\theta}(s)$ in the meantime during the executions. Let θ_v be the parameters of the critic neural network and $V^{\theta_v}(s)$, an estimate of $V^{\pi_\theta}(s)$, be the output of the critic neural network. Hence, the advantage $A(s_t, a_t)$ can be estimated as $r_t + \gamma V^{\theta_v}(s_{t+1}) - V^{\theta_v}(s_t)$, which can be used for the update of the actor neural network. It is important to note that the function of the critic neural network is to help train the actor neural network. After training, only the actor neural network is required to make scheduling decisions.

5.4 GoTask Approach

In this section, we first present the basic design of our approach GoTask. Next, we introduce how to encode the state of the DAG scheduling problem and how to define action space and reward in deep reinforcement learning. Finally, we describe the training algorithm we used to train the neural networks in GoTask.

5.4.1 Design

The basic design of GoTask is the reinforcement learning agent continuously observes the state of the system at discrete timesteps, which includes the state of machines in the cluster and the state of DAG jobs. According to the observation, the agent performs encoding based on the current state to get a representation, and feeds the neural network the representation to make a scheduling decision. The decision can be a schedule action which means the agent is going to schedule some pending tasks (whose preceding tasks have been completed) to run, or a

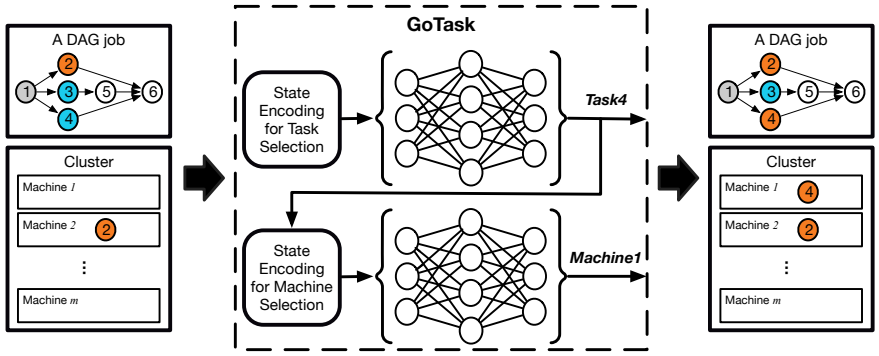


Figure 5.4: An example of GoTask process

void action which means no task is scheduled to run at this timestep. However, if we adopt the common deep reinforcement learning with a single neural network to learn scheduling policies, the size of the action space can be as large as $(M+1)^N$. It is because the agent may schedule any subset of the pending tasks to any subset of the machines which have available resources at a timestep. This large action space size makes the deep reinforcement learning almost impossible to learn good scheduling polices [129]. To address this problem, we propose a two-stage deep reinforcement learning approach, where the first stage leverages a deep reinforcement learning agent to learn policies for selecting a task from the pending tasks, and the second stage leverages another agent to learn policies for selecting a machine for running the selected task.

Figure 5.4 shows an example of GoTask process. Regarding the DAG job, task1 is completed; task2 is running on machine2; task3 and task4 are pending tasks which are ready to run. At a scheduling event, GoTask first observes the system and encodes the state to feed the neural network for task selection. According to the output of the neural network for task selection in Figure 5.4, it selects task4 from the pending tasks. Next, GoTask encodes the state with the profile of task4 for machine selection. It then selects machine1 according to another neural network. With these two selections, the agent schedules task4 to machine1 at this timestep. Note that the neural networks in Figure 5.4 are both actor neural networks that are leveraged to learn scheduling policies. This process continues until all tasks are completed. GoTask then trains the neural networks with the rewards received from the executions to empirically improve its policies. Therefore, the challenges of applying deep reinforcement learning are how to encode the state for task selection and machine selection, how to define practical action spaces, and how to define rewards to distinguish the quality of different scheduling actions.

5.4.2 Task Selection with Deep Reinforcement Learning

This subsection presents our approach for learning task selection with deep reinforcement learning, which includes the definition of state space, action space and reward.

State Space

To feed the neural network for task selection, we define the input state as $ST = (X, Y)$ which consists of two parts: vector X represents the state of all clustered machines, and vector Y represents the state of all pending tasks. To be more specific, we explain them as follows.

- $X = (X_1, X_2, ..., X_M)$ encodes the status of all the M machines in the cluster, where the status of each machine m_i is denoted by the vector $X_i = (v_i^1, v_i^2, ..., v_i^R, n_i)$. v_i^j is the amount of resource r_j available on the machine m_i, and n_i is number of tasks running on machine m_i at the current moment. These two information indicate the resource utilization and load situation of the machines in the cluster.

- $Y = (Y_1, Y_2, ..., Y_{N'})$ encodes the profile of all pending tasks, and N' is the number of pending tasks at the current moment. The profile of pending task t_i is denoted by $Y_i = (d_i^1, d_i^2, ..., d_i^R, s_i, ns_i, nc_i, lc_i)$. d_i^j is the amount of resource r_j that pending task t_i demands; s_i is the duration of pending task t_i; ns_i is the number of the total succeeding tasks of pending task t_i; nc_i is the number of the tasks in the longest path of pending task t_i; lc_i is the duration of the critical path of pending task t_i. We detail them as follows.

  In order to learn better scheduling policies for DAG jobs, the information of inter-task dependencies is essential for the scheduling decision making since a task may be blocked for a long time because of the precedence constraints. However, if we directly treat the dependency matrix $\mathbb{P} = [p_{ij}]_{N \times N}$ as a part of the state representation and feed the neural network the matrix, the neural network is still not able to well learn the knowledge of the inter-task dependencies. As the dependency matrix only shows the direct dependencies of a DAG job, the indirect dependencies are hard to be captured by the neural network. To tackle this problem, we encode the inter-task dependencies by representing the longest path and the critical path of each task in the state. The longest path of task t_i is the path with the most number of tasks from task t_i to the end of the DAG job. The critical path of task t_i is the path with the longest duration from task t_i to the end of the DAG job. Figure 5.5 shows an example of a DAG job; the number in the circle represents the duration of the task.

For task1, the number of the total succeeding tasks is 5, as the other 5 tasks are all the succeeding tasks of task1. The number of the tasks in the longest path is 4, as the longest path of task1 is $task1 \rightarrow task3 \rightarrow task5 \rightarrow task6$. The duration of the critical path is 10, as the path $task1 \rightarrow task2 \rightarrow task6$ has the longest duration for task1. Accordingly, if $task1$ is pending, it is $ns_1 = 5$, $nc_1 = 4$, and $lc_1 = 10$. We then leverage these information to represent the inter-task dependencies of a DAG job. Although the inter-task dependencies are only partially encoded in the state, it does have significant impact on the policy learning which is demonstrated in Section 5.5.3. In our implementation, we perform depth first search to get these information when a DAG job request arrives, and the time complexity is $O(N^2)$.

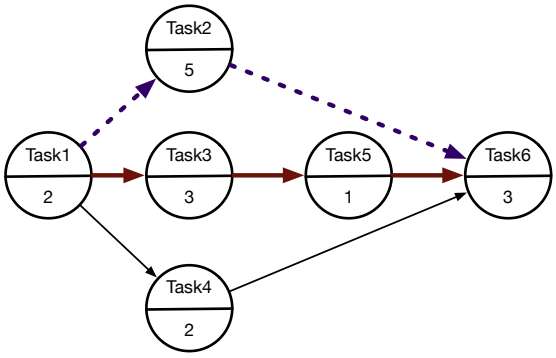


Figure 5.5: An example of a DAG job: red bold path is the longest path of task1; blue dashed path is the critical path of task1.

Action Space

At each scheduling event of GoTask, as many tasks can be ready at the same time, the agent may schedule any subset of the pending tasks to run. It makes the size of the action space as large as $2^{N'_{max}}$ (N'_{max} is the maximum number of pending tasks), while the deep reinforcement learning is still impossible to learn a good scheduling policy [125]. To address this problem, we allow the agent to execute more than one action at one timestep, so that one action only schedules one pending task. Hence, we can define the action space of task selection as a set: $\{\emptyset, 1, 2, ..., N'_{max}\}$, and the size of action space is reduced to $(N'_{max} + 1)$. $action = \emptyset$ means a void action that no pending task is selected to run; $action = i$ means selecting ith task in the vector Y to run. At one timestep, the agent continuously schedules pending tasks until it chooses the

void action or an invalid action (e.g., it selects a machine that does not have enough resources to run the selected task). With a valid action that a pending task is scheduled to a machine, the agent would observe the system and perform the scheduling event again immediately.

Reward

In reinforcement learning, the reward is a signal that the environment tells the agent how well it is doing on the task. Typically, the reward is a scalar value. Since the objective of reinforcement learning is to maximize the expected cumulative reward, the definition of the reward must reflect the goal of the problem to be solved. In GoTask, we set the reward to $-\tau$ for all the actions that are taken during the job execution. τ is the time difference between the current action and the last action. If two actions are taken at the same timestep, the reward τ is 0 for the latter one. Therefore, if we set the discount factor γ as 1, the cumulative reward during the job execution is equal to the negative job completion time $-$JCT. Our goal is then consistent with the goal of the reinforcement learning. The smaller the JCT is, the greater the cumulative reward is, and vice versa.

5.4.3 Machine Selection with Deep Reinforcement Learning

This subsection presents our approach for learning machine selection for running the selected task with deep reinforcement learning.

State Space

To feed the neural network for machine selection, we define the input state as $SM = (X, Y_i, Z)$ which consists of three parts: X is the same as the vector in ST; Y_i is the profile vector of the pending task t_i selected by the above neural network; Z denotes the fitness scores of several packing heuristics. Efficient task packing with multi-resource constraints is challenging, which typically requires complex heuristics [66]. To facilitate policy learning, we introduce several heuristics to help the agent better select the machine. More specifically, we explain Z as follows.

- $Z = (Z_1, Z_2, ..., Z_K)$ contains the fitness scores of each heuristic, and K is the number of the heuristics. To represent the fitness when applying different heuristics, we leverage a score based approach to encode the state. For the ith heuristic, the score vector is defined as $Z_i = (z_i^1, z_i^2, ..., z_i^M)$, where z_i^j denotes the score when scheduling the task on machine m_j using the ith heuristic. In our current implementation, we provide three heuristics, which are described as follows.

1) **Most loaded**: Schedule the task to the most loaded machine. To give an example, we consider only two type of resources: CPU and memory. Thus, the score of applying most loaded heuristic of scheduling the task to machine m_j is calculated as $\frac{1}{2}(\frac{d_i^{cpu}}{v_j^{cpu}} + \frac{d_i^{mem}}{v_j^{mem}})$ (d_i^{cpu} is the amount of CPU that task t_i demands, v_j^{cpu} is the amount of available CPU on machine m_j). Accordingly, the higher the score is, the more loaded the machine is.

2) **Least loaded**: Schedule the task to the least loaded machine. The score is calculated as $\frac{1}{2}(\frac{v_j^{cpu}-d_i^{cpu}}{v_j^{cpu}} + \frac{v_j^{mem}-d_i^{mem}}{v_j^{mem}})$.

3) **Balanced loaded**: Schedule the task to the machine with balanced resource usage rate. The score is calculated as $1 - (\frac{d_i^{cpu}}{v_j^{cpu}} - \frac{d_i^{mem}}{v_j^{mem}})^2$.

Note that the score is calculated only if machine m_j has sufficient resources to run the pending task, otherwise the score is 0. This score based approach can be easily extended to handle more resource constraints.

Action Space

We define the action space for machine selection as a set: $\{\emptyset, 1, 2, ..., M\}$. $action = \emptyset$ means a void action that no machine is selected to run the selected pending task; $action = i$ means machine m_i is selected to run the selected pending task. As mentioned in the previous section, the agent continuously schedules pending tasks to machines at one timestep until it chooses the void action or an invalid action.

Reward

The reward for the machine selection is the same as the task selection. We set the reward to $-\tau$ for all the actions that are taken during the job execution. τ is the time difference between the current action and the last action.

5.4.4 Training Algorithm

After defining state space, action space and reward, we could build the deep reinforcement learning agents to learn scheduling policies. According to the actor-critic method, we train two neural networks: actor neural network and critic neural network, for task selection and machine selection respectively. As the critic neural network is an estimate of the optimal value function, the output of the critic network is an estimate of the maximum expected return from a state. While the actor neural network is an estimate of the optimal policy, the output of the actor network is a vector whose element represents the probability of a corresponding action. Algorithm 8 is used to train the two neural networks for

Algorithm 8: Actor-critical method in GoTask for task selection

```
   /* Assume the actor neural network with parameters θ, and the
      critical neural network with parameters θ_v                  */
 1 while A DAG job arrives do
 2     dθ ← 0;
 3     dθ_v ← 0;
 4     t ← 0;
 5     repeat
 6         Observe state s_t;
 7         Encode the state for task selection;
 8         Perform action a_t according to policy π_θ;
 9         Receive reward r_t ;
10         t ← t + 1;
11     until The job is completed;
12     R ← 0;
13     for i ← t − 1; i ≥ 0; i ← i − 1 do
14         R ← r_i + R;
15         dθ ← dθ + ∇_θ log π_θ(s_i, a_i)(R − V^{θ_v}(s_i));
16         dθ_v ← dθ_v + ∂(R − V^{θ_v}(s_i))^2/∂θ_v;
17     end
18     Perform update of θ using dθ;
19     Perform update of θ_v using dθ_v;
20 end
```

task selection in GoTask. The Algorithm for machine selection is similar. The basic process can be described as follows. When a DAG job request arrives, the agent starts scheduling the job. According to its current policy (output of the actor neural network), the agent continuously selects the pending tasks of the DAG job to run. After the job is completed, the agent first uses the entire trajectory including all the states, the actions and the rewards, to calculate the accumulated gradients. Then, the agent updates the parameters of the two neural networks with the accumulated gradients. This process would be iteratively repeated many times to empirically learn a better scheduling policy.

5.5 Evaluation

In this section, we evaluate GoTask through simulations. In the evaluation, we experimentally compare GoTask with commonly adopted scheduling heuristics, and try to understand the convergence and improvement about GoTask

approach.

5.5.1 Implementation

We implement GoTask as a prototype DAG scheduler using Python. The deep neural networks implemented in GoTask are based on Tensorflow [21] software library. If we try to apply GoTask to a computer cluster and train the neural networks online from scratch, the policies derived from the agent are poor at the beginning. It is because deep reinforcement learning typically needs a lot of trials and errors to learn a good policy [126]. Thus, offline training is essential for the policy learning before scheduling DAG jobs online. In order to train the neural networks offline, we implement a simulator for simulation of task execution on multi-resource clusters, which is based on DeepRM [102]. DeepRM is the first example solution that applies deep reinforcement learning to cluster scheduling problem. We extend the simulator of DeepRM to support multiple heterogeneous machines in the cluster, as it originally model the cluster as a resource pool. During the runtime, the simulator schedules tasks according to the decision made by the policy neural networks. GoTask then can rapidly learn the scheduling policies during the interaction with the simulator. In the evaluation, all experiments are conducted with the simulator.

5.5.2 Experimental Methodology

Cluster. We mimic two different clusters with 10 machines in our experiments. For the first cluster, we use 10 homogeneous machines with 8 CPU cores and 16 GB RAM. Considering the heterogeneity, we use 4 machines with 4 CPU cores and 8 GB RAM, 3 machines with 8 CPU cores and 16 GB RAM, and 3 machines with 16 CPU cores and 32 GB RAM for the second cluster.

Workloads. We yield DAG job requests according to the work [46]. Specifically, we use the Layer-by-Layer method to generate DAGs, as data processing jobs can be commonly divided into many stages (e.g., MapReduce jobs include map tasks and reduce tasks). The parameters of the Layer-by-Layer method used in our experiments are defined as: the number of vertices (i.e., the number of tasks in a job) is picked randomly from 100 to 200; the number of layers is picked randomly from 4 to 8; the probability of edge creating between two vertices is 0.1. Considering diversity of DAG jobs, we also randomly generate the resource demands and duration for each task in the DAG jobs. We define the length of one timestep as $1t$, and the duration of each task is uniformly picked at random from $[1t, 24t]$. The number of CPU cores demanded by each task is uniformly picked at random from $[1, 6]$. The number of GB RAM demanded by each task is uniformly picked at random from $[1, 12]$. According to these parameters, we generate 5,000 DAG jobs in total. We use 4,000 DAG jobs of them for training, and other 1,000 DAG jobs for testing.

Neural Network Configuration. As neural networks require fixed-size input, we show up to 16 pending tasks in the state representation. If there are more than 16 pending tasks at a moment, we prioritize the tasks according to the duration of tasks critical path. All neural networks in GoTask have 2 fully connected hidden layers with 256 neurons of ReLU6 nonlinearity. We update the parameters of the neural networks using the *rmsprop* [73] algorithm. The learning rate of the actor and critical neural network are configured to be 0.0001 and 0.001 respectively. We adopt a state-of-the-art asynchronous method, A3C [110], to speed up training. We train GoTask on a machine with a Intel Xeon E5-2630 2.4GHz CPU and a Nvidia GTX TitanX GPU. The neural networks are trained for 10,000 iterations in our experiments.

Baselines. We compare GoTask with following heuristics:
For task selection:

- Shortest Task First (STF) schedules pending tasks in increasing order of the task duration.
- Critical Path First (CPF) schedules pending tasks in increasing order of the duration of critical path [90].

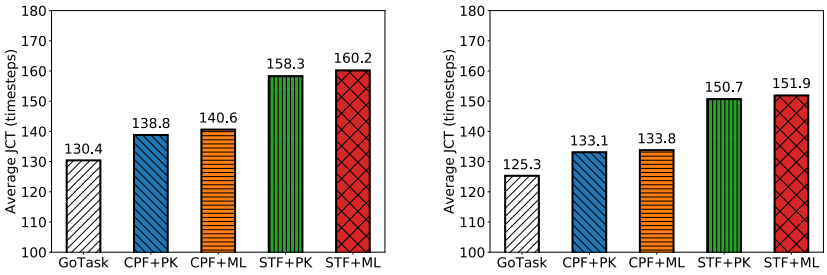
For machine selection:

- Most Loaded (ML) schedules the pending task to the most loaded machine in the cluster.
- Multi-Resource Packer (PK) schedules pending tasks in increasing order of alignment between resource demands and resource availability [66].

5.5.3 Experimental Results

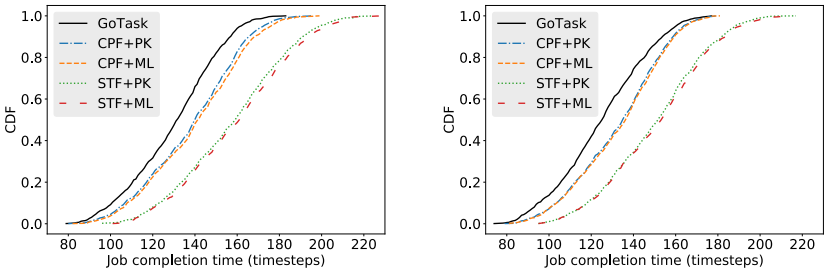
Comparison with Baselines

First, we evaluate the scheduling performance of different approaches. In the experiment, we first train GoTask with the 4,000 DAG jobs. Then, we use other 1,000 DAG jobs to test the trained model and the baselines. Figure 5.6 shows the average job completion time (JCT), and Figure 5.7 shows the Cumulative Distribution Function (CDF) of job completion times for different approaches. We observe that GoTask improves the JCT by 6% to 18% compared to all the baselines for both the homogeneous and the heterogeneous cluster. Shortest Task First (STF) heuristic does not perform well as it only considers the duration of a individual task. Critical Path First (CPF) heuristic performs much better than STF, but it only focuses on the maximum duration of a DAG job, without considering the resource demands of tasks. Multi-Resource Packer (PK) performs slightly better than Most Loaded (ML) heuristic due to more effective packing heuristic. However, they both try to improve the resource efficiency,



(a) Homogeneous Cluster (b) Heterogeneous Cluster

Figure 5.6: Comparing average job completion time with different baselines



(a) Homogeneous Cluster (b) Heterogeneous Cluster

Figure 5.7: CDF of job completion times for different approaches

but ignore the task characteristics of DAG jobs (i.e., inter-task dependencies). For GoTask, one interesting observation is that it may withhold a task when there are machines which have enough resources to run the task. This can make some room for the tasks that will be pending soon. The machines available now may be the better choices for these tasks. We observe this happening few times during the execution. But, the main reason of the improvement is that GoTask can run more independent tasks in parallel to improve the job completion time. Based on the policies learned from experience, GoTask can select a suitable pending task to run on a suitable machine at a time, which improves the overall throughput of the cluster. All theses achievements is because GoTask takes cluster resource utilization, task duration, task resource demands and inter-task dependencies into consideration when making scheduling decisions, and those information are well encoded in the state representation. To conclude, with the deep reinforcement learning, GoTask finds a better balance to schedule

DAG jobs among different constraints and requirements, which leads to a good scheduling policy.

Improvement from State Encoding

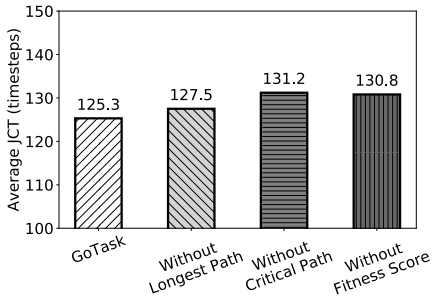


Figure 5.8: Comparing different agents on the heterogeneous cluster

Next, we investigate how much gains GoTask can get from the state encoding, as we present a sophisticated way to encode the inter-task dependencies for task selection, and a fitness score based approach to encode several packing heuristics for machine selection. In this experiment, we build other three agents to train the neural networks without *longest path*, *critical path*, or *fitness score* in the state representation, which are for evaluating how much impact it has on the scheduling performance. Figure 5.8 shows the average job completion time on the heterogeneous cluster. According to the results, the scheduling performance can benefit from all the encoded information. The gain from critical path is larger than the longest path, as it is more likely to cause longer job completion time if a task with longer critical path has been blocked for a longer period of time during the execution. The improvement of the fitness score is also non-trivial to the scheduling performance. It demonstrates that learning a good packing policy on multi-resource clusters without knowledge is quite challenging for the deep reinforcement learning agent. Giving several packing heuristics to the agent can result in a much better scheduling policy.

Learning Curve

Additionally, in order to understand the convergence, we show the learning curve along the training in Figure 5.9. In our experiments, we train the neural networks for 10,000 iterations. At each iteration, we sample 128 DAG jobs from the 4,000 DAG jobs to train GoTask. During the runtime, GoTask schedules the DAG jobs according to its policy neural networks and receives the corresponding

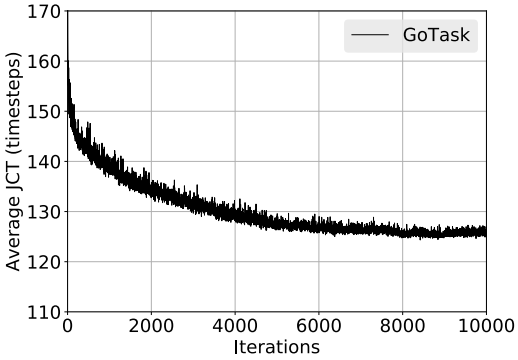


Figure 5.9: Learning curve on the heterogeneous cluster

rewards. After the DAG jobs are completed, GoTask uses the trajectories of executions to calculate the gradients, and then leverages the gradients to update the neural networks. In this experiment, we use the trained model after each iteration to schedule the 1,000 DAG jobs which are for testing. Figure 5.9 shows the average job completion time of each iteration on the heterogeneous cluster. We observe that the average JCT gradually decreases with the number of training iterations, and GoTask outperforms the baselines after around 2,000 iterations. The learning curve tends to be stable after 8,000 iterations where the agent may have learned the best possible scheduling policy. In our experiments, each iteration takes about 3.6 seconds. Hence, it takes around 10 hours to finish the training.

Configuration of Neural Network

Finally, we evaluate the performance of different configurations of the neural networks in GoTask. We conduct this experiment on the heterogeneous cluster. First, in order to understand the impact of different number of neurons in the hidden layers, we fix the number of hidden layers to 2 and range the number of neurons from 64 to 512. Each hidden layer has the same number of neurons, and the configuration of neural networks in task selection stage and machine selection stage is the same. For each configuration, we train the neural networks for 10,000 iterations. Figure 5.10(a) shows that GoTask achieves the best performance when there are 256 neurons in the hidden layers. If there are fewer neurons, GoTask does not have enough neural network parameters to approximate the scheduling policy. If there are too many neurons, GoTask may capture some unnecessary properties, i.e., overfitting, which leads to degraded performance. Second, we investigate the impact of different number of hidden

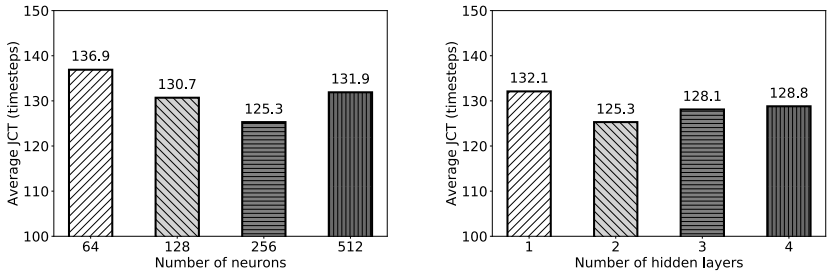


Figure 5.10: Performance of different neural network configurations on the heterogeneous cluster

layers in GoTask. We fix the number of neurons to 256 and range the number of hidden layers from 1 to 4. Figure 5.10(b) shows that GoTask achieves the best performance when there are 2 hidden layers. If there are fewer hidden layers, GoTask does not have enough neural network parameters to learn a good scheduling policy. If there are more hidden layers, GoTask may degrade the performance because of overfitting.

5.6 Discussion

In our simulation experiments, we demonstrate that GoTask can effectively learn scheduling policies for DAG jobs from experience. The experimental results show that GoTask noticeably outperforms the baselines on two different clusters. Next, we will apply GoTask to the production cluster and evaluate the performance in realistic scenarios. Specifically, we plan to implement a pluggable scheduler on Kubernetes [9] for online scheduling and learning. Based on the simulations results, we believe that our approach is practical even in realistic scenarios. However, during the design and implementation of GoTask, we can see there are some limitations that still need to be addressed in the future. First, as neural networks require fixed-size input, we fix the state representation to show up to 16 pending tasks. This causes that GoTask can only observe partial pending tasks at one time, and may fail to select the best suitable pending task to run. Thus, we intend to investigate more compact state representation approaches, such as graph convolutional networks [86], to enhance the scalability. Second, as GoTask is implemented to observe the system at each timestep, the action sequence during a job execution can be very long. It is because the sequence may contain many void actions. Long

action sequences need more exploration during the training and can make the deep reinforcement learning algorithms extremely slow [129]. Therefore, we plan to investigate event-driven approaches to build GoTask, which only performs a scheduling event when a task completes or a task becomes ready. Third, the training speed is another issue. When the cluster or the workloads change, we need to retrain the neural networks to adapt to the new scenario, which takes another long time to converge. Thus, we will investigate machine learning techniques, such as transfer learning [135], to speed up the retraining process.

5.7 Related work

The problem investigated in this chapter - Learning DAG Scheduling on Multi-Resource Clusters - is related to a variety of research topics as follows.

DAG Scheduling Many DAG schedulers have been proposed for different purposes [121, 37, 148]. Graphene [68] DAG scheduler schedules jobs that have a complex dependency structure and heterogeneous resource demands. It first schedules troublesome tasks and then schedules the remaining tasks without violating dependencies to improve job completion time. Zhu et al. [162] proposed a multi-objective solution based on evolutionary algorithms to handle workflow scheduling problem in cloud, which optimizes both makespan and cost. Yao et al. [150] presented a fault-tolerant workflow scheduling approach to meet the soft deadline requirements. Different from them, we apply reinforcement learning to learn DAG scheduling policies directly from the experience.

Deep Reinforcement Learning Recently, deep reinforcement learning has made great success in many areas [50, 92]. Mnih et al. [108] presented the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. And they have successfully applied their approach to the computer video games. Silver et al. [125] proposed AlphaGo to master the game of Go with deep neural networks and tree search. They introduced a new search algorithm that combines Monte Carlo simulation with value and policy networks, and successfully applied in the game of Go. Inspired by these works, we investigate how to apply the deep reinforcement learning to the DAG scheduling problem.

Scheduling with Reinforcement Learning Li et al. [96] developed a model-free approach for distributed stream data processing using deep reinforcement learning. They also demonstrated that the actor-critic method can produce better scheduling solutions than Deep Q Network based method in distributed stream data processing. DeepRM [102] is the first example solution that applies deep reinforcement learning to cluster scheduling problem. It translates the problem of packing tasks with multiple resource demands into a learning problem. DeepRM is designed to handle job scheduling in an online

manner, and represents the state of the system as images which include the resource profiles of jobs and the current allocation of cluster resources. However, they cannot handle DAG jobs due to the limitation of their job model, and they simply model the cluster as a resource pool. To the DAG scheduling, Wu et al. [147] proposed an adaptive DAG tasks scheduling algorithm using deep reinforcement learning. Orhean et al. [111] presented a scheduling approach using reinforcement learning for heterogeneous distributed systems. However, they both ignore the multi-resource requirements of the DAG jobs, and mainly focus on computational power and capacity to make scheduling decisions. Unlike previous researches, we investigate an approach that can effectively learn DAG scheduling with multi-resource constraints on heterogeneous clusters.

5.8 Conclusion

In this chapter, we present GoTask, an approach that can learn to well schedule DAG jobs on heterogeneous clusters. GoTask directly learns scheduling policies from experience through deep reinforcement learning. In order to handle the complexity and scale of the DAG scheduling problem, we propose a two-stage approach in GoTask, where the first stage leverages a deep reinforcement learning agent to learn policies for selecting a pending task of a DAG job, and the second stage leverages another agent to learn policies for selecting a machine to run the selected task. We implement a GoTask prototype and a simulator for simulation of task execution on multi-resource clusters. In the evaluation, the experimental results showed that GoTask can noticeably benefit from the proposed state encoding approaches, and outperforms commonly adopted scheduling heuristics, which improves job completion time by 6% to 18%.

6

Conclusion and Future Work

Cloud computing is becoming increasingly popular and widely used in almost every aspect of today's business. By encapsulating the runtime context of a software system, container technologies can significantly simplify the deployment and maintenance of cloud applications. Meanwhile, the performance of cloud applications is getting more and more attention as many applications can only deliver the expected business value when their quality of services and user experiences are guaranteed. To achieve the desired performance, efficient cloud resource scheduling is crucial. Although many research efforts have been devoted to cloud resource scheduling problems, there are still many challenges in this area. **With respect to application type and scale**, when the scale of applications grows, particularly when the number of distributed data sources or sensors increases, a large number of concurrent application requests will have to be supported by the cloud system simultaneously. As distributed applications often have diverse resource requirements, it imposes stringent challenges to support such applications on a cloud infrastructure. In terms of applications types, service-based applications and batch jobs coexist in cloud computing. To maintain the performance required by the applications, the application-specific properties (e.g., inter-task dependencies) have to be handled together with the resources constraints by the cloud application schedulers. **With respect to quality-critical requirements**, the application performance can be dramatically influenced by the resources provided by the underlying infrastructure (e.g., CPU capacity, memory size, and network bandwidth). The resource selection on cloud infrastructure is thus crucial for satisfying the specific runtime requirements of applications (e.g., the timely response of service-based applications and the short processing time of batch jobs). **With respect to infrastructure heterogeneity and scale**, a distributed cloud infrastructure, particularly virtual infrastructure, is often across multiple data centers or providers, which can consist of virtual machines with highly diverse configurations. The scale of modern cloud computer cluster can contain several thousand, or even more, machines for a large number of applications to run

simultaneously. The business value of such large scale infrastructure heavily relies on the quality of scheduling for different application requirements.

In this thesis, we investigated resource scheduling problem for quality-critical applications on cloud infrastructure. We started from the simple scenario where applications are deployed on a dedicated cloud infrastructure. We have proposed a deadline-aware deployment system for **time critical applications** to meet **deployment deadlines** in Chapter 2. And then, we focused on a common scenario where multiple applications share the **heterogeneous infrastructure**. We have proposed an enhanced container scheduler for **concurrent container requests** to meet **multi-resource requirements** in Chapter 3. Finally, we investigated application-centric scheduling approaches for two types of cloud applications. For **service-based applications**, we have proposed a service placement solution to optimize **inter-machine traffic** in Chapter 4. For **DAG jobs**, we have tackled the complexity of DAG scheduling with **multi-resource requirements** by using deep reinforcement learning based approach in Chapter 5.

In this chapter, we first summarize the contributions of this thesis in Section 6.1. Then, we propose several directions for future research in Section 6.2.

6.1 Conclusion

We list the main contributions of the thesis as follows:

- **RQ1 How can we effectively deploy distributed applications with critical time constraints in clouds?**

  To answer this question, we have analyzed the time cost in the procedure of cloud application deployment and observed that the difference of deployment is mainly caused by the transmission time of container images over network. To optimize the usage of the network bandwidth, we have proposed a Deadline-aware Deployment System (DDS) for deploying time-critical applications in clouds. We employed Earliest Deadline First (EDF) method to prioritize the requests with the deadline constraints in DDS. We designed a bandwidth-aware algorithm to achieve parallel transmission without causing network bandwidth contention. In the experiments, we have demonstrated that DDS leverages network bandwidth sufficiently, and significantly reduces the number of missed deadlines during deployment.

- **RQ2 How can we efficiently handle concurrent container requests with multi-resource constraints on heterogeneous clusters?**

To answer this question, we first analyzed the characteristics of existing cluster schedulers, and then proposed a graph-based scheduler called Enhanced Container Scheduler (ECSched). We formulated the container scheduling problem as a minimum cost flow problem (MCFP) and represented the container requirements using a specific graph data structure (i.e., flow network). In the flow network, we proposed two strategies to encode the multi-resource demands of requested containers, and two adjustments of the flow network to handle container affinity and machine affinity requirements. We implemented ECSched with a container manager and an appropriate variant of MCFP algorithms. In the experiments, we have shown that ECSched outperforms state-of-the-art container schedulers, which can lower the average container completion time by up to 1.3× and noticeably improve the resource utilization. For the scheduling overhead, the large-scale simulations showed that ECSched introduces a small overhead, but it is acceptable in practice.

- **RQ3 How can we optimize the placement of service-based applications in clouds?**

  To answer this question, we have proposed a new approach to optimize the placement of service-based applications in clouds. The approach first partitions the application into several parts while keeping overall traffic between different parts to a minimum, and then packs the different parts into machines based on their resource requirements. Based on a well designed randomized contraction algorithm, we proposed two algorithms: Binary Partition and K Partition, to find a high-quality partition for service-based applications. And we proposed a packing algorithm based on a packing heuristic with traffic awareness. We combined the algorithms of partitioning and packing with a resource demand threshold to optimize placement solutions. We conducted extensive experiments to demonstrate the proposed algorithms. The results showed that our approach outperforms existing container cluster schedulers and representative heuristics, leading to much less overall inter-machine traffic.

- **RQ4 How can we learn scheduling policies of DAG jobs with deep reinforcement learning on multi-resource clusters?**

  To answer this question, we have proposed a deep reinforcement learning based approach called GoTask. In order to tackle the complexity of the DAG scheduling problem, we have proposed a two-stage learning approach in GoTask, for learning policies for selecting a pending task of a DAG job, and for selecting a machine to run the selected task respectively. In the task selection stage, we adopted an approach based on longest

path and critical path to encode inter-task dependencies. In the machine selection stage, we represented fitness scores of several packing heuristics in the state to facilitate the learning of scheduling policy. At runtime, GoTask continuously observes the state of machines in the cluster and the state of the DAG job. According to the observation, GoTask performs encoding based on the current state to get a representation, and feeds the neural network the representation to make a scheduling decision. We evaluated the performance of GoTask via simulations. The experimental results showed that GoTask outperforms commonly adopted scheduling heuristics, which improves job completion time by 6% to 18%.

The work presented in the thesis is conducted in the context of a number of EU projects. The proposed algorithms have been implemented in the software system Dynamic Real-time Infrastructure Planner (DRIP) in the EU SWITCH project[1], and have been further exploited in the followup project EU ARITCONF project[2]. The DRIP has been used as part of the infrastructure optimization solution in the EU ENVRIPLUS project[3] and EU ENVRI-FAIR project[4] for supporting big data management services in environmental and earth sciences. It has also been used in a number of use cases, e.g., life event broadcast [160] and disaster early warning systems [161].

6.2 Future Work

There are a few future directions that need exploration:

- *Handling dynamics of cloud resources.* The algorithms and approaches proposed in this thesis are based on the fixed and static resource model. However, the dynamics of cloud resources (e.g., resources may join or exit at any time) would have much impact on the application performance. Thus, it is worth to investigate how to model the resource dynamics and how to incorporate the resource dynamics into resource scheduling. This will provide a more resilient and robust scheduling mechanism for cloud applications.

- *Understanding runtime properties of cloud applications.* In this thesis, the basic application requirements and constraints are provided by users. However, there are some essential runtime properties which are unknown in advance, such as the actual resource consumption and the interference between difference applications. These runtime properties are crucial for

[1] EU H2020 SWITCH: https://www.switchproject.eu
[2] EU H2020 ARTICONF: https://www.articonf.eu
[3] EU H2020 ENVRIPLUS: https://www.envriplus.eu
[4] EU H2020 ENVRI-FAIR: https://www.envri.eu/envri-fair

achieving a stable application performance, which also need to be handled very well when running applications in clouds. To tackle this problem, we plan to use machine learning based technologies to learn and understand the runtime properties of different applications.

- *Extending to fog and edge computing.* With the fast growth of Internet of Things (IoT) and rapid development of 5G technology, we can expect that hundreds of billions of devices will be connected to the internet to bring promising new applications in the future. To handle such massive numbers of devices and corresponding data, fog computing and edge computing are emerging as the main paradigm to mitigate the gap between IoT devices and cloud computing. Integrating cloud, fog and edge computing and scheduling diverse resources together could have great benefits for handling IoT applications. Thus, we plan to extend our work to incorporate fog and edge resources to cope with more complex and dynamic environments.

Bibliography

[1] Alibaba cluster trace. https://github.com/alibaba/clusterdata/.

[2] Amazon emr. https://aws.amazon.com/emr/.

[3] Microsoft azure. https://azure.microsoft.com/.

[4] Control groups (cgroups). https://www.kernel.org/doc/Documentation/cgroup-v1/.

[5] Amazon web services. https://aws.amazon.com/.

[6] Google cloud platform. https://cloud.google.com/.

[7] Google container registry. https://cloud.google.com/container-registry/, .

[8] Google cluster trace. https://github.com/google/cluster-data/, .

[9] Google kubernetes. https://kubernetes.io/.

[10] Kernel-based virtual machine. https://www.linux-kvm.org/.

[11] Mesosphere marathon. https://mesosphere.com/.

[12] Mesos containerizer. http://mesos.apache.org/.

[13] Microservices a definition of this new architectural term. https://martinfowler.com/articles/microservices/.

[14] Linux namespaces. https://en.wikipedia.org/wiki/Linux_namespaces/.

[15] Openstack cloud software. https://www.openstack.org/.

[16] Coreos rkt. https://coreos.com/rkt/.

[17] Docker swarm. https://docs.docker.com/engine/swarm/.

[18] Vmware esxi hypervisor. https://www.vmware.com/products/esxi-and-esx/.

[19] Docker Hub. URL https://hub.docker.com/.

[20] Apache hadoop. https://hadoop.apache.org/.

[21] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[22] S. Abrishami, M. Naghibzadeh, and D. H. Epema. Cost-driven scheduling of grid workflows using partial critical paths. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1400–1414, 2012.

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

[24] Y. Ajiro and A. Tanaka. Improving packing algorithms for server consolidation. In *Int. CMG Conference*, volume 253, 2007.

[25] M. Alicherry and T. Lakshman. Network aware resource allocation in distributed clouds. In *2012 Proceedings IEEE INFOCOM*, pages 963–971. IEEE, 2012.

[26] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pfabric: Minimal near-optimal datacenter transport. 43(4):435–446, 2013.

[27] N. Alshuqayran, N. Ali, and R. Evans. A systematic mapping study in microservice architecture. In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 44–51. IEEE, 2016.

[28] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder. Performance evaluation of microservices architectures using containers. In *2015 IEEE 14th International Symposium on Network Computing and Applications*, pages 27–34. IEEE, 2015.

[29] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[30] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*, 2017.

[31] U. Awada and A. Barker. Improving resource efficiency of container-instance clusters on clouds. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and*

Grid Computing (CCGRID), pages 929–934. IEEE, 2017.

[32] R. Bajaj and D. P. Agrawal. Improving scheduling of tasks in a heterogeneous environment. *IEEE Transactions on Parallel and Distributed Systems*, 15(2):107–118, 2004.

[33] A. Balalaie, A. Heydarnoori, and P. Jamshidi. Migrating to cloud-native architectures using microservices: an experience report. In *European Conference on Service-Oriented and Cloud Computing*, pages 201–215. Springer, 2015.

[34] I. Baldin, J. Chase, Y. Xin, A. Mandal, P. Ruth, C. Castillo, V. Orlikowski, C. Heermann, and J. Mills. Exogeni: A multi-domain infrastructure-as-a-service testbed. In *The GENI Book*, pages 279–315. Springer, 2016.

[35] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

[36] L. A. Barroso. Warehouse-scale computing: Entering the teenage decade. 2011.

[37] S. Baskiyar and R. Abdel-Kader. Energy aware dag scheduling on heterogeneous systems. *Cluster Computing*, 13(4):373–383, 2010.

[38] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 65–72. ACM, 2009.

[39] O. Biran, A. Corradi, M. Fanelli, L. Foschini, A. Nus, D. Raz, and E. Silvera. A stable network-aware vm placement for cloud systems. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 498–506. IEEE, 2012.

[40] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 285–300, 2014.

[41] D. Breitgand, A. Marashini, and J. Tordsson. Policy-driven service placement optimization in federated clouds. *IBM Research Division, Tech. Rep*, 9:11–15, 2011.

[42] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, omega, and kubernetes. *Communications of the ACM*, 59(5):50–57, 2016.

[43] E. Casalicchio and L. Silvestri. Mechanisms for SLA provisioning in cloud-based service providers. *Computer Networks*, 57(3):795–810, 2013.

[44] T. Cerny, M. J. Donahoo, and M. Trnka. Contextual understanding of microservice architecture: current and future directions. *ACM SIGAPP Applied Computing Review*, 17(4):29–45, 2018.

[45] L. Chen, K. Chen, W. Bai, and M. Alizadeh. Scheduling mix-flows in commodity datacenters with Karuna. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 174–187. ACM, 2016.

[46] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner. Random graph generation for scheduling simulations. In *Proceedings of the 3rd international ICST conference on simulation tools and techniques*, page 60. ICST (Institute for Computer Sciences, Social-Informatics and , 2010.

[47] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[48] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, pages 499–510, 2015.

[49] C. Delimitrou, D. Sanchez, and C. Kozyrakis. Tarcil: reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 97–110. ACM, 2015.

[50] L. Deng, D. Yu, et al. Deep learning: methods and applications. *Foundations and*

Trends® in Signal Processing, 7(3–4):197–387, 2014.

[51] E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. In *Soviet Math. Doklady*, volume 11, pages 1277–1280, 1970.

[52] J. Dong, X. Jin, H. Wang, Y. Li, P. Zhang, and S. Cheng. Energy-saving virtual machine placement in cloud data centers. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 618–624. IEEE, 2013.

[53] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: yesterday, today, and tomorrow. In *Present and ulterior software engineering*, pages 195–216. Springer, 2017.

[54] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)*, 19(2):248–264, 1972.

[55] M. H. Ferdaus, M. Murshed, R. N. Calheiros, and R. Buyya. Virtual machine consolidation in cloud data centers using aco metaheuristic. In *European Conference on Parallel Processing*, pages 306–317. Springer, 2014.

[56] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 99–112. ACM, 2012.

[57] M. Gabay and S. Zaourar. Vector bin packing with heterogeneous bins: application to the machine reassignment problem. *Annals of Operations Research*, 242(1):161–194, 2016.

[58] W. Gao, H. Jin, S. Wu, X. Shi, and J. Yuan. Effectively deploying services on virtualization infrastructure. *Frontiers of Computer Science*, 6(4):398–408, 2012.

[59] Y. Gao, H. Guan, Z. Qi, Y. Hou, and L. Liu. A multi-objective ant colony system algorithm for virtual machine placement in cloud computing. *Journal of Computer and System Sciences*, 79(8):1230–1242, 2013.

[60] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, volume 11, pages 24–24, 2011.

[61] I. Gog, M. Schwarzkopf, A. Gleave, R. N. Watson, and S. Hand. Firmament: Fast, centralized cluster scheduling at scale. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 99–115, 2016.

[62] A. V. Goldberg. An efficient implementation of a scaling minimum-cost flow algorithm. *J. Algorithms*, 22(1):1–29, 1997.

[63] A. V. Goldberg and M. Kharitonov. On implementing scaling push-relabel algorithms for the minimum-cost flow problem. In *Network Flows and Matching*, pages 157–198, 1991.

[64] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by successive approximation. *Mathematics of Operations Research*, 15(3):430–466, 1990.

[65] O. Goldschmidt and D. S. Hochbaum. Polynomial algorithm for the k-cut problem. In *[Proceedings 1988] 29th Annual Symposium on Foundations of Computer Science*, pages 444–451. IEEE, 1988.

[66] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review*, 44(4):455–466, 2015.

[67] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *OSDI*, pages 65–80, 2016.

[68] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. G: Packing and dependency-aware scheduling for data-parallel clusters. In *Proceedings of OSDI'16: 12th USENIX Symposium on Operating Systems Design and Implementation*, page 81, 2016.

[69] M. A. Hajji and H. Mezni. A composite particle swarm optimization approach for the composite saas placement in cloud environment. *Soft Computing*, 22(12):4025–4045, 2018.

[70] W. Hasselbring and G. Steinacker. Microservice architectures for scalability, agility

and reliability in e-commerce. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 243–246. IEEE, 2017.

[71] K. Hightower, B. Burns, and J. Beda. *Kubernetes: Up and Running: Dive Into the Future of Infrastructure*. " O'Reilly Media, Inc.", 2017.

[72] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. 11(2011):22–22, 2011.

[73] G. Hinton, N. Srivastava, and K. Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on*, page 14, 2012.

[74] X. Hou, Y. Lu, and S. Dey. Wireless vr/ar with edge/cloud computing. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–8. IEEE, 2017.

[75] Y. Hu, H. Li, and Y. Peng. NVLAN: A novel VLAN technology for scalable multi-tenant datacenter networks. In *Advanced Cloud and Big Data (CBD), 2014 Second International Conference on*, pages 190–195. IEEE, 2014.

[76] Y. Hu, J. Wang, H. Zhou, P. Martin, A. Taal, C. de Laat, and Z. Zhao. Deadline-aware deployment for time critical applications in clouds. In *European Conference on Parallel Processing*, pages 345–357. Springer, 2017.

[77] Y. Hu, H. Zhou, C. de Laat, and Z. Zhao. Ecsched: Efficient container scheduling on heterogeneous clusters. In *European Conference on Parallel Processing*, pages 365–377. Springer, 2018.

[78] K.-C. Huang and B.-J. Shen. Service deployment strategies for efficient execution of composite saas applications on cloud platform. *Journal of Systems and Software*, 107: 127–141, 2015.

[79] C. Inzinger, S. Nastic, S. Sehic, M. Vögler, F. Li, and S. Dustdar. Madcat: A methodology for architecture and deployment of cloud application topologies. In *2014 IEEE 8th international symposium on service oriented system engineering*, pages 13–22. IEEE, 2014.

[80] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276. ACM, 2009.

[81] G. Juve and E. Deelman. Automating application deployment in infrastructure clouds. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pages 658–665. IEEE, 2011.

[82] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, et al. Morpheus: Towards automated slos for enterprise clusters. In *OSDI*, pages 117–134, 2016.

[83] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[84] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, pages 485–497, 2015.

[85] D. R. Karger. Global min-cuts in rnc, and other ramifications of a simple min-cut algorithm. In *SODA*, volume 93, pages 21–30, 1993.

[86] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[87] M. Klein. A primal method for minimal cost flows with applications to the assignment and transportation problems. *Management Science*, 14(3):205–220, 1967.

[88] D. Kliazovich, P. Bouvry, and S. U. Khan. Dens: data center energy-efficient network-aware scheduling. *Cluster computing*, 16(1):65–75, 2013.

[89] V. R. Konda and J. N. Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

[90] Y.-K. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE transactions on parallel and distributed systems*, 7(5):506–521, 1996.

[91] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

[92] B. M. Lake, T. D. Ullman, J. B. Tenenbaum, and S. J. Gershman. Building machines that learn and think like people. *Behavioral and Brain Sciences*, 40, 2017.

[93] S. Lee, R. Panigrahy, V. Prabhakaran, V. Ramasubramanian, K. Talwar, L. Uyeda, and U. Wieder. Validating heuristics for virtual machines consolidation. *Microsoft Research, MSR-TR-2011-9*, pages 1–14, 2011.

[94] W. Leinberger, G. Karypis, and V. Kumar. Multi-capacity bin packing algorithms with applications to job scheduling under multiple constraints. In *Parallel Processing, 1999. Proceedings. 1999 International Conference on*, pages 404–412. IEEE, 1999.

[95] P. Leitner, J. Cito, and E. Stöckli. Modelling and managing deployment costs of microservice-based cloud applications. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, pages 165–174. ACM, 2016.

[96] T. Li, Z. Xu, J. Tang, and Y. Wang. Model-free control for distributed stream data processing using deep reinforcement learning. *Proceedings of the VLDB Endowment*, 11(6):705–718, 2018.

[97] W. Li, P. Svärd, J. Tordsson, and E. Elmroth. A general approach to service deployment in cloud environments. In *Cloud and Green Computing (CGC), 2012 Second International Conference on*, pages 17–24. IEEE, 2012.

[98] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[99] A. Lodi, S. Martello, and D. Vigo. Recent advances on two-dimensional bin packing problems. *Discrete Applied Mathematics*, 123(1):379–396, 2002.

[100] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[101] Z. Á. Mann. Allocation of virtual machines in cloud data centers-a survey of problem models and optimization algorithms. *Acm Computing Surveys (CSUR)*, 48(1):11, 2015.

[102] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 50–56. ACM, 2016.

[103] P. Mell, T. Grance, et al. The nist definition of cloud computing. 2011.

[104] X. Meng, V. Pappas, and L. Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *2010 Proceedings IEEE INFOCOM*, pages 1–9. IEEE, 2010.

[105] D. Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[106] H. Mi, H. Wang, G. Yin, Y. Zhou, D. Shi, and L. Yuan. Online self-reconfiguration with performance guarantee for energy-efficient large-scale cloud computing data centers. In *Services Computing (SCC), 2010 IEEE International Conference on*, pages 514–521. IEEE, 2010.

[107] D. Milojičić, I. M. Llorente, and R. S. Montero. Opennebula: A cloud management tool. *IEEE Internet Computing*, 15(2):11–14, 2011.

[108] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[109] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[110] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver,

and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[111] A. I. Orhean, F. Pop, and I. Raicu. New scheduling approach using reinforcement learning for heterogeneous distributed systems. *Journal of Parallel and Distributed Computing*, 117:292–302, 2018.

[112] J. B. Orlin. A faster strongly polynomial minimum cost flow algorithm. *Operations research*, 41(2):338–350, 1993.

[113] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[114] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder. Heuristics for vector bin packing. *research. microsoft. com*, 2011.

[115] C. Peng, M. Kim, Z. Zhang, and H. Lei. VDN: Virtual machine image distribution network for cloud data centers. In *INFOCOM, 2012 Proceedings IEEE*, pages 181–189. IEEE, 2012.

[116] S. Rampersaud and D. Grosu. Sharing-aware online virtual machine packing in heterogeneous resource clouds. *IEEE Transactions on Parallel and Distributed Systems*, 28(7):2046–2059, 2017.

[117] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao. Efficient queue management for cluster scheduling. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 36. ACM, 2016.

[118] K. Razavi and T. Kielmann. Scalable virtual machine deployment using vm image caches. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 65. ACM, 2013.

[119] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 7. ACM, 2012.

[120] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *Proceedings of the 2015 ACM SIGMOD international conference on Management of Data*, pages 1357–1369. ACM, 2015.

[121] R. Sakellariou and H. Zhao. A hybrid heuristic for dag scheduling on heterogeneous systems. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, page 111. IEEE, 2004.

[122] A. Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[123] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. Legoos: A disseminated, distributed {OS} for hardware resource disaggregation. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 69–87, 2018.

[124] W. Shi and B. Hong. Towards profitable virtual machine placement in the data center. In *2011 Fourth IEEE International Conference on Utility and Cloud Computing*, pages 138–145. IEEE, 2011.

[125] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[126] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[127] W. Smith, I. Foster, and V. Taylor. Predicting application run times using historical information. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 122–142. Springer, 1998.

[128] M. Stillwell, D. Schanzenbach, F. Vivien, and H. Casanova. Resource allocation algorithms for virtualized service hosting platforms. *Journal of Parallel and distributed Computing*, 70(9):962–974, 2010.

[129] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[130] S. Taherizadeh, A. C. Jones, I. Taylor, Z. Zhao, and V. Stankovski. Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review. *Journal of Systems and Software*, 136:19–38, 2018.

[131] J. Thönes. Microservices. *IEEE software*, 32(1):116–116, 2015.

[132] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[133] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. Iperf: The TCP/UDP bandwidth measurement tool. *htt p://dast. nlanr. net/Projects*, 2005.

[134] J. Tordsson, R. S. Montero, R. Moreno-Vozmediano, and I. M. Llorente. Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers. *Future generation computer systems*, 28(2):358–367, 2012.

[135] L. Torrey and J. Shavlik. Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*, pages 242–264. IGI Global, 2010.

[136] W. Tsai, X. Bai, and Y. Huang. Software-as-a-service (SaaS): perspectives and challenges. *Science China Information Sciences*, 57(5):1–15, 2014.

[137] B. Vamanan, J. Hasan, and T. Vijaykumar. Deadline-aware datacenter tcp (D2TCP). *ACM SIGCOMM Computer Communication Review*, 42(4):115–126, 2012.

[138] L. M. Vaquero, A. Celorio, F. Cuadrado, and R. Cuevas. Deploying large-scale datasets on-demand in the cloud: treats and tricks on data distribution. *IEEE Transactions on Cloud Computing*, 3(2):132–144, 2015.

[139] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

[140] J. Wan, S. Tang, H. Yan, D. Li, S. Wang, and A. V. Vasilakos. Cloud robotics: Current status and open issues. *IEEE Access*, 4:2797–2807, 2016.

[141] J. Wang, A. Taal, P. Martin, Y. Hu, H. Zhou, J. Pang, C. de Laat, and Z. Zhao. Planning virtual infrastructures for time critical applications with multiple deadline constraints. *Future Generation Computer Systems*, 75:365–375, 2017.

[142] M. Wang, X. Meng, and L. Zhang. Consolidating virtual machines with dynamic bandwidth demand in data centers. In *Infocom*, volume 201, pages 71–75, 2011.

[143] W. Wang, B. Li, and B. Liang. Dominant resource fairness in cloud computing systems with heterogeneous servers. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*, pages 583–591. IEEE, 2014.

[144] W. Wang, B. Liang, and B. Li. Multi-resource fair allocation in heterogeneous cloud computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 26(10): 2822–2835, 2014.

[145] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: Meeting deadlines in datacenter networks. 41(4):50–61, 2011.

[146] G. J. Woeginger. There is no asymptotic ptas for two-dimensional vector packing. *Information Processing Letters*, 64(6):293–297, 1997.

[147] Q. Wu, Z. Wu, Y. Zhuang, and Y. Cheng. Adaptive dag tasks scheduling with deep reinforcement learning. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 477–490. Springer, 2018.

[148] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra. Hierarchical dag scheduling for hybrid distributed systems. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 156–165. IEEE, 2015.

[149] J. Xu and J. A. Fortes. Multi-objective virtual machine placement in virtualized data center environments. In *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social*

Computing, pages 179–188. IEEE Computer Society, 2010.

[150] G. Yao, Y. Ding, and K. Hao. Using imbalance characteristic for fault-tolerant workflow scheduling in cloud systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3671–3683, 2017.

[151] Z. I. M. Yusoh and M. Tang. A penalty-based genetic algorithm for the composite saas placement problem in the cloud. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010.

[152] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[153] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.

[154] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. *Proceedings of the VLDB Endowment*, 7(13):1393–1404, 2014.

[155] Z. Zhang, Z. Li, K. Wu, D. Li, H. Li, Y. Peng, and X. Lu. Vmthunder: fast provisioning of large-scale virtual machine clusters. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3328–3338, 2014.

[156] Z. Zhao, P. Martin, J. Wang, A. Taal, A. Jones, I. Taylor, V. Stankovski, I. G. Vega, G. Suciu, A. Ulisses, et al. Developing and operating time critical applications in clouds: the state of the art and the SWITCH approach. *Procedia Computer Science*, 68:17–28, 2015.

[157] Z. Zhao, A. Taal, A. Jones, I. Taylor, V. Stankovski, I. G. Vega, F. J. Hidalgo, G. Suciu, A. Ulisses, P. Ferreira, et al. A software workbench for interactive, time critical and highly self-adaptive cloud applications (switch). In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 1181–1184. IEEE, 2015.

[158] Z. Zhao, P. Martin, C. De Laat, K. Jeffery, A. Jones, I. Taylor, A. Hardisty, M. Atkinson, A. Zuiderwijk, Y. Yin, and Y. Chen. Time critical requirements and technical considerations for advanced support environments for data-intensive research. In *2nd International Workshop on Interoperable infrastructures for interdisciplinary Big Data sciences (IT4RIs) in the context of IEEE Real-time System Symposium (RTSS)*, 2016.

[159] H. Zhou, Y. Hu, J. Wang, P. Martin, C. De Laat, and Z. Zhao. Fast and dynamic resource provisioning for quality critical cloud applications. In *Real-Time Distributed Computing (ISORC), 2016 IEEE 19th International Symposium on*, pages 92–99. IEEE, 2016.

[160] H. Zhou, S. Koulouzis, Y. Hu, J. Wang, C. de Laat, A. Ulisses, and Z. Zhao. Migrating live streaming applications onto clouds: Challenges and a cloudstorm solution. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 321–326. IEEE, 2018.

[161] H. Zhou, A. Taal, S. Koulouzis, J. Wang, Y. Hu, G. Suciu, V. Poenaru, C. de Laat, and Z. Zhao. Dynamic real-time infrastructure planning and deployment for disaster early warning systems. In *International Conference on Computational Science*, pages 644–654. Springer, 2018.

[162] Z. Zhu, G. Zhang, M. Li, and X. Liu. Evolutionary multi-objective workflow scheduling in cloud. *IEEE Transactions on parallel and distributed Systems*, 27(5):1344–1357, 2015.

Summary

Cloud computing can provide virtualized, elastic, and on-demand computer system resources for supporting complex distributed applications, which has become the ubiquitous and primary computing paradigm for today's business. Meanwhile, the performance of cloud applications is getting more and more attention, particularly along with quality-critical applications which have critical requirements for Quality of Service (QoS) or Quality of Experience (QoE). It is because they can only achieve their expected value and social impact when the application performance is guaranteed. Typically, cloud applications involve distributed and parallel components to handle massive and complex tasks. During runtime, the application components often have diverse resource requirements, such as a combination of CPU, memory, and disk, which have to be satisfied by the underlying cloud infrastructure for running the application properly. Moreover, some application-specific properties, such as network traffic among collaborative services and dependency of batch tasks, require careful treatment by cloud resource schedulers for achieving the desired performance. When the scale of cloud applications and the complexity of cloud infrastructure increasingly grow, effective cloud resource scheduling mechanisms become extremely important.

We are thus motivated to investigate how to efficiently schedule resources to satisfy quality-critical requirements of diverse applications on cloud infrastructure. The scientific contributions presented in the thesis are as follows:

- We propose a Deadline-aware Deployment System (DDS) for deploying time critical applications in clouds. Considering the deadline constraints of deployment requests, we employ Earliest Deadline First (EDF) to prioritize the requests in DDS. Furthermore, we design a bandwidth-aware algorithm to achieve parallel transmission without causing network bandwidth contention. Experimental results show that DDS leverages network bandwidth sufficiently, and significantly reduces the number of missed deadlines during deployment.

- We propose an Enhanced Container Scheduler (ECSched) for efficiently scheduling concurrent container requests with multi-resource constraints on heterogeneous clusters. To handle concurrent requests, we formulate the container scheduling problem as a minimum-cost flow problem (MCFP) and represent the container requirements using a specific graph data structure (flow network). In the flow network, we propose a novel approach to encode the multi-resource demands and affinity requirements of requested containers. Experimental results show that ECSched can achieve better scheduling quality than state-of-the-art container schedulers, which can lower the average container completion time by up to 1.3× and noticeably improve resource utilization.

- We propose a new approach to optimize the placement of service-based applications in clouds. Our approach involves two key steps: 1) The requested application is partitioned into several parts while keeping overall traffic between different parts to a minimum. 2) The parts in the partition are packed into machines with multi-resource constraints. Combining these two steps, the proposed approach can find an appropriate placement solution for service-based applications in clouds. Experimental results show that our approach outperforms existing container cluster schedulers and representative heuristics, leading to much less overall inter-machine traffic.

- We present GoTask, an approach that can learn to well schedule DAG jobs on multi-resource clusters. GoTask directly learns scheduling policies from experience through deep reinforcement learning. In order to handle the complexity and scale of the DAG scheduling problem, we propose a two-stage approach in GoTask. The first stage leverages a deep reinforcement learning agent to learn policies for selecting a pending task of a DAG job, and the second stage leverages another agent to learn policies for selecting a machine to run the selected task. Experimental results show that GoTask outperforms commonly adopted scheduling heuristics, which improves job completion time by 6% to 18%.

Samenvatting

Cloud computing kan gevirtualiseerde, elastische en on-demand computersysteembronnen leveren voor het ondersteunen van complexe gedistribueerde applicaties en is het alomtegenwoordige en primaire computerparadigma voor het hedendaagse bedrijfsleven geworden. Ondertussen krijgen de prestaties van cloudapplicaties steeds meer aandacht, vooral samen met kwaliteitskritieke applicaties die kritische eisen stellen aan de Quality of Service (QoS) of de Quality of Experience (QoE). Dit komt omdat cloudapplicaties alleen hun verwachte waarde en sociale impact kunnen bereiken wanneer de prestaties van de applicatie zijn gegarandeerd. Cloudapplicaties omvatten doorgaans gedistribueerde en parallelle componenten om massieve en complexe taken uit te voeren. Tijdens runtime hebben de applicatiecomponenten vaak verschillende resourcevereisten, zoals een combinatie van CPU, geheugen en schijfruimte, waaraan moet worden voldaan door de onderliggende cloudinfrastructuur om de applicatie correct te laten werken. Bovendien vereisen sommige applicatiespecifieke eigenschappen, zoals netwerkverkeer tussen samenwerkingsservices en de afhankelijkheid van batchtaken, een zorgvuldige behandeling door cloud resource schedulers om de gewenste prestaties te bereiken. Wanneer de schaal van cloudapplicaties en de complexiteit van cloudinfrastructuur steeds groter worden, worden effectieve mechanismen voor cloud resource scheduling uiterst belangrijk.

We zijn dus gemotiveerd om te onderzoeken hoe we efficiënt resources kunnen plannen om te voldoen aan kwaliteitskritieke vereisten van diverse applicaties op cloudinfrastructuur. De wetenschappelijke bijdragen in het proefschrift zijn als volgt:

- We stellen een Deadline-bewust Deployment System (DDS) voor voor het implementeren van tijdkritische applicaties op clouds. Gezien de deadline beperkingen van implementatieverzoeken, gebruiken we Earliest Deadline First (EDF) om de aanvragen in DDS te prioriteren. Verder ontwerpen we een bandbreedtebewust algoritme om parallelle overdracht te bereiken zonder contentie met netwerkbandbreedte te veroorzaken. Experimentele resultaten tonen aan dat DDS voldoende gebruikmaakt van netwerkbandbreedte en het aantal gemiste deadlines tijdens de implementatie aanzienlijk vermindert.

- We stellen een Enhanced Container Scheduler (ECSched) voor voor het efficiënt scheduling van gelijktijdige containeraanvragen met multi-resource beperkingen op heterogene clusters. Om gelijktijdige aanvragen af te handelen, formuleren we het container scheduling probleem als een MCFP (Minimum Cost Flow Problem) en vertegenwoordigen we de containervereisten met behulp van een specifieke grafische gegevensstructuur

(stroomnetwerk). In het stroomnetwerk stellen we een nieuwe aanpak voor om de multi-resource-eisen en affiniteitsvereisten van gevraagde containers te coderen. Experimentele resultaten tonen aan dat ECSched een betere scheduling skwaliteit kan bereiken dan geavanceerde container schedulers, die de gemiddelde doorlooptijd van de container met maximaal 1.3× kunnen verkorten en het gebruik van middelen aanzienlijk kunnen verbeteren.

- We stellen een nieuwe aanpak voor om de plaatsing van op services gebaseerde applicaties in de cloud te optimaliseren. Onze aanpak omvat twee belangrijke stappen: 1) De gevraagde toepassing is verdeeld in verschillende partities terwijl het totale verkeer tussen verschillende partities tot een minimum wordt beperkt. 2) De onderdelen in de partitie zijn verpakt in machines met multi-resource beperkingen. Door deze twee stappen te combineren, kan de voorgestelde aanpak een geschikte plaatsingsoplossing vinden voor op services gebaseerde applicaties op clouds. Experimentele resultaten tonen aan dat onze aanpak beter presteert dan bestaande container cluster schedulers en representatieve heuristieken, wat leidt tot veel minder algemeen verkeer tussen machines.

- We presenteren GoTask, een aanpak die kan leren DAG-taken goed te plannen op clusters met meerdere bronnen. GoTask leert scheduling sbeleid rechtstreeks uit ervaring via deep reinforcement learning. Om de complexiteit en schaal van het DAG scheduling probleem aan te pakken, stellen we een tweefasenaanpak voor in GoTask. De eerste fase maakt gebruik van een deep reinforcement learning agent om beleid te leren voor het selecteren van een taak in behandeling van een DAG-taak, en de tweede fase maakt gebruik van een andere agent om beleid te leren voor het selecteren van een machine om de geselecteerde taak uit te voeren. Experimentele resultaten tonen aan dat GoTask beter presteert dan algemeen aanvaarde scheduling heuristieken, en de doorlooptijd van taken met 6% tot 18% verbetert.

Publications

Journals

- **Hu, Y.**, Zhou, H., de Laat, C. and Zhao, Z. Concurrent Container Scheduling on Heterogeneous Clusters with Multi-Resource Constraints. Future Generation Computer Systems, Volume 102, Pages 562-573. 2020, Elsevier.
- **Hu, Y.**, de Laat, C. and Zhao, Z. Optimizing Service Placement for Microservice Architecture in Clouds. Applied Sciences. (Under review)
- **Hu, Y.**, de Laat, C. and Zhao, Z. Learning DAG Scheduling with Multi-Resource Constraints on Heterogeneous Clusters. Concurrency and Computation: Practice and Experience. (Under review)
- Zhou, H., **Hu, Y.**, Ouyang, X., Su, J., Koulouzis, S., de Laat, C., Zhao, Z., CloudsStorm: A Framework for Seamlessly Programming and Controlling Virtual Infrastructure Functions during the DevOps Lifecycle of Cloud Applications. Software: Practice and Experience, Volume 49, Pages 1421-1447. 2019, Wiley.
- Koulouzis, S., Martin, P., Zhou, H., **Hu, Y.**, Wang, J., Carval, T., Grenier, B., Heikkinen, J., de Laat, C., Zhao, Z., Time-critical Data Management in Clouds: Challenges and a Dynamic Real-Time Infrastructure Planner (DRIP) solution. Concurrency and Computation: Practice and Experience, e5269. 2019, Wiley. (as co-first author)
- Wang, J., Taal, A., Martin, P., **Hu, Y.**, Zhou, H., Pang, J., de Laat, Cees., Zhao, Z., Planning Virtual Infrastructures for Time Critical Applications with Multiple Deadline Constraints. Future Generation Computer Systems, Volume 75, Pages 365-375. 2017, Elsevier.

Conferences

- **Hu, Y.**, Wang, J., Zhou, H., Martin, P., Taal, A., De Laat, C. and Zhao, Z. Deadline-aware Deployment for Time Critical Applications in Clouds. In 2017 European Conference on Parallel Processing (EuroPar) (Pages 345-357). Springer, Cham.
- **Hu, Y.**, Zhou, H., de Laat, C. and Zhao, Z. Ecsched: Efficient Container Scheduling on Heterogeneous Clusters. In 2018 European Conference on Parallel Processing (EuroPar) (Pages 365-377). Springer, Cham.

- **Hu, Y.**, de Laat, C. and Zhao, Z. Multi-objective Container Deployment on Heterogeneous Clusters. International Workshop on Network-Aware Big Data Computing, In Proceedings of 2019 IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID) (Pages 592-599). IEEE. (**Best paper award**)
- **Hu, Y.**, de Laat, C. and Zhao, Z. Learning Workflow Scheduling on Multi-Resource Clusters. In 2019 IEEE International Conference on Networking, Architecture and Storage (NAS) (Pages 1-8). IEEE.
- Zhou, H., **Hu, Y.**, Wang, J., Martin, P., De Laat, C. and Zhao, Z., 2016, May. Fast and Dynamic Resource Provisioning for Quality Critical Cloud Applications. In 2016 IEEE International Symposium on Real-Time Distributed Computing (ISORC) (Pages 92-99). IEEE.
- Zhou, H., Wang, J., **Hu, Y.**, Su, J., Martin, P., De Laat, C. and Zhao, Z. Fast Resource Co-provisioning for Time Critical Applications based on Networked Infrastructures. In 2016 IEEE International Conference on Cloud Computing (CLOUD) (Pages 802-805). IEEE.
- Wang, J., Zhou, H., **Hu, Y.**, de Laat, C. and Zhao, Z. Deadline-aware Coflow Scheduling in a DAG. In 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom) (Pages 341-346). IEEE.
- Elzinga, O., Koulouzis, S., Taal, A., Wang, J., **Hu, Y.**, Zhou, H., Martin, P., de Laat, C. and Zhao, Z. Automatic Collector for Dynamic Cloud Performance Information. In 2017 International Conference on Networking, Architecture, and Storage (NAS) (Pages 1-6). IEEE.
- Zhou, H., **Hu, Y.**, Su, J., de Laat, C. and Zhao, Z. Cloudsstorm: An Application-driven Framework to Enhance the Programmability and Controllability of Cloud Virtual Infrastructures. In 2018 International Conference on Cloud Computing (Pages 265-280). Springer, Cham.
- Zhou, H., Taal, A., Koulouzis, S., Wang, J., **Hu, Y.**, Suciu, G., Poenaru, V., de Laat, C. and Zhao, Z. Dynamic Real-Time Infrastructure Planning and Deployment for Disaster Early Warning Systems. In 2018 International Conference on Computational Science (Pages 644-654). Springer, Cham.
- Zhou, H., Koulouzis, S., **Hu, Y.**, Wang, J., de Laat, C., Ulisses, A. and Zhao, Z. Migrating Live Streaming Applications onto Clouds: Challenges and a CloudStorm Solution. In 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion) (Pages 321-326). IEEE.

- Shi, Z., Zhou, H., **Hu, Y.**, Surbiryala, J., de Laat, C., Zhao, Z., Operating Permissioned Blockchain in Clouds: A Performance Study of Hyperledger Sawtooth, In 2019 IEEE International Symposium On Parallel And Distributed Computing (ISPDC) (Pages 50-57). IEEE.

Other Publications

- **Hu, Y.**, de Laat, C. and Zhao, Z. ECSched: Efficient Container Scheduling on Heterogeneous Clusters. Poster in ICT.OPEN2018.
- **Hu, Y.**, de Laat, C. and Zhao, Z. GoDAG: Learning Workflow Scheduling in Multi-Resource Clusters. Poster in ICT.OPEN2019.

Prototypes

- DDS – Deadline-aware Deployment System.
  Code: `https://github.com/huyang1022/Deployment-Agent`
- ECSched – Enhanced Container Scheduler.
  Code: `https://github.com/huyang1022/ECSched`
- GoTask – Deep Reinforcement Learning Based Scheduler.
  Code: `https://github.com/huyang1022/RLSched`

Acknowledgements

Four years ago, when I decided to come to the Netherlands to pursue my PhD degree, I never imagined it was such a challenging journey. Finally, I arrived at the end of my PhD thesis and gained a precious experience. It is so important and meaningful not only because of the scientific knowledge I have gained, but also because I have learned a lot of life lessons that I will carry with me for the rest of my life. Fortunately, I was not alone in the journey. I am so grateful to all the people who have inspired and helped me throughout these unforgettable and unique years. Now, this is the best time I express my gratitude to the people for their support, encouragements, and friendship.

First of all, I would like to thank my promotor, Prof. Cees de Laat. You are always happy and patient to listen to my problems and give me enormous helpful and useful advice. I would like to thank my supervisor, Dr. Zhiming Zhao. You always show your passion about science and cheer up every member in your research group. I am pretty grateful for all of this help and tutoring.

I appreciate the supervision and suggestion from Prof. Xicheng Lu, Prof. Yuxing Peng and Prof. Dongsheng Li in China. You taught me the basic research skills and gave me the greatest support during my bachelor and master study. After I came to the Netherlands, you were still concerned about my PhD progress and always encouraged me.

I would like to express my gratitude to the committee members, Prof. Alexandru Iosup, Prof. Radu Prodan, Prof. Dongsheng Li, Prof. Rob V. van Nieuwpoort, Prof. Pieter Adriaans, and Dr. Adam Belloum for taking the time to read this thesis.

I would like to thank all my colleagues in the Systems and Networking (SNE) Lab. It was great to work with you, no matter the scientific problems or other interesting topics. You were so nice and created a warm atmosphere in the lab. Thank you very much from my deep heart to Ameneh, Ana, Ana, Andy, Arie, Benjamin, Catalin, Clemens, Dolly, Fahimeh, Giovanni, Giulio, Jamila, Joseph, Julius, Lukasz, Marijke, Mikolaj, Misha, Mostafa, Paola, Paul, Pieter, Ralph, Reggi, Sara, Simon, Spiros, Uraz, Yuri.

Furthermore, I extend sincere gratitude to all my Chinese friends: Biwen Wang, Hao Zhu, Hongyun Liu, Huan Zhou, Hui Xiong, Jian Lin, Jinglan Wang, Jun Xiao, Junchao Wang, Lingling Zhang, Long Cheng, Lu Zhang, Lu-Chi Liu, Peng Wang, Qi Wang, Renjie Lv, Ruyue Xin, ShuaiShuai Wang, Si Wen, Songyu Yang, Wenyang Wu, Xianya Mi, Xiaofeng Liao, Xiaolong Liu, Xin Zhou, Yifan Chen, Yiwei Sun, Yumei Wang, Zenlin Shi, Zeshun Shi, Zijian Zhou, Ziming Li. Thank you all for your company and for the beautiful moment you gave me. It was my pleasure to meet all of you and best wishes to all of you.

Ludan, it was so lucky to meet you during my PhD. Thank you for bringing the sunshine into my life and encouraging me during the most difficult time.

Last but not least, I would like to thank my family. Thanks for your understanding, endless love, and unconditional support. Words are too limited to express my gratitude to you.