KubeHICE: Performance-aware Container Orchestration on Heterogeneous-ISA Architectures in Cloud-Edge Platforms

Saqing Yang, Yi Ren, Jianfeng Zhang, Jianbo Guan, Bao Li
College of Computer
National University of Defence Technology
Changsha 410073, China
{yangsaqing, renyi, guanjb}@nudt.edu.cn

***Abstract*—To manage edge applications more conveniently, cloud providers extend the container and the container orchestrator from the cloud to the edge. The existing container orchestrators are designed for homogenous clusters at the cloud, while for nodes in cloud-edge computing environments, the hardware is commonly heterogeneous. Thus, current container orchestrators cannot automatically deal with heterogeneous-ISA. Besides, even if the nodes in a cloud-edge platform have the same ISA, their performance may be significantly different. The main reason lies in that the resource model of current container orchestrators is designed for homogenous clusters and assumes that all CPU cores of every node have the same performance, which will lead to low resource utilization and overload as far as heterogeneous clusters are concerned. In this paper, we present the design and implementation of KubeHICE, a performance-aware container orchestrator for heterogeneous-ISA architectures in cloud-edge platforms, which extends open source Kubernetes by AIM (Automatic Instruction Set Architecture Matching) and PAS (Performance-Aware Scheduling), two novel functional approaches that we propose. AIM is proposed for handling heterogeneous ISA, it can automatically find a node that is compatible with ISAs supported by the containerized application. PAS is designed for scheduling containers according to computing capability of cluster nodes. For PAS, we build a CPU resource allocation model to describe the computing capability differences of CPUs in a heterogeneous-ISA cluster. Based on this model, PAS is capable of estimating single-thread performance of a CPU through the workloads, which therefore enables KubeHICE to estimate the performance differences between containers that are assigned to heterogeneous nodes. Experimental results show that KubeHICE adds no additional overhead to container orchestration and it is effective in performance estimation and resource scheduling. We demonstrate the advantages of KubeHICE in several real-world scenarios, proving for example up to 40% increase in CPU utilization when eliminating heterogeneity. We believe that KubeHICE is not only technically beneficial to containerized applications in heterogeneous cloud-edge platforms, but also can a referable solution for Kubernetes community to support future IoT or edge server enabled container orchestration.**

***Index Terms*—Edge Computing, Cloud Computing, Scheduling, Heterogeneous ISA, Containers, Orchestration**

I. Introduction

The emergence and development of Internet of Things (IoT) as well as the success of rich cloud services have pushed the growth of edge computing, which brings applications and services away from centralized nodes and calls for data processing closer to data sources such as IoT devices or local edge servers [1]. Due to the proximity to data sources and user requests, edge computing delivers the potential advantages such as fast response time, lower power consumption and bandwidth cost saving, compared with cloud computing. Edge computing is performed at the edge of the network, processing downstream data on behalf of cloud services and upstream data on behalf of IoT services [2]. As an evolving computing paradigm, edge computing extends the cloud platforms and plays an important role in cooperation with cloud computing to provide fast and flexible services to end users.

As a kind of lightweight virtualization technology, container has not only been widely utilized in resource management and DevOps of cloud platforms and data centers in recent years but also gradually applied to some new fields such as edge computing [3]. Many applications are written and deployed in microservices enabled by the support of container technology to enable resource consolidation and to facilitate the orchestration, deployment, and maintenance of resources by encapsulation the applications into isolated lightweight virtualized environments. For Linux systems, container is built based on the namespace mechanism (e.g., Docker [4]) or highly optimized virtual machine (e.g., Kata Container [5]). Different from cloud nodes, hardware resources are provided by every edge device node are relatively limited. Since container has the advantages of being less resource consuming, providing near-native performance, and offering more refined resource control compared with traditional hypervisor-based virtual machine, it is more preferable for the resource virtualization of edge nodes and has been applied to quite a few edge platforms [6]–[9].

Nowadays, heterogeneity becomes common as far as different instruction set architectures (ISAs) and computing capability of the edge nodes are concerned nowadays [8], which is more obvious than in cloud data centers due to the diversity of edge devices such as various sensors, wearable devices, and IoT devices. For instance, in edge computing environments, the CPUs of most PCs and enterprise servers are in ISA of x86, while that of mobile devices are usually in ISA of ARM. And ARM-based enterprise servers, laptops, and PCs are increasing. Even though for CPUs in the same ISA, their computing capability is often not same due to CPU specification differences (e.g., frequency), architecture differences (e.g., scratch-pad memory [10]–[12]), the different number of cores, etc. The coexistence of different ISAs such as x86 and ARM, as well as heterogeneity in computing capability of each node, are common in edge environments. It is challenging to effectively use these heterogeneous resources as far as container orchestration is concerned.

Containers are deployed onto nodes of a cluster in the data center by orchestrators. Docker Swarm [13] and Kubernetes [14] are two typical open source orchestrators. For example, Kubernetes is widely used in public clouds, such as Google Kubernetes Engine (GKE), Amazon Elastic Kubernetes Service (EKS), etc. Through container orchestration, the resources such as CPU, memory, and network on each node are allocated to the deployed containers [15]. Nowadays, container orchestration is extended from clouds to edge platforms to improve resource management for containerized applications at the edge. KubeEdge [16], [17] is an emerging open source software for cloud-edge environments. It is implemented based on Kubernetes by extending the latter with functionalities such as network protocol infrastructure, runtime environment on the edge, and offline autonomy. OpenYurt [18] is another extention for cloud-edge environments, which provides similar functionalities as KubeEdge does. In this paper, we mainly focus on the research of Kubernetes based solutions since Kubernetes is one of the most widely used open source orchestrators currently.

At present, container orchestrators such as Kubernetes and Docker Swarm do not support container orchestration in heterogeneous cloud-edge platforms, which demand automatic ISA matching between worker nodes and container images. With the current release of Kubernetes, the deployment of container images onto the heterogeneous cloud and edge nodes will lead to container deployment failures. In order to avoid such failures, a large amount of manual work such as rules specification in deployment scripts is required, which is not desirable when thousands of containers are concerned. Furthermore, in heterogeneous clusters, the computing capability of different nodes are often significantly different. If existing scheduling methods are utilized directly without modification or extension into heterogeneous clusters, problems would occur. For example, when the same amount of workload is allocated to heterogeneous nodes (with the same number of CPUs), the low-performance nodes will be prone to be overloaded and the CPU utilization of high-performance nodes will be lower. Thus, the performance of containerized applications (e.g., QPS) that running on diverse nodes will be significantly different.

To address the above problems, in this paper, we present KubeHICE, a performance-aware container orchestration approach on Heterogeneous-ISA Architectures in Cloud-Edge Platforms, which is composed of two core methods we propose: AIM (Automatic Instruction Set Architecture Matching) and PAS (Performance-Aware Scheduling). AIM is proposed for handling heterogeneous ISA, it can automatically find a node that is compatible with ISAs supported by the containerized application. PAS schedules containers according to the computing capability of the nodes and estimates their performance by the statistic status of local applications. By deploying a monitor on each node that submits local containers' status in aggressive mode, we can get the status (e.g., CPU utilization) of containers. We design and implement KubeHICE based on open source Kubernetes Scheduler Framework, with the support of AIM and PAS. KubeHICE is fully compatible with existing controllers of Kubernetes and can be integrated into Kubernetes or KubeEdge as pluggable components to enable container orchestration for heterogeneous-ISA architectures in cloud-edge environments. We successfully tested KubeHICE on machines including servers and embedded devices with more than one ISA. We evaluate and demonstrate the benefits of KubeHICE. With KubeHICE, container deployment failures introduced by heterogeneous ISA in cloud-edge platforms can be avoided automatically and the CPU utilization is improved.

The approach of KubeHICE is also appliable to container orchestrators other than Kubernetes and KubeEdge. We believe that KubeHICE is not only technically beneficial to containerized applications in heterogeneous cloud-edge platforms but also a referable solution for open source Kubernetes community to support future IoT or edge server enabled container orchestration. The contributions of this paper are listed as below:

- We have investigated the current state of container ecosystem and container orchestration on heterogeneous-ISA architectures. Then we proposed AIM that is capable of automatically finding a node that is matchable to the ISA of the containerized application during the deployment of the container.
- Containers show different performance when they run on different nodes with diverse CPU capability. To improve CPU utilization and overall throughput, we propose PAS, a new approach of CPU resource allocation, to eliminate the difference by scheduling containers according to the computing capability.
- Based on Kubernetes and KubeEdge, we design and implement KubeHICE, an orchestrator for heterogeneous-ISA architectures. It can automatically avoid container deployment failures caused by ISA heterogeneity and significantly improves CPU utilization.

The rest of this paper is organized as follows. Section II presents our motivations as well as the preliminary background and analysis on container and container orchestration in a heterogeneous cloud-edge environment. Section III proposes PAS and AIM, and presents the design of KubeHICE. Section IV describes our implementation. Section V demonstrates comprehensive experimental evaluations and the results between our approaches and the native orchestration of Kubernetes. Section VI presents related works. The paper concludes with a discussion in Section VII.

II. Preliminary Background and Analysis

In this section, we discuss the motivation further in more detail to show the necessity and importance of container orchestration on heterogeneous-ISA architectures in cloud-edge platforms. We also give an introduction to container and container orchestrator mechanisms. Then we analysis several problems we found in the most popular container orchestrator on heterogeneous-ISA architectures.

A. Motivation

Fig. 1 shows the heterogeneous-ISA architecture in cloud-edge. Developers implement applications and compile different versions of executable binary files for multi-platforms. Then, the application's binary, environment variables, and other data are encapsulated into a container image. There are different versions of images for an application, e.g., one for x86 based platform and the other for ARM based. Finally, images are pushed to the image repository (e.g., Docker Hub). The control plane runs in the cloud is responsible for controlling work nodes as well as edge nodes. It assigns the containers encapsulating specific applications to the target node, which are possible in the cloud, on edge servers, or edge embedded devices. And the edge nodes are managed by the edge extension of Kubernetes such as KubeEdge, which works as a proxy of the edge nodes for the control plane. Reacting to the requests from the control plane, the container engines (e.g., Docker) on each local node pull container images from the image repository or local caches (e.g., Docker cache), and then create container instances based on container images. Cloud nodes are accessible by edge nodes, but cloud nodes cannot directly access edge nodes.

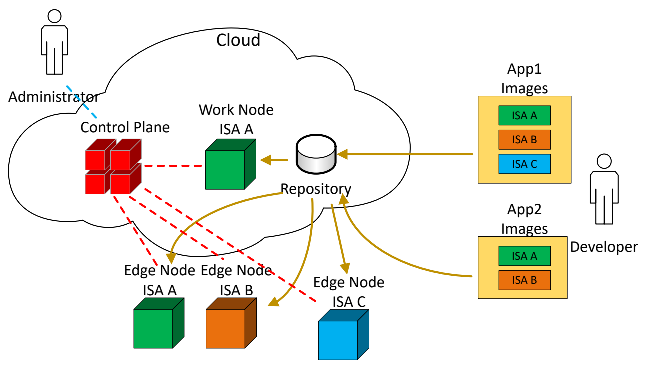


Fig. 1. The heterogeneous-ISA architecture in cloud-edge

The performance of heterogeneous nodes is different and the performance of an application varies under different CPU quotas. To demonstrate that, we use Docker Container to control the available CPU resource of an application, and tests the QPS of Nginx on five different nodes under different CPU resource quotas, as shown in Fig. 2(a). VM1 and VM3 are two VMs on an Intel x86-64 2.9GHz physical machine, VM 2 is a VM on an Intel x86-64 2.6GHz physical machine, Edge 1 and Edge 2 are two ARM64 1.5GHz Cortex-A72 embedded computers. The results show that an application running on homogeneous nodes gets similar performance if the same amount of CPU resource is assigned. But the performance of applications under the same resource assignment configuration on heterogeneous nodes is significantly different. In Fig. 2(a), the performance of Nginx (i.e., QPS) is linearly related to CPU resource quota, which gives us a clue that the performance of an application can be tuned through adjusting the CPU quota or by deploying it onto different nodes with diverse CPU capability. We also illustrate the test results that under the same QPS, there is a significant difference in CPU usage among heterogeneous nodes, as shown in Fig. 2(b). For current container orchestrators (e.g., Kubernetes), they do not distinguish the CPU heterogeneity on the different nodes. Thus, system failures during image pulling often occur and low CPU utility is inevitable as far as heterogeneous nodes are concerned.

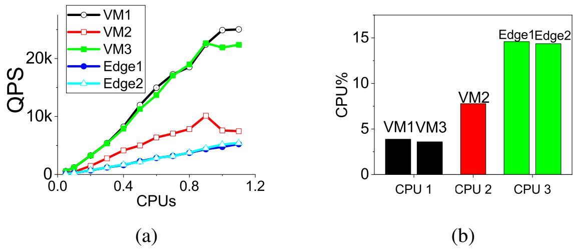


Fig. 2. (a) Nginx QPS of heterogeneous nodes under different CPU quotas; (b) CPU utilization with the same QPS.

B. Container and Container Orchestrator

Container-based virtualization is an approach to isolate guest processes with which the virtualization layer runs as a layer within the host OS. And the kernel is shared by the host OS and the isolated guests on top of it. It is a lightweight approach compared with hypervisor-based virtualization. Due to inevitable flaws in the kernel, guest containers could break through the isolation layer to use these flaws, threatening the host OS and other guest containers. The most common way to deal with this problem is to nest containers to a VM at present. It uses the features of the VM's strong isolation and the container's lightweight (e.g., quick start). Another way is to use the hypervisor-based container, similar to the first way, except that the guest kernel is highly optimized. This type of container, such as Kata Container [19] and gVisor [5], is recognized as a micro VM, which provides strong isolation, but without a significant increase in startup time and overhead.

Open Container Initiative (OCI) defines standards of interfaces among components of the container ecosystem. Thus, developers can easily add new features to existing container engines. Container images and image repositories are also defined in OCI, so that container images can be shared by different container engines.

Since we focus on the optimization of Kubernetes in the heterogeneous cloud-edge environment, we will give a brief introduction of its basic mechanisms here. Kubernetes is well designed and implemented so that it is efficient and at the same time eases the management of processes associated with deploying, scaling, and managing containerized applications. The main components of Kubernetes are:

- **Pod.** Pod is the basic working unit in Kubernetes. Each Pod encapsulates a single container or a set of containers, scheduled to run on one worker node, with all containers inside the Pod sharing the same IP address and hostname as well as other resources. Furthermore, different deployment rules can be set for Pods using *Controllers*. Pods are deployed and managed by *Controllers*. For example, *Deployment* is a *Controller* for managing a stateless service application workload that has one or more replicas in a cluster.
- **Worker nodes.** Worker nodes are VMs or physical machines that perform the assigned tasks. For each worker node, one or multiple Pods can run on it.
- **Control plane.** Control plane controls Kubernetes nodes. *API Server*, *Controller-Manager*, *Scheduler*, and *Etcd* are the principal components of the control plane. The control plane manages and schedules the different resources existing in the cluster, such as storage, memory, CPUs, network ports. All components communicate through *API Server*. The *Controller Manager* is responsible for nodes notification, populating Pods, creating API access tokens, etc. The *Scheduler* is responsible for scheduling Pods onto nodes. When a new Pod is created, the *Scheduler* selects a node from nodes available in the cluster and assigns the Pod to that node. The *Etcd* is a key-value store used as Kubernetes' backing store for all cluster data.

In the current releases of Kubernetes, the location of applications is insignificant since the nodes in the cluster are homogeneous. And containers are allocated CPU resources that 1 CPU is equivalent to 1 vCPU/Core for cloud providers and 1 hyperthread on bare-metal Intel processors. Containers are allowed to use a CPU proportionally when the value of CPU resource less than one.

C. Container Orchestration on Cloud-Edge

The constraints in Kubernetes such as homo network management and the limitation of its resource assignment prevent it from working correctly and smoothly in an edge or cloud-edge heterogeneous-ISA environment. At present, there are several kinds of solutions for edge container orchestration in the industry. One kind of solution is to reduce the hardware cost by tailoring some non-essential capacity, such as K3s [20]. The other is to manage edge devices by the cloud through an edge proxy such as KubeEdge [16], [17] and OpenYurt [18], which are both implemented based on Kubernetes. For example, KubeEdge connects and coordinates nodes in the cloud-edge environments for applications leveraging computing resources from both the two kinds of nodes to achieve better performance and user experience. To handle the heterogeneity of nodes, OpenYurt provides a node grouping capability that divides heterogeneous nodes into several homogeneous groups by adding labels to nodes, and the Kubernetes scheduler can schedule containers in a group by matching these labels. However, they both do not support automatic container orchestration in such a heterogeneous environment.

D. Container on Heterogeneous-ISA architectures

Native compiled binaries and libraries for an ISA cannot be deployed and executed on a machine with another different ISA. And container images, which encapsulate executable binaries, libraries, environment variables, etc., cannot be deployed on a physical node with incompatible ISA. The size of container images for different ISA of an application is also different, but we investigate several most popular official images on Docker Hub, and the result is that most of the differences are less than 10%, so this difference can be ignored. In the existing container ecosystem, there are two approaches to deploy multi-architecture container images onto nodes in a heterogeneous-ISA cluster through Kubernetes:

- **By image name.** Each image name is unique in a node or repository. The container engine finds the container image with a specific name from the image repository and pulls it to run on the local node. To support multiple platforms, developers build different versions of images compatible to different platforms, and the ISA information is embedded in the image name(e.g., *app-arm:v1* or *app:v1-amd64*). Users can get the ISA info of an image from its image name, but since there is no unified naming standard, failures occur often when heterogeneous nodes are expected to be automatically deployed and pulled.
- **By image index.** Image index is a higher-level manifest which contains one *index name*, one or more images' manifests, and platform fields (e.g., *architecture*). It is specified by the developers and pushed with the corresponding container images to the image repository. In the request to pull an image, platform-specific parameters are supposed to be included so that the image matching the platform-specific parameters is returned by the image repository. However, the performance of CPUs with different ISAs varies and their energy consumption is different also, which leads to the fact that the quota of 1 CPU assigned to an image is usually not equivalent as far as CPU heterogeneity is concerned. Thus, the existing CPU resource model in Kubernetes is not adaptable to such a situation since it can not handle this kind of difference. As a result, the scheduler in current releases of Kubernetes cannot efficiently orchestrate the images onto nodes with heterogeneous CPU resources.

The deployment script and image index of above two existing approaches are as follows:

```
# The first approach: by image name
# A "deployment" on amd64 nodes
containers:
- image: app-amd64:v1 # the "image name"
nodeSelector:
  kubernetes.io/arch: amd64
```

```
# A "deployment" on arm64 nodes
containers:
- image: app-arm64:v1
nodeSelector:
  kubernetes.io/arch: arm64

# The second approach: by image index
# The image index of the "app:v1"
image: app:v1 # the "index name"
manifests:
- image: app-amd64:v1
  platform:
    architecture: amd64
- image: app-arm64:v1
  platform:
    architecture: arm64
# A "deployment" on amd64 and arm64 nodes
containers:
- image: app:v1
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
        - matchExpressions:
            - key: kubernetes.io/arch
              operator: In
                values:
                  - amd64
                  - arm64
```

The label of architecture (*kubernetes.io/arch*) is automatically added to each node during initialization.

III. Design

A. Design Goals

To fulfill the demands for compatibility with existing container software ecosystem and flexibility at the cloud-edge, we design KubeHICE for the cloud-edge environment as shown in Fig. 1 with the following goals:

- Capable of automatic identification of the ISA supported by the containerized application and to be able to assign the image to the ISA matching nodes.
- Compatible with existing container ecosystems that neither follow uniform image naming rules nor provide *image index* mechanisms and be able to assign the container images to heterogeneous nodes that support more than one ISA.
- Provide solutions to support automatic estimation of the single-thread performance of each node.
- Be able to guarantee applications' performance and improve the CPU utilization on heterogeneous-ISA architectures through performance-aware container image orchestration.

B. An Overview of KubeHICE

We present KubeHICE, a performance-aware container orchestration approach on heterogeneous-ISA architectures for cloud-edge platforms, as illustrated in Fig. 3. The *Controller Manager* is responsible for controlling the number of replicas of the container and managing access permissions. The *Scheduler* assigns containers to work nodes. *AIM* reads the ISA information of the containerized application from *Etcd* and automatically matches ISA. *PAS* schedules the container according to the performance estimation for each node. The *Performance Analyzer* reads the data to estimate nodes' performance. *Kubelet* is a daemon running on each node that is

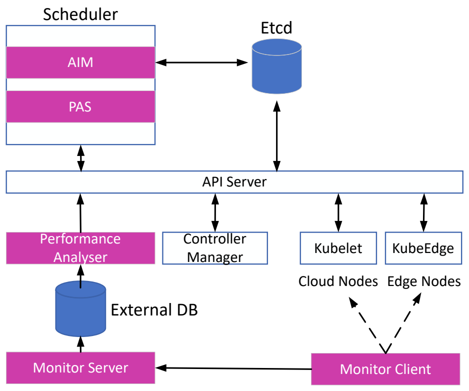


Fig. 3. Overview of KubeHICE

responsible for creating container instances. *KubeEdge* is the edge extension of Kubernetes that manages edge nodes. The *Monitor Client* runs on each node and reports the CPU usage status of containers to the *Monitor Server*. The *Monitor Server* stores the data in the *External DB*. Note that in Fig. 3, the rectangles filled with purple are the incremental components that we design to extent and optimize existing Kubernetes.

The workflow of KubeHICE is shown in Fig. 4 with the following steps:

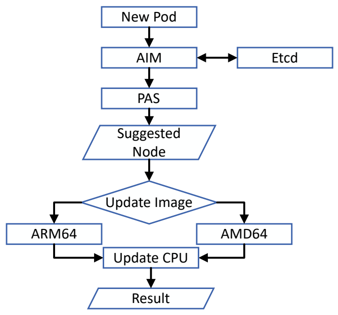


Fig. 4. Workflow of KubeHICE

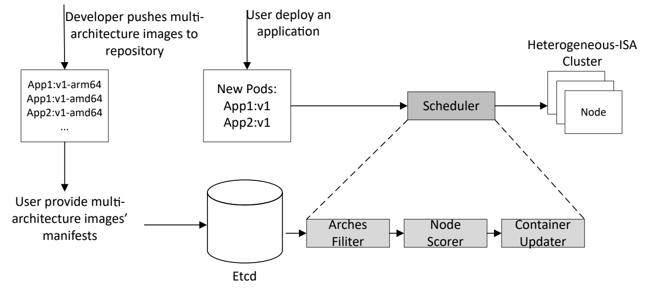
1) After receiving a new Pod, *AIM* reads the set of platforms supported by the application from the *Etcd* and then filters non-compatible nodes. The database stores fields of the application name, the image name, and the information of the platform.
2) To make KubeHICE CPU resource allocation approach compatible with existing scheduling methods, *PAS* converts the original CPU resource value by multiplying it with the parameter read from the node label. The converted value reflects the real capacity of the CPU resource in heterogeneous clusters and can be used for CPU resource-related scheduling such as *NodeResourcesBalancedAllocation*, a scheduler plugin of the native Kubernetes scheduler.
3) *PAS* sets the priority of alternative nodes according to the performance of the nodes and the CPU resource allocation parameters in the container configuration to

avoid scheduling heavy workloads to nodes with low performance.

4) After the location (called *suggested node*) of a container (or Pod in Kubernetes) is determined. Since some parameters (e.g., *image*) are related to the *suggestion node*, a temporary value is filled in during initialization. The configuration of the container will be updated according to the architecture and performance of the *suggested node*. Container Engine creates a new container instance based on the updated container configuration.

C. Automatic ISA Identification and Matching

At present, the container technology is most commonly used on Linux based systems. Thus, we mainly consider the ISA heterogeneity under Linux platform. We store the application's platform information in the *Etcd* and we do not require the image repository to support the image index and to provide the image name with a unique and standard format. The image field of the deployment configuration is filled with the application name (e.g., *App1:v1* in Fig. 5). When creating



Developer pushes multi-architecture images to repository
User deploy an application
Heterogeneous-ISA Cluster
App1:v1-arm64
App1:v1-amd64
App2:v1-amd64
...
New Pods:
App1:v1
App2:v1
Scheduler
Node
User provide multi-architecture images' manifests
Etcd
Arches Filiter
Node Scorer
Container Updater

Fig. 5. Automatic ISA matching and performance-aware scheduling

a new container, the first step in Fig. 5 is to check the supported architectures (Arches Filter) and then to filter the ISA-incompatible nodes in the cluster. The available nodes after the *Arches Filter* may still have more than one ISA. The Kubernetes Pod contains more than one container and must be deployed on the same node, so filter nodes by the intersected set of all containers ($Pod.Arches = Cluster.Arches \cap App1.Arches \cap App2.Arches \cap App3.Arches \cap ...$).

After getting the *suggested node*, update (Container Updater) the image field to image name (e.g., *App1:v1-arm64* in Fig. 5).

D. Performance-aware Scheduling

Due to the performance difference between nodes in the cluster, the existing resource allocation model of the CPU is inaccurate and needs to be enhanced by establishing an optimized one. In the traditional CPU resource allocation model, 1 CPU is equivalent to 1 vCPU/Core for cloud providers and 1 hyperthread on bare-metal Intel processors. But the capacity of CPU cores is not equivalent for CPU with heterogeneous-ISA architectures. As shown in Fig. 6(a), 2 applications need 1 CPU and 0.2 CPU respectively. These applications work fine on Node 1, but when running on Node 2 with the same CPU configuration, the performance will be half of that on Node 1. Applications are location-related in heterogeneous clusters.

Container runtime (e.g., runC [21]) uses Linux cgroups to achieve fine-grained CPU control. Container instances use CPU time proportionally according to CPU resource quota. When the CPU resource quota is less than one core, the application performance (QPS in Fig, 2) increases linearly with the increase of the CPU resource quota. A single application in a microservice architecture is generally simple enough and requires less than one core. Therefore, our architecture adjusts CPU quotas of the container to eliminate performance differences according to the performance of a node.

We design a CPU resource allocation approach for the heterogeneous environment, as shown in Fig. 6(a). Under this approach, we define a parameter k to describe the single-thread performance of nodes. k_j represents the single-threaded performance of node j (e.g., $k_1 = 1, k_2 = 0.5$ in Fig. 6). When deploying a container, specify a parameter k_b, which indicates that the CPU resource parameter of the container is configured based on the node with k_b capability (e.g., $k_b = 1$ in Fig. 6). When the container is assigned to node j, PAS will allocate CPU resource by update the CPU resource parameter $CPU = CPU \cdot k_b/k_j$. Application 2 has the same performance on node 2 and node 1. Applications are not location-related in heterogeneous clusters.

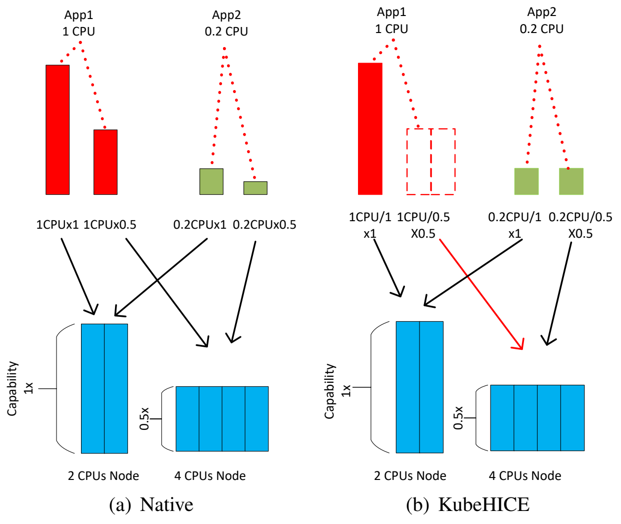


Fig. 6. CPU resource allocation of Native Kubernetes and KubeHICE

k_j determines the CPU resource allocation. The previous approaches mainly use a variety of benchmarks to run on each node respectively and get the node's performance according to the test results. The kinds of workloads are diverse, so it is difficult to test completely, and there will be additional overhead. According to container status to estimate the performance of the node is available in the cluster, as shown in Fig. 1, the CPU utilization of running containers reflects the node's performance and can be easily acquired from modern monitor platforms (e.g., Prometheus [22]), load level(e.g., the number of requests, forwarding weights), etc. In Kubernetes, the container control mode is shown in Fig. 7.

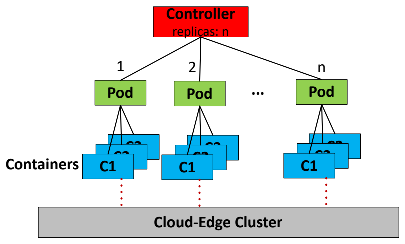


Fig. 7. Container control model in Kubernetes

A controller controls *n* replicas of Pods, which are the same and deployed on multiple nodes. The self-healing of Kubernetes is to check the number of "healthy" Pods and create new Pods if the number is less than *n*. The performance of nodes can be estimated by the CPU utilization rate of these containers in Pods, as shown in formula (1)(2)(3)(4):

$$W_{ij} = C_{ij}k_j = C_{i0}k_0\Delta W_{ij0}, \Delta W_{ij0} = W_{ij}/W_{i0} \tag{1}$$

$$k_j = C_{i0}k_0\Delta W_{ij0}/C_{ij} \tag{2}$$

$$P_{ij} = C_{ij}k_j/W_{ij}, \overline{P_i} = Geomean(P_{ij}) \tag{3}$$

$$\Delta P_{ij} = |1 - P_{ij}/\overline{P_i}| \tag{4}$$

W_{ij} - Load level of container i on node *j*.

W_{i0} - Load level of container i on reference node 0, k_0 is known.

C_{ij} - CPU utilization of container i on node *j*.

ΔW_{ij0} - Ratio of container i load levels between node j and node 0.

P_{ij} - CPU utilization of the unit load of container *i*, ideally the same of all nodes.

ΔP_{ij} - Error of P_{ij}.

Frequent update of k incurs additional overhead, so update only if ΔP_{ij} is larger than a threshold. When there are multiple different applications, k is calculated in a weighted mean.

Some of the applications have high requirements for single-threaded performance. In this situation, more resource allocation does not guarantee expected performance for certain. For example, application 1 in Fig. 6(b) cannot be assigned to Node2, even if Node2 allocates 2 CPUs to it, the reason is that application 1 can only use up to 1 CPU. Therefore, to guarantee applications' performance, our architecture follows the formula (5) when calculating node priority. Each node is traversed only once, so it is linear time complexity.

$$S = \begin{cases} L_0 & k'_j \geq 1, or \frac{C_{max}}{k'_j} \leq 1 \\ L_1 + (L_0 - L_1)k'_j & \frac{C_{min}}{k'_j} \geq \lceil C_{min} \rceil, 1 < \frac{C_{max}}{k'_j} \leq \lceil C_{max} \rceil \\ L_2 + (L_1 - L_2)k'_j & \frac{C_{min}}{k'_j} \leq \lceil C_{min} \rceil, 1 < \frac{C_{max}}{k'_j} \leq \lceil C_{max} \rceil \\ L_3 & \frac{C_{min}}{k'_j} > \lceil C_{min} \rceil \end{cases} \tag{5}$$

S - The score of a node, the higher the priority. The range is from 0 to 100. 0 indicates that the performance of the node does not meet the minimum requirements of the application.

L_i - $100 = L_0 > L_1 > L_2 > L_3 = 0$, L_0 means no performance loss, L_1 means there may be a performance loss, L_2 means that the minimum running requirements are met but there will be a peak performance loss, L_3 means the performance is too low to run the application, the value of L_1 and L_2 can be adjusted.

k'_j - $k'_j = k_j/k_b$.

C_{min} - Minimum CPU resource requirement of a container, the sum of C_{min} of all containers on a node can't be greater than the node's total resource, *resources.requests["cpu"]* in Kubernetes.

C_{max} - Maximum CPU resource requirement of a container, *resources.limits["cpu"]* in Kubernetes, $C_{max} \geq C_{min}$.

IV. Implementation

We implement KubeHICE in Golang language with about 1500 lines of code. The implementation of KubeHICE is based on the Kubernetes Scheduler Framework. We integrate KubeHICE into Kubernetes and KubeEdge. As shown in Fig. 8, KubeHICE is composed of 5 modules. They are Arches Filter, Node Scorer, Container Updater, KubeHICE Monitor, and Performance Analyzer.

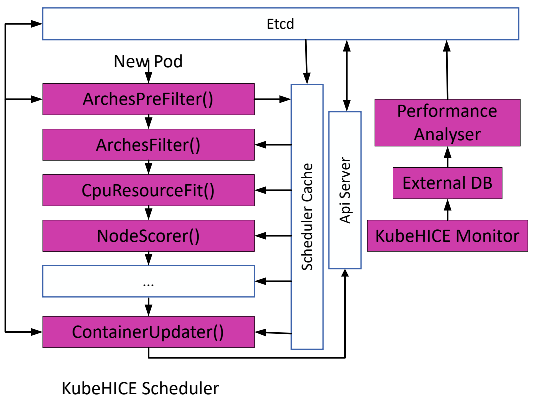


Fig. 8. Implementation of KubeHICE

A. Arches Filter

KubeHICE Scheduler receives the new Pod whose parameter *image* is the application name (i.e., *App1:v1* in Fig. 5). *ArchesPreFilter()* queries the *Etcd* according to *image* to get available architectures (*arches*) and writes *arches* to *Scheduler Cache* (i.e., *framework.CycleState*), and then *ArchesFilter()* filters worker nodes according to *arches*. The architecture of nodes is read from the node label *kubernetes.io/arch*.

If the container image doesn't support image index, users must input platform information to *Etcd* manually, otherwise querying the manifest automatically.

B. Node Scorer

The implementation is depicted in formula (5). The priority of a node is measured by a score with a range of 0 to 100. The higher the score, the higher the priority. Besides, if the score is equal to 0 (i.e., L_3), it means that the performance of this node is too low to meet the minimum requirements for the container and must be filtered. k_b is configured as a Pod label, k_j is read from the node label *hice/performance*, which is set and updated by *Performance Analyzer*. To improve the performance of the scheduler, when the number of available nodes is more than 50, Kubernetes usually select a subset of nodes to *scorer()*. Since there is more than one *scorer()* in Kubernetes, a weighted sum will be calculated and sorted to obtain the optimal node.

C. Container Updater

Parameters (i.e., *image*, *resources.requests["cpu"]*, *resources.limits["cpu"]*) of the new Pod are determined after getting the *suggested node*. The native *Kubernetes Scheduler* and *API Server* cannot update parameters of a container object. Thus, we implement a *ContainerUpdater()* that reads the Pod object from *Etcd*, updates these three parameters. Finally, write the Pod object back to the Etcd to complete the node binding. If the schedule fails, roll back to the original value.

D. KubeHICE Monitor

Because cloud nodes cannot directly access the edge nodes due to the constraint of network architecture. Thus, we implement the *KubeHICE Monitor*, which contains a server and a client. The server is deployed in the cloud, while the Client is deployed on each node to actively report container status. The server receives the status from the client and stores it in the External DB. The client reads the container status from *Kubelet* on cloud nodes or *KubeEdge EdgeCore* on edge nodes.

E. Performance Analyzer

The *Performance Analyzer* reads data from the *External DB*, evaluate nodes' CPU performance as formula (1)(2)(3)(4). The parameter k is calculated according to the load-balancer forwarding weight, the number of requests, and so on. After k has been calculated, write the value to the node label *hice/performance*.

V. Evaluation

KubeHICE is evaluated in three aspects: 1) we evaluate the validity of automatic ISA matching of KubeHICE by deploying container appliances in 4 scenarios; 2) we measure the ability to estimate the performance factor k through 3 real-world containerized applications; 3) we verify the effectiveness of performance-aware scheduling.

A. Experimental Setup

We evaluate KubeHICE in a test environment shown in Fig. 9. The Kubernetes control plane, KubeHICE scheduler, and *Performance Analyzer* are deployed to run on a cloud VM. And *Arches Filter*, *Node Scorer*, and *Pod Updater* are included in the KubeHICE scheduler. For the network enviroment, Kubernetes control plane and the three cloud nodes (VM1, VM2, VM3) run in the overlay container network in the cloud. And the three edge nodes (Edge1, Edge2, Edge3) each run in their subnets. The test server configurations are specified in Table I.

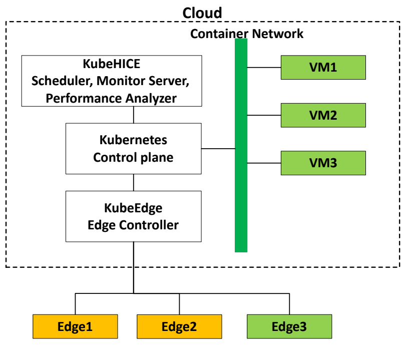


Fig. 9. KubeHICE test environment

TABLE I
SERVER CONFIGURATION IN TEST

Server	Configuration	
	Hardware	*Operate System*
VM1	Amd64, 2 core, 2.9GHz, 4G memory	Ubuntu 18.04.5
VM2	Amd64, 2 core, 2.6GHz, 4G memory	Ubuntu 18.04.5
VM3	Amd64, 2 core, 2.9GHz, 4G memory	Ubuntu 18.04.5
Edge1	Arm64, 4 core, 1.5GHz, 3.7G memory	Ubuntu 20.04.2
Edge2	Arm64, 4 core, 1.5GHz, 3.7G memory	Ubuntu 20.04.2
Edge3	Amd64, 2 core, 2.6GHz, 4G memory	Ubuntu 18.04.5

B. Evaluation of Automatic ISA Matching

We have found four styles of multi-architecture images in Docker Hub and tested them. The results in Table II: 1) *0* means that a successful deployment; 2) *1* means that an image-pull error because the container was allocated to a node, but the node's ISA is not found in the image index in the image repository; 3) *2* means that a container-create error that the image has been pulled successfully, but the image does not match the ISA of the node, so the container instance cannot be created.

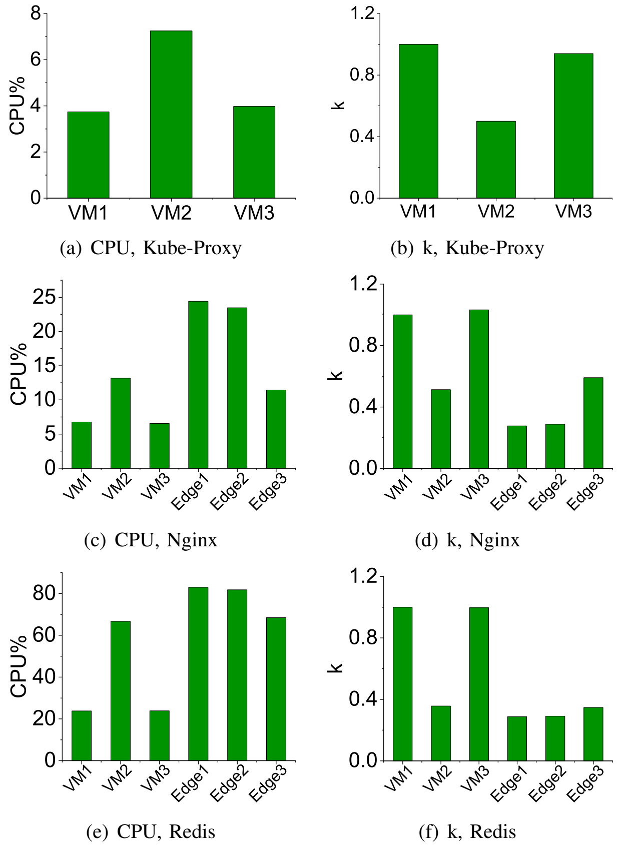
TABLE II
DEPLOYMENT RESULT IN TEST

Result		Style	
Native	*KubeHICE*	*Support Image Index?*	*Supported ISAs*
0	0	True	Arm64,Amd64,etc.
1	0	True	Arm64(or Amd64)
2	0	False	Arm64,Amd64,etc.
2	0	False	Arm64(or Amd64)

The test results show that KubeHICE can automatically match the ISA between images and nodes to avoid deployment errors. In terms of obtaining the architecture supported by the application, we tested several different repositories. For applications that support image indexes, the time of querying the manifest file ranges from less than *1s* (local) to a few minutes (Docker Hub). For applications that do not support image indexes, users must manually enter them into the *Etcd*. The overhead of querying *Etcd* is very low and can be ignored.

C. Evaluation of Estimating Nodes Performance

We estimate the node performance k by several real-world workloads that contains multiple replicas. Fig. 10 shows 3 workloads: 1)Kube-Proxy is a network component of Kubernetes that is deployed on every cloud node; 2) Nginx is a popular Web server, load balancer, reverse proxy server, and so on; 3) Redis is a popular key-value cache. The test results show



(a) CPU, Kube-Proxy (b) k, Kube-Proxy
(c) CPU, Nginx (d) k, Nginx
(e) CPU, Redis (f) k, Redis

Fig. 10. Estimate k based on CPU utilization of three real-world applications

that the application on homogeneous nodes (VM1&VM3, VM2&Edge3) performs similarly. The error range of Kube-Proxy is less than 3%. And that of Nginx is less than 8%. For Redis, the error range is less than 3%. The k_j calculated according to different types of workloads is approximate, though there are some differences. Among them, the error of VM2 and Edge3 is obvious that is about 22% but within the acceptable range. The results show that the performance difference of nodes can be estimated accurately according to the existing workload.

D. Evaluation of Performance-aware Scheduling

We deploy a Web service, containing two replicas of the Nginx Pod (only contains an Nginx container), controlled by a *Deployment Controller*. The two Pods are served out through a load balancer that uses a polling forward policy. The Web service is stress-tested under Kubernetes native and KubeHICE scheduling results. The test results are shown in Fig. 11. Compared with scheduling among homogeneous

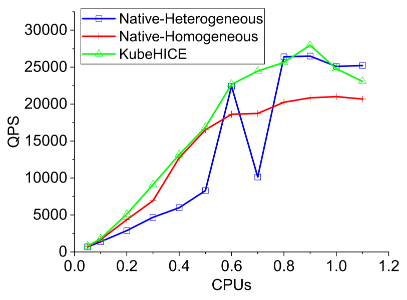
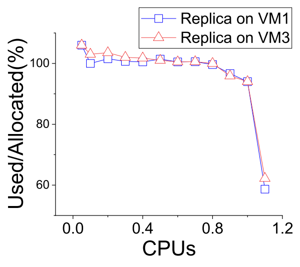


Fig. 11. CPU allocation and QPS of the Web service

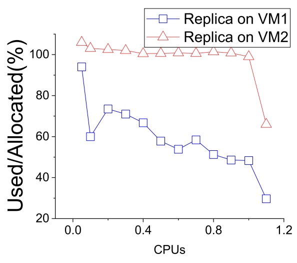
nodes (the line of Native-Homogeneous in Fig. 11), the QPS of the Web service suffers a loss when the native scheduler is scheduling among heterogeneous nodes. The line of Native-Heterogeneous in Fig. 11 shows that when the CPU quota is less than 0.5CPU and equal to 0.7CPU, one on VM2 and the other on VM1/VM3, the instance at VM2 became a performance bottleneck, and the CPU resources of the other were not fully utilized. KubeHICE schedules the two replicas among heterogeneous nodes (the line of KubeHICE in Fig. 11), allocates CPU resources to the container based on the CPU resource allocation approach shown in Fig. 6(b). Even if one of the replicas is allocated to VM2, but the resource allocation is adjusted according to the performance difference, the Web service still maintains performance comparable to the schedule between homogeneous nodes. When the CPU quota more than 0.5CPU (the line of KubeHICE in Fig. 11), KubeHICE scheduler can avoid assigning the container to low performance Node (i.e., VM2).

We test the CPU utilization in the different scenarios as shown in Fig. 12. The CPU utilization of two replicas scheduled on homogeneous nodes is close to 100%. When the native scheduler schedules the two replicas on heterogeneous nodes, the CPU utilization of the replica on VM2 (lower performance than VM1) is close to 100%, but the average CPU utilization of the replica on VM1 is less than 60%. The CPU utilization of two replicas scheduled by KubeHICE on heterogeneous nodes is close to 100% and the improvement of CPU utilization of the replica on VM1 is up to 40%. The evaluation shows that performance-aware scheduling of KubeHICE can efficiently improve CPU utilization and the application is not location-related.

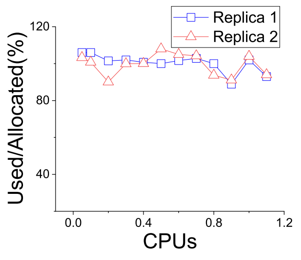
We also test the maximum number of containers on each node, as shown in Fig. 13. KubeHICE can control the number of containers of nodes according to nodes' CPU performance.



(a) Native, Homogeneous



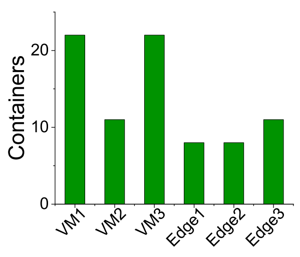
(b) Native, Heterogeneous



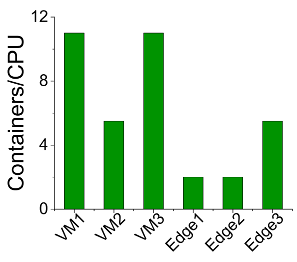
(c) KubeHICE, Heterogeneous

Fig. 12. CPU utilization of two replicas

The number of containers on each CPU is positively correlated with the single-thread performance, which effectively avoids overload caused by allocating too many containers to low-performance nodes and improves the resource utilization of high-performance nodes.



(a) Total containers



(b) Containers on per CPU

Fig. 13. Number of containers

VI. Related work

Containerization at cloud-edge. In recent years, container technologies have been widely used in data centers because of the lower overhead compared with traditional VM [2], [6], [23]. Edge computing is the extension of the cloud, several studies [6], [9], [24], [25] have applied container and container orchestration technologies at the edge. Cloud providers use edge extensions [16], [18] to provide edge computing services in public clouds.

Heterogeneous. Heterogeneity becomes quite common. And it is important to provide heterogeneous ISA aware container orchestration for cloud-edge environments. However, there is still a lack of research on it both in academic and industrial fields at present. Using heterogeneous systems can effectively reduce energy consumption in data centers [26]. Popcorn Linux [8], [27], [28] implements the migration of heterogeneous ISA applications. G E H Ahmed et al. [29]–[32] design scheduling algorithms for tasks that have different execution times or energy efficiencies when using heterogeneous hardware. H Zhao [33] studies the deployment mode of micro-service in the edge heterogeneous environment, calculated the response time of the application placed in the heterogeneous nodes, and then determined the placement problem of the micro-service in the heterogeneous environment by calculating the global minimum response time, and designed the request forwarding strategy to deal with the performance heterogeneity problem of different replicas. Y Mao [34] studies a container placement problem in a heterogeneous cluster for the difference of container resource requirements.

VII. Conclusion

Container orchestrator has become the de facto method for hosting distributed applications in virtualized clouds and edge computing environments. The existing container orchestrators are designed for homogeneous cloud datacenters. They serve as one of the key components for providing the efficiency of container management. However, the hardware heterogeneity makes the existing mechanisms insufficient in cloud-edge platforms in terms of container image ISA matching and deployment, hardware utilization, etc. We propose KubeHICE, a container orchestrator for heterogeneous-ISA architectures by extending Kubernetes, the most popular container orchestrator. KubeHICE supports automatic ISA matching between container images and nodes, And it provides performance-aware scheduling for container deployment. KubeHICE uses a novel CPU resource allocation approach to eliminate the performance of containers on heterogeneous nodes. It schedules containers to nodes with suitable performance. For end users, KubeHICE shields the heterogeneity of CPU, guarantees the performance of applications, and improves CPU utilization with an increase of up to 40%.

Acknowledgment

The work in this paper is supported by the Natural Science Foundation of China (Under Grant NO.: 61872444 and U19A2060) and the National Key Research and Development Program of China (2018YFB1003602).

References

[1] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.
[2] Q. Zhang, X. Zhang, Q. Zhang, W. Shi, and H. Zhong, "Firework: Big data sharing and processing in collaborative edge environment," in *2016 Fourth IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, 2016, pp. 20–25.
[3] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar, and J. Ott, "Consolidate iot edge computing with lightweight virtualization," *IEEE Network*, vol. 32, no. 1, pp. 102–111, 2018.
[4] (2021) Docker. [Online]. Available: https://www.docker.com/
[5] (2021) Kata containers. [Online]. Available: https://katacontainers.io/
[6] L. Yin, J. Luo, and H. Luo, "Tasks scheduling and resource allocation in fog computing based on containers for smart manufacturing," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 10, pp. 4712–4721, 2018.

[7] T. Rausch, A. Rashed, and S. Dustdar, “Optimized container scheduling for data-intensive serverless edge computing,” *Future Generation Computer Systems*, vol. 114, pp. 259–271, 2021.

[8] A. Barbalace, M. L. Karaoui, W. Wang, T. Xing, P. Olivier, and B. Ravindran, “Edge computing: The case for heterogeneous-isa container migration,” in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 73–87.

[9] L. Ma, S. Yi, N. Carter, and Q. Li, “Efficient live migration of edge services leveraging container layered storage,” *IEEE Transactions on Mobile Computing*, vol. 18, no. 9, pp. 2020–2033, 2019.

[10] Y. Guo, Q. Zhuge, J. Hu, M. Qiu, and E. H.-M. Sha, “Optimal data allocation for scratch-pad memory on embedded multi-core systems,” in *2011 International Conference on Parallel Processing*, 2011, pp. 464–471.

[11] M. Qiu, L. Zhang, M. Guo, F. Hu, S. Liu, and E. H.-M. Sha, “Global variable partition with virtually shared scratch pad memory to minimize schedule length,” in *2009 International Conference on Parallel Processing Workshops*, 2009, pp. 478–483.

[12] M. Qiu, Z. Chen, and M. Liu, “Low-power low-latency data allocation for hybrid scratch-pad memory,” *IEEE Embedded Systems Letters*, vol. 6, no. 4, pp. 69–72, 2014.

[13] (2021) Swarm mode overview. [Online]. Available: https://docs.docker.com/engine/swarm/

[14] (2021) Kubernetes. [Online]. Available: https://kubernetes.io

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

[16] (2021) Kubeedge. [Online]. Available: https://kubernetes.io

[17] Y. Xiong, Y. Sun, L. Xing, and Y. Huang, “Extend cloud to edge with kubeedge,” in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, 2018, pp. 373–377.

[18] (2021) Openyurt. [Online]. Available: https://OpenYurt.io

[19] (2021) gvisor. [Online]. Available: https://github.com/google/gvisor

[20] (2021) K3s. [Online]. Available: https://k3s.io/

[21] (2021) runc. [Online]. Available: https://github.com/opencontainers/runc

[22] (2021) Prometheus. [Online]. Available: https://github.com/prometheus/prometheus

[31] D.-K. Kang, G.-B. Choi, S.-H. Kim, I.-S. Hwang, and C.-H. Youn, “Workload-aware resource management for energy efficient heterogeneous docker containers,” in *2016 IEEE Region 10 Conference (TENCON)*, 2016, pp. 2428–2431.

[23] Q. Zhang, L. Liu, C. Pu, Q. Dou, L. Wu, and W. Zhou, “A comparative study of containers and virtual machines in big data environment,” in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 178–185.

[24] S. V. Gogouvitis, H. Mueller, S. Premnadh, A. Seitz, and B. Bruegge, “Seamless computing in industrial systems using container orchestration,” *Future Generation Computer Systems-the International Journal of Escience*, vol. 109, pp. 678–688, 2020.

[25] T. Rausch, A. Rashed, and S. Dustdar, “Optimized container scheduling for data-intensive serverless edge computing,” *Future Generation Computer Systems*, vol. 114, pp. 259–271, 2021.

[26] J. Nider and M. Rapoport, “Cross-isa container migration,” in *Proceedings of the 9th ACM International on Systems and Storage Conference*, ser. SYSTOR ’16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2928275.2933275

[27] M. L. Karaoui, A. Carno, R. Lyerly, S.-H. Kim, P. Olivier, C. Min, and B. Ravindran, “Scheduling hpc workloads on heterogeneous-isa architectures: Poster,” in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 409–410. [Online]. Available: https://doi.org/10.1145/3293883.3295717

[28] (2021) Popcorn linux. [Online]. Available: http://www.popcornlinux.org/index.php

[29] A. Suyyagh and Z. Zilic, “Energy and task-aware partitioning on single-isa clustered heterogeneous processors,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 2, pp. 306–317, 2020.

[30] G. El Haj Ahmed, F. Gil-Castiñeira, and E. Costa-Montenegro, “Kubcg: A dynamic kubernetes scheduler for heterogeneous clusters,” *Software: Practice and Experience*, vol. 51, no. 2, pp. 213–234, 2021.

[32] L. Baresi and G. Quattrocchi, “Cocos: A scalable architecture for containerized heterogeneous systems,” in *2020 IEEE International Conference on Software Architecture (ICSA)*, 2020, pp. 103–113.

[33] H. Zhao, S. Deng, Z. Liu, J. Yin, and S. Dustdar, “Distributed redundancy scheduling for microservice-based applications at the edge,” *IEEE Transactions on Services Computing*, pp. 1–1, 2020.

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