Future Generation Computer Systems 165 (2025) 107569

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

EVRM: Elastic Virtual Resource Management framework for cloud virtual instances

Desheng Wang[a], Yiting Li[a], Weizhe Zhang[a,b,c,*], Zhiji Yu[a], Yu-Chu Tian[d], Keqin Li[e]

[a] School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, Shenzhen, 518055, China
[b] Department of New Networks, Peng Cheng Laboratory, Shenzhen, 518055, China
[c] Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies, Shenzhen, 518055, China
[d] School of Computer Science, Queensland University of Technology, Brisbane, QLD 4000, Australia
[e] Department of Computer Science, State University of New York, New Paltz, NY 12561, USA

ARTICLE INFO

Keywords:
Multiple virtual resources
Elastic resource management
Deep reinforcement learning
OpenStack

ABSTRACT

As cloud demand for computation and network resources fluctuates, effective resource management becomes essential for optimizing allocation and enhancing performance in virtualization-based applications. Current methods struggle to efficiently schedule multiple virtual resources for dynamic workloads. To address this, we propose a self-adaptive elastic virtual resource management (EVRM) framework that comprises a monitor, analyzer, planner, and executor, enabling dynamic scheduling of CPU, memory, and bandwidth for virtual instances. Central to EVRM is a resource management model employing a novel deep reinforcement learning approach, the deep deterministic policy gradient-based resource allocation (DDPG-RA), which coordinates resource allocation by automatically exploring optimization policies and learning complex relationships between resource allocation and performance. Additionally, DDPG-RA features an action refinement algorithm to derive multiple resource allocations from its outputs. Experimental results using OpenStack demonstrate that EVRM significantly enhances performance, achieving approximately 52.87% faster benchmark completion times and a 41.37% reduction in average time under both light and heavy loads, outperforming three competing approaches while optimizing physical resource utilization.

1. Introduction

In the past few years, cloud computing has undergone rapid development to provide computing services [1,2]. This leads to the rejuvenation of virtualization technologies, such as KVM-based virtual machine (VM) and Docker-based container [3]. Virtualization enables virtual instances (i.e., VMs or containers) to run in one compute server using multiplexing hardware resources with robust scalability. This is a way to improve hardware resource utilization while maintaining a reasonable quality of service (QoS). However, the exponential growth of internet traffic and changing cloud resource demands present significant challenges in maintaining consistent cloud service performance [4–6]. For instance, virtual instances are initially allocated to tenants and configured with static resources when booted. However, applications running in virtual instances bring dynamic workload, which has led to unstable service performance and low resource utilization [7,8]. The inefficiency of allocating resources, including CPU, memory, and network bandwidth, causes massive performance degradation [9,10].

To cope with the fluctuations in cloud resource demands, this paper manages the virtual instances by elastic resource management, referring to resizing (scaling-up or scaling-down) the resource amount of virtual instances [11]. Virtual resource management is challenging and responsible for achieving resource optimization of virtual instances to get excellent running performance. This paper focuses on dynamically managing computation resources (CPU, memory), and network resource (bandwidth), in response to the changing workloads of virtual instances on a server. Our work in this study aims to improve the virtual instances' running performance by making full utilization of multiple resources.

Since virtual instances' running performance is usually unavailable with specific quantitative results, most researches [12,13] only consider the resource utilization when allocating virtual resources or collecting system information. However, virtual instances' running performance relates to a few factors in addition to the resource utilization. Therefore, the first challenge of elastic virtual resource management is **CH1:** *How*

* Corresponding author at: School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, Shenzhen, 518055, China.
E-mail addresses: wangdesheng@hit.edu.cn (D. Wang), 190110119@stu.hit.edu.cn (Y. Li), wzzhang@hit.edu.cn (W. Zhang), 1960901627@qq.com (Z. Yu), y.tian@qut.edu.au (Y.-C. Tian), lik@newpaltz.edu (K. Li).

https://doi.org/10.1016/j.future.2024.107569
Received 7 October 2023; Received in revised form 11 October 2024; Accepted 16 October 2024
Available online 19 November 2024

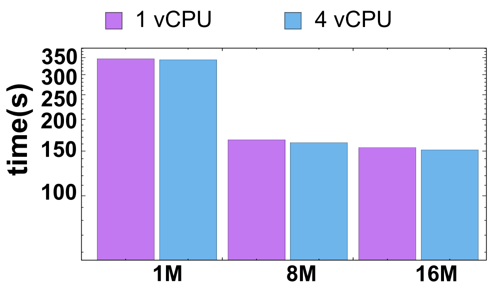


Fig. 1. Completion time under different computation and network resource configurations of virtual instances.

to build a elastic virtual resource management model that optimizes virtual instances' running performance?

Furthermore, numerous resource management methods are proposed to schedule a single type of resources [13,14]. Moreover, coordinating resource schedulers has also been investigated to manage CPU and memory resources together. For example, the Adares [15] is devised to leverage a contextual bandits framework to take hot addition/removal actions for virtual CPU (vCPU) and memory resources of VMware's vSphere VMs. With the advantages of portability and minimal overhead over VMs, containers gain popularity as lightweight virtualization (or containerization) technology [16]. Resource management in container environments possesses limited research, particularly on coordinating multiple resources management. For example, the Thoth [17] achieves automatic CPU and memory scaling for container-based cloud environment via Q-learning method. However, these works fail to well consider that the running performance of applications always relates to multiple types of resources [18], including computation and network resources. For example, the performance of memory-intensive or network-intensive tasks tends to tightly related to CPU resources. Therefore, the second challenge is **CH2:** *How to build the relationship between computation and network resources when we apply a resource allocation?*

Feedback-based resource allocation has been exploited by recent work for its simplicity [19], obtaining the optimal virtual resources allocation for only current system state. However, these allocation solutions can be sub-optimal for the future system states due to the dynamic workload of virtual instances on a server. Furthermore, reinforcement learning (RL) methods are also applied to allocate virtual resources [19]. However, these works set limited action space during resource allocation, ignoring the continuous action space and resulting in limited resource allocation performance. Therefore, the third challenge is **CH3:** *How to obtain resource allocation solution to optimize the adaption of dynamic workload for virtual instances on a server?*

It is evident that solely scaling CPU and memory overlooks the critical aspect of application performance, i.e., bandwidth resource. To substantiate this assertion, we conducted an experiment designed to illustrate the value of scaling bandwidth alongside CPU and memory. In our setup, one VM acts as the client, transmitting images, while another VM serves as the server, executing complex Canny algorithm. During the experiment, we identified bandwidth as the primary bottleneck, rather than CPU or memory limitations. By testing various configurations of CPU for client and bandwidth for both VMs, we demonstrate the necessity of bandwidth scaling alongside CPU and memory optimization. Our findings, detailed in Fig. 1, underscore the importance of holistic resource scaling for optimal application performance.

We configured bandwidth settings of 1M, 8M, and 16M while allocating CPU cores to 1 or 4. Notably, the transition time significantly exceeds the computation time. For instance, transitioning bandwidth from 1M to 8M reduced the total runtime from 350 s to 200 s. Conversely, varying CPU cores from 1 to 4 across different bandwidth configurations resulted in marginal improvements. These results indicate that optimizing bandwidth, particularly when it is the bottleneck, significantly enhances application performance.

Based on **CH1**, **CH2** and **CH3**, we model the relationship between the resource allocation and system performance accurately, in which some resource configuration values are discrete (CPU) while others (memory, bandwidth) are continuous, and it is hard to unify them. Meanwhile, to optimize system performance, we utilize the relationship model to dynamically allocate multiple resources for virtual instances.

Targeting **CH1**, we analyze the key factors that affect the resource utilization and running performance of virtual instances. Furthermore, a resource management approach requires interaction with the cloud environment, where each resource allocation is selected according to the current state and is executed to transition to the next state during the interaction process. Therefore, the resource management problem obeys Markov Decision Process (MDP). We finally construct an MDP model for elastic virtual resource management problem.

Targeting **CH2** and **CH3**, since complex dependency relations and ever-changing workloads are not fully understood to estimate the resource demands, we apply deep reinforcement learning (DRL) to learn the optimization policy to allocate multiple virtual resources and adapt its decision during virtual instances running on a server. The learned model can maximize the potential reward, i.e., the overall optimization of resource utilization and virtual instance running performance. Specifically, we propose a novel deep deterministic policy gradient based resource allocation (DDPG-RA) method, to coordinate CPU, memory, and bandwidth resources of virtual instances on a server. DDPG-RA employs deep deterministic policy gradient (DDPG) [20] to automatically explore the optimization policy during the training process, learning the subtle relationships between multiple virtual resources allocation, resource utilization, and virtual instance running performance. Moreover, our DDPG-RA cooperates with a proposed action refinement algorithm to obtain multiple virtual resources allocation from DDPG output. Finally, we present a self-adaptive resource management (EVRM) system that dynamically schedules CPU, memory, and bandwidth resources in VM and container environments, applying DDPG-RA with learned knowledge to manage multiple virtual resources allocation to match the changing resource demands.

The contributions of this paper include:

- We build a novel model for elastic virtual resource management, indicating the key factors that affect the resource utilization and running performance of virtual instances. To our best knowledge, we are the first to investigate the coordinating CPU, memory, and bandwidth.
- We propose a self-adaptive resource management algorithm, called DDPG-RA, which firstly employs a DDPG model and a proposed action refinement algorithm to dynamically and quantitatively calculate target resource allocation with maximum potential reward.
- We develop a self-adaptive resource management framework EVRM, applying DDPG-RA with learned knowledge to adapt to the dynamic changes of workload, with improving the resource utilization and virtual instance running performance as a goal.

We evaluate our elastic virtual resource management approach on real-world Nova and Nova-docker services in OpenStack and conduct multiple tests to evaluate our framework. The evaluation results verified that with efficient use of physical resources, the proposed approach improves the performance of virtual instances by shortening the completion time of various benchmarks in the validation of VMs/containers. Compared to the existing baselines, EVRM reduces the completion time by about 52.87% on average during light-loaded evaluation and about 41.37% during heavy-loaded evaluation.

The remainder of this paper is organized as follows: Section 2 reviews the related work. Section 3 gives the system overview of our EVRM. Section 4 introduces our resource management algorithm in detail. Experiments are conducted in Section 5. Finally, Section 6 concludes the paper.

2. Related work

Many studies have explored elastic resource management for cloud efficiency. We can classify these works according to the resource types, including memory, CPU, bandwidth, and coordinating cases.

Memory management: Research in [21] proposed an adaptive Global-scheduling algorithm to control memory. Lahmann et al. [22] investigated memory allocation in container applications, aiming to scheduling memory utilization in containers, and presented the differences between VM-based and container-based resource allocation. Nicodemus et al. [23] presented a elastic container memory allocation method to satisfy the changing QoS requirements. However, it only considers current state. Melissaris et al. [24] presented the elastic cloud services of cloud data platform, Snowflake, which scales memory when queries require significantly more memory than CPU. However, it ignores the joint scaling of other virtual resources.

CPU management: Makridis et al. [14] presented adaptive CPU resource provisioning schemes by configuring timeslice and ratelimit values in the Xen Credit scheduler. Jang et al. [25] provided Boost-scheduling scheme for VMs to lead boost credit via the evaluation function, achieving automatic boost frequency management. In addition, Wang et al. [26] and Wang et al. [27] designed predictive methods based on machine learning to decrease CPU slack while avoiding CPU insufficiency for various CPU workloads. Zhao et al. [28] investigated tiny autoscalers to dynamically allocate CPU for short-running serverless workloads in Kubernetes. However, these methods ignore the utilization relationships between multiple resources.

Bandwidth management: Karmakar et al. [29] devised heuristic and integer linear programming algorithms for bandwidth allocation. Chen et al. [30] modeled the bandwidth allocation optimization problem and proposed a distributed bandwidth allocation scheme based on the extension of fairness mechanism and dual decomposition to improve network performance while improving bandwidth utilization. Li et al. [31] implemented a coordinating bandwidth control architecture, which considers bandwidth resource as sharing resource via group, provides multi-dimensional bandwidth sharing scheme for inter-framework, inter- and intra-application, and develops a path selection method based on network dependency. Zeng et al. [32] devised an SR-IOV-based network I/O allocation framework, named Raccoon, for a workload-aware VM scheduling algorithm to facilitate hybrid I/O workloads in virtual environments. Based on this, Ye et al. [13] implemented an autonomic framework Sova, coordinating DSR-IOV and live migration to improve the VMs QoS.

Hybrid resource management: Adares leverages a contextual bandits framework to control the CPU and memory adaptations for VMs [15]. Moreover, research in [12] presented hybrid CPU and network resource allocation schemes to improve network-intensive applications' performance. In addition, Sangpetch et al. [17] devised Thoth, a dynamic Q-learning resource management system, to adjust appropriate resources for the container-based cloud platform. Google [33] devised Autopilot, which employs machine learning and a set of finely-tuned heuristics, to adjust vertical scaling for CPU/memory resources. Prakash et al. [34] focused on container memory and CPU control in the derivative cloud, where containers are nested in VMs. They considered that VMs' resource adjustment (including memory and CPU) could significantly degrade the performance of containers running in VMs, thereby developing a policy-driven method to reduce the impact of the VM resource management process on these containers. Chouliaras et al. [35] devised PACE to automatically allocate container CPU and memory resources based on threshold-based rules, CNN and K-means approaches. While Jeong et al. [4] presented HiPerRM to auto-scale resource based on the forecasted pod's CPU and memory usage. Sheganaku et al. [11] and Vu et al. [36] proposed autoscaling approaches (including vertical and horizontal scaling) to optimize container resource allocation. Before allocating resources to virtual instances, each work should analyze the relationship between the running performance and the resource allocation.

Efforts have been made to improve elastic virtual resource management. However, these methods ignore the combination of computation and network resources, named CPU, memory, and bandwidth. Furthermore, most research fails to optimize the adaption of dynamic workload for virtual instances on a server. This study takes advantage of OpenStack architecture to improve elastic resource management. Furthermore, we apply DRL to interact with complex environment dynamically and obtain learned agent to improve the resource utilization and virtual instance running performance.

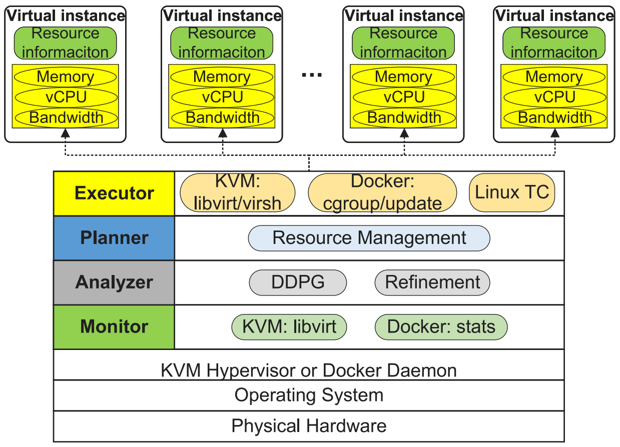


Fig. 2. Elastic virtual resource management framework.

3. System overview

This section introduces the system architecture of the proposed elastic resource management framework, named EVRM, for virtual instances in KVM and Docker (as shown in Fig. 2). To build this framework, we have adopted the MAPE architecture [37]. EVRM operates as an event loop system comprising four main modules deployed within the host environment: the monitor module, the analyzer module, the planner module, and the executor module. (i) The monitor collects multiple resource information of each virtual instance, including CPU, memory, and network. We explore several methods to obtain resource information during our research. (ii) Analyzer module retrieves the resource information of all virtual instances from the monitor module periodically. (iii) Planner makes a global decision for the executor module to automatically schedule multiple resources for a plurality of virtual instances with our proposed resource management algorithm (introduced in Section 4). (iv) Finally, executor provisions target resource for each virtual instance.

3.1. Monitor module

The EVRM needs to obtain the CPU, memory, and network information of each virtual instance for the analyzer module in this work. Table 1 presents the resource status information for both KVM and Docker instances.

We adopt the `libvirt` toolkit to implement an intermediate VM monitor, profiling CPU and memory information through `virDomain.info()` and `virDomain.MemoryStats()`, respectively. The former presents the CPU time used in nanoseconds (ns) in the fourth return parameter. Similarly, we obtain the free memory size of each VM. In addition, we can obtain the communication packet information through observing `/proc/net/dev`.

These raw information needs to be transformed to the input of our analyzer module to serve our proposed resource management

Table 1
Resource information of each virtual instance.

Type	Raw data (Returned from Monitor)	Extracted data (Input of Analyzer)
KVM	$cpu_i(t)$: CPU used time at t of v_i	uc_i: Eq. (1)
	$f_i(t)$: free memory size at t of v_i	um_i: Eq. (1)
	$recv_i(t)$: received packet size at t of v_i; $trans_i(t)$: transmitted packet size at t of v_i	ub_i: Eq. (1)
Docker	uc_i: CPU utilization of v_i	uc_i
	um_i: memory utilization of v_i	um_i
	$recv_i(t)$; $trans_i(t)$	ub_i: Eq. (1)

algorithm. For the ith virtual instance v_i, we obtain its CPU usage $uc_i(t)$, memory usage $um_i(t)$, and bandwidth usage $ub_i(t)$, respectively, as follows:

$$\begin{cases} uc_i(t) = \dfrac{cpu_i(t_2) - cpu_i(t_1)}{(t_2 - t_1) \cdot c_i(t)}, \; um_i(t) = \dfrac{m_i(t) - f_i(t)}{m_i(t)}, \\ ub_i(t) = \dfrac{recv_i(t_2) + trans_i(t_2) - recv_i(t_1) - trans_i(t_1)}{(t_2 - t_1) \cdot b_i(t)}, \end{cases} \tag{1}$$

where $m_i(t)$, $c_i(t)$, $b_i(t)$ are memory, CPU, bandwidth configuration of v_i at time t, respectively.

Docker containers' resource information, including CPU, memory, and network, are profiled through `docker stats` command, which records the values of the CPU and memory parameters under `cgroup` file system through Docker daemon.

3.2. Analyzer and planner modules

As shown in Fig. 2, the analyzer module obtains the resource information of all virtual instances from the monitor module periodically, and planner makes a global decision for the executor module to automatically schedule resource between a plurality of instances. To maintain the synchronization between the monitor and analyzer modules, the EVRM periodically monitors and schedules resources. The planner module uses our proposed resource scheduling algorithm, named DDPG-RA, to calculate target resource allocation for each instance according to the current workload in each cycle. The DDPG-RA is the core of the analyzer module, and we will introduce it in Section 4.

3.3. Executor module

After obtaining these target resource configuration values from the Planner module, EVRM executes the resource allocation through the executor module, which we introduce in this section. Two common parameters in KVM that can be adjusted for CPU are the maximum number of CPUs and the live number of CPUs. This same principle applies to memory allocation. By adjusting the live number of CPUs and ensuring it remains within the bounds of the maximum number, we can prevent downtime when modifying allocated resources.

KVM Executor: KVM hypervisor has a kernel module called kvm.ko, which manages virtual CPU and memory. The interface is provided by `libvirt`, whereas the memory allocation function `virDomainSetMemoryFlags()` and CPU allocation function `virDomainSetVcpus()` can dynamically change the target amount of memory and vCPU allocated to each VM, respectively. We can also use the `virsh` management tool to control memory and CPU allocation. As with some bandwidth control schemes [38], we control the bandwidth for each vNIC, i.e., limiting the maximum upload (download) bandwidth, via Linux TC tool.

Docker Executor: Our EVRM applies the original `docker update` configuration tool to control memory and CPU allocation by parameters `cpus` and `memory`. Finally, similar to VMs, we adopt the Linux TC tool to allocate bandwidth resources for containers.

Table 2
Summary of key notations.

Notation	Description
n	The number of running virtual instances
v_i	The ith virtual instance
$m_i(t)$	The memory configuration of v_i at time t
$nm_i(t)$	The memory normalization of v_i at time t
$um_i(t)$	The memory utilization of v_i at time t
$\bar{m}_i$	The maximum memory configuration of v_i
$\underline{m}_i$	The minimum memory configuration of v_i
$c_i(t)$	The vCPU configuration of v_i at time t
$nc_i(t)$	the vCPU normalization of v_i at time t
$uc_i(t)$	The vCPU utilization of v_i at time t
$\bar{c}_i$	The maximum vCPU configuration of v_i
$\underline{c}_i$	The minimum vCPU configuration of v_i
$b_i(t)$	The bandwidth configuration of v_i at time t
$nb_i(t)$	The bandwidth normalization of v_i at time t
$ub_i(t)$	The bandwidth utilization of v_i at time t
$\bar{b}_i$	The maximum bandwidth configuration of v_i
$\underline{b}_i$	The minimum memory configuration of v_i
$OU(t)$	The number of over- and under-loaded virtual instances at time t
$VA(t)$	The sum of the variances of resource utilization at time t
$EV(t)$	Evaluation value of each resource management operation at time t
HC	The total vCPU amount of host
HM	The total virtual memory amount of host

4. Elastic virtual resource management algorithm

Although oversubscription resources of virtual instances have been regular practice, oversubscription contention interference among running instances cannot be avoided entirely, which results in unpredictable service times [31,34,39]. To cope with the abrupt workloads, we propose a DDPG-based resource allocation algorithm, named DDPG-RA, to calculate target resource allocation for each virtual instance according to the current resource information. Our DDPG-RA employs DRL model DDPG to automatically explore the optimization policy during the training process, learning the complex relations between multiple virtual resources allocation, resource utilization, and virtual instance running performance. Due to the complexity of multiple virtual resource management, DDPG-RA includes an action refinement algorithm to convert the ratio-based output into target resource allocation size.

4.1. Problem formulation

We establish a resource management model to manage virtual instances' CPU, memory, and bandwidth resources in a physical server. Table 2 shows the key notations we use in our algorithm.

We define $V = \{v_1, v_2, \dots, v_n\}$ to represent a set of all virtual instances, where n is the number of virtual instances. We define $M = \{m_1(t), m_2(t), \dots, m_n(t)\}$ to represent the memory configuration sizes of each virtual instance at time t. Moreover, we set that $UM = \{um_1(t), um_2(t), \dots, um_n(t)\}$ is the memory utilization rates of each virtual instance at time t, and the value of $um_i(t)$ is between $[0, 1]$. Let HM be the total virtual memory amount of host.

This paper applies the number of vCPUs as a quantitative indicator. We define $C = \{c_1(t), c_2(t), \dots, c_n(t)\}$ to represent the vCPU amount of each virtual instance at time t. At the same time, we define $UC = \{uc_1(t), uc_2(t), \dots, uc_n(t)\}$ as the virtual machine CPU utilization at time t, and the value of $uc_i(t)$ is between $[0, 1]$. We define HC as the total vCPU amount of host.

In order to satisfy that tenants can also enjoy higher network service quality during traffic peak periods, cloud service providers will also provide a unique bandwidth plan to allocate part of the physical bandwidth directly to the virtual instances. We define $B =$

$\{b_1(t), b_2(t), \ldots, b_n(t)\}$ and $UB = \{ub_1(t), ub_{2(t)}, \ldots, ub_n(t)\}$ as the bandwidth upper threshold and occupancy rate of each virtual instance at time t, and the value of $ub_i(t)$ is between $[0, 1]$.

We define resources' maximum and minimum amount as $\bar{m}_i$, $\bar{c}_i$, $\bar{b}_i$, $\underline{m}_i$, $\underline{c}_i$, $\underline{b}_i$, respectively. In order to avoid diversities among multiple resources, we normalize the resource configuration of virtual memory $m_i(t)$, CPU $c_i(t)$, and bandwidth $b_i(t)$ into $nm_i(t)$, $nc_i(t)$, $nb_i(t)$ for v_i, as follows:

$$\begin{cases} nm_i(t) = m_i(t)/\bar{m}_i, \\ nc_i(t) = c_i(t)/\bar{c}_i, nb_i(t) = b_i(t)/\bar{b}_i. \end{cases} \tag{2}$$

This study aims to improve the running performance of applications by fully utilizing the resources. We analyze the key factors that affect the resource utilization and running performance, and define the following objective function.

With our framework's goal to improve the virtual instances' running performance by making full utilization of multiple resources, each virtual instance should release its under-utilized resources and gain its over-utilized resources. Here we define two thresholds for each resource. For memory utilization, we define ξ_m^{low} as the lower threshold of memory utilization. When it is lower than this threshold, memory utilization is under-utilized, and its memory resource should be released. At the same time, we define ξ_m^{high} as the upper threshold of memory utilization. When a virtual instance utilizes more memory than ξ_m^{high}, the virtual instance suffers from performance degradation due to overloaded memory utilization, and should increase memory size. Similarly, we define two sets of thresholds for CPU and bandwidth utilization, ξ_c^{low} and ξ_c^{high}, ξ_b^{low} and ξ_b^{high}, respectively. Each virtual instance should avoid over-utilized and under-utilized states considering the corresponding upper and lower thresholds. Therefore, we define $OU(t)$ as the number of virtual instances in a lousy state at time t, including an over-utilized state and an under-utilized state:

$$OU(t) = \sum_{i=1}^{n} x_i, \tag{3}$$

where $x_i = 0$ when it satisfies $uc_i \in \left[\xi_c^{low}, \xi_c^{high}\right]$, $um_i \in \left[\xi_m^{low}, \xi_m^{high}\right]$, and $ub_i \in \left[\xi_b^{low}, \xi_b^{high}\right]$. Otherwise, $x_i = 1$.

Furthermore, it is unreasonable that some virtual instances possess over-utilized resource utilization while some are light-loaded. Therefore, a better resource allocation choice is to obtain a balanced resource utilization among virtual instances. We define $VA(t)$ as the sum of the variances of the three resource utilization at time t:

$$VA(t) = \gamma_m \cdot var(UM) + \gamma_c \cdot var(UC) + \gamma_b \cdot var(UB), \tag{4}$$

where $var(\cdot)$ is the function for calculating the variance, and γ_m, γ_c, and γ_b are the importance coefficients respectively, set by uses. The smaller the $VA(t)$, the smaller the difference in resource utilization of each virtual instance.

Additionally, we define $EV(t)$ as the evaluation value of each resource management operation at time t; enormous $EV(t)$ value means excellent performance. In this case, we evaluate the resource scheduling for each virtual instance and define the evaluation value of v_i at time t as $e_i(t)$, $EV(t)$ as follows:

$$EV(t) = \sum_{i=1}^{n} e_i(t) = \sum_{i=1}^{n} (e_i^c(t) + e_i^m(t) + e_i^b(t)). \tag{5}$$

For the calculation of $e_i(t)$, we evaluate the effects of scheduling each resource (including CPU, memory, and bandwidth), and obtain the corresponding evaluation values, defined as $e_i^c(t)$, $e_i^m(t)$, and $e_i^b(t)$, and $e_i(t) = e_i^c(t) + e_i^m(t) + e_i^b(t)$. Based on variations in the resource utilization status, we obtain $e_i^c(t)$, $e_i^m(t)$, and $e_i^b(t)$ through the rules presented in Fig. 3 where the normalized evaluation values (i.e., from a_1 to a_6) are initialized from environmental parameters. For example, as the arrows numbered (2) and (5), the evaluation values are a_2 and a_5, under the

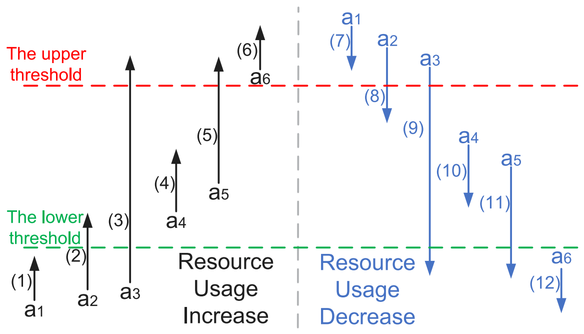


Fig. 3. Rules for evaluation values.

cases that the resource usage change from under-loaded to the normal state and from the normal to over-loaded state.

In order to improve the virtual instances' running performance, our framework dynamically allocates multiple virtual resources to minimize the $OU(t)$ and $VA(t)$ and maximize the $EV(t)$. Thus, the objective function is formulated to minimize H, which is expressed as a weighted combination of the three parts during the whole time:

$$\min H = \int \left(\lambda_1 \cdot OU(t) + \lambda_2 \cdot VA(t) - \lambda_3 \cdot EV(t)\right) dt, \tag{6}$$

where λ_1, λ_2 and λ_3 are the importance coefficients, initialized from environmental parameters.

Eq. (6) outlines the impact of scheduling resources effectively. To achieve this, we developed and trained a DDPG model. In each iteration, the DDPG model determines the allocation of resources for the subsequent iteration. Using parameters a_1 to a_6, Eq. (6) assesses the resource utilization state post-scheduling for the current iteration.

In our model, $OU(t)$, $VA(t)$ and $EV(t)$ represent the number of instances in a bad state, the sum of variances in CPU, memory, and bandwidth utilization, and the effects of scheduling each resource, respectively. The parameters λ_1, λ_2, and λ_3 denote the significance of $OU(t)$, $VA(t)$ and $EV(t)$ in the optimization objective. Increasing λ_1 is recommended if the objective is to ensure all virtual instances run in a good state, while boosting λ_2 is suggested for maintaining resource usage balance. Similarly, λ_3 serves similar purposes for other aspects of resource management. Regarding the normalized evaluation values a_1 to a_6 initialized from environmental parameters, these values determine the rewards associated with different performance levels, indicating a preference to avoid exceeding upper or lower resource usage thresholds.

Due to the synchronous execution of multiple applications in the realistic system environment, complex dependency relations and ever-changing resource requirements make it challenging to outline the resource demands [33], which is an uncertain optimization problem. Therefore, this study employs DRL to develop a dynamic resource allocation approach. The DRL works by obtaining strategy improvement through continuous interactions with the changing environment in discrete time steps.

4.2. Markov decision process model

During the MDP process, the agent chooses to perform a resource allocation action a_t with maximum potential reward according to the current state s_t, obtains an immediate reward r_t after performing the action, and enters a new state s_{t+1}.

To make effective decisions to allocate memory, CPU, and bandwidth, the state set contains the resource information of each virtual instance v_i, including the amount of the three configuration resources $\{m_i(t), c_i(t), b_i(t)\}$ and the corresponding utilization rates of the three resources $\{um_i, uc_i, ub_i\}$. Thus, we define the state $s_t \in S_t$ at time t as:

$s_t = \{s_t^i | i \in [1, n]\}, s_t^i = \{nm_i(t),$

$$um_i(t), nc_i(t), uc_i(t), nb_i(t), ub_i(t)\} , \tag{7}$$

where s_t^i is the state for virtual instance v_i at time t. The $nm_i(t)$, $nc_i(t)$, and $nb_i(t)$ are the normalization values of memory, CPU, and bandwidth configuration (as shown in Eq. (2)). The configuration information is obtained from monitor module.

The action includes operations on the three resources. Therefore, we define the action at t, $a_t \in A_t$ as:

$$a_t = \left\{a_t^i | i \in [1, n]\right\}, a_t^i = \left\{a_m^i(t), a_c^i(t), a_b^i(t)\right\}, \tag{8}$$

where a_t^i is the ratio-based action result for virtual instance v_i at time t. The $a_m^i(t)$, $a_c^i(t)$, and $a_b^i(t)$ are the ratio values to schedule memory, CPU, and bandwidth of each virtual instance v_i.

DRL aims to get the maximum cumulative reward after performing an action. In this study, the objective function is to minimize H in Eq. (6). Therefore, the immediate reward function is set to negative sum of the immediate values in Eqs. (3), (4), and (5), so that the maximum cumulative reward represents the minimal H, as shown below:

$$r_t = -\left(\lambda_1 \cdot OU(t) + \lambda_2 \cdot VA(t) - \lambda_3 \cdot EV(t)\right). \tag{9}$$

4.3. DDPG-based resource allocation algorithm

This study intends to improve the virtual instances' running performance by making full utilization of multiple resources. We apply DRL to learn the optimization policy which can maximize the potential reward. Since the system state space and resource allocation action space are continuous in our elastic virtual resource management problem, an integrated value-function- and policy-based DRL approach naturally satisfies our problem. We propose a novel Resource Allocation algorithm based on DDPG method [20], named DDPG-RA, to adapt decision in the continuous state and action space of the elastic virtual resource management problem (Algorithm 1). The DDPG-RA aims to learn a model that maximizes the potential reward, i.e., the overall optimization of virtual instance running performance. Our DDPG-RA employs the DDPG model, including the Actor and Critic networks, uses the Actor and the Critic networks to generate an action and guide the Actor network to produce better actions, respectively. (i) Primary Actor network, defined as $\pi_\theta(S)$, takes the current state s_t as input and obtains action a_t; (ii) Primary Critic network, defined as $Q_\omega(S, A)$, calculates the potential reward value according to s_t, a_t; (iii) Target Actor network (defined as $\pi'_{\theta'}(S)$) and (iv) Target Critic network (defined as $Q'_{\omega'}(S, A)$), copy from the $\pi_\theta(S)$ and $Q_\omega(S, A)$ at a certain period.

The primary Actor network is a deterministic strategy instead of a predictive strategy. For the same system state, the action generated by the primary Actor network should make the potential reward value predicted by the primary Critic network as great as possible. Specifically, the primary Actor network's loss function requires obtaining the maximum potential reward value of the primary Critic network. Therefore, to minimize the strategy loss function, we reverse the primary Critic network's output as the loss function:

$$J(\theta) = -E\left[Q_\omega(s, a)\, |_{s=s_t, a=\pi_\theta(s)}\right]. \tag{10}$$

where $Q_\omega(s, a)$ is an action-value function in Critic network, and the $Q_\omega\left(s_t, a_t\right)$ is expressed as follows:

$$Q_\omega\left(s_t, a_t\right) = E\left[r_t + \gamma \cdot Q_\omega\left(s_{t+1}, a_{t+1}\right)\right], \tag{11}$$

where $\gamma \in [1, 0]$ represents the discount rate.

With a random mini-batch sample $\langle s_t, a_t, s_{t+1}, r_t\rangle (t \in \{1, \dots, X\})$ from replay memory, the parameter θ in the primary Actor network is updated through the sampled policy gradient as:

$$\nabla_\theta J(\theta) = -E\left[\nabla_a Q_\omega(s, a)\, |_{s=s_t, a=\pi_\theta(s)} \nabla_\theta \pi_\theta(s)\, |_{s=s_t}\right], \tag{12}$$

Specifically, at each training step, θ is updated as follows:

$$\theta = \theta - \frac{\alpha_\pi}{X} \sum_{t=1}^{X} \left[\nabla_a Q_\omega(s, a)\, |_{s=s_t, a=\pi_\theta(s)} \nabla_\theta \pi_\theta(s)\, |_{s=s_t}\right], \tag{13}$$

where α_π is the primary Actor network's learning rate.

As for the primary Critic network's training process, we define the primary Critic network's loss function with mean square error function:

$$J(\omega) = E\left[\left(y_t - Q_\omega\left(s_t, a_t\right)\right)^2\right], \tag{14}$$

where the target value y_t as follows:

$$y_t = r_t + \gamma \cdot Q'_{\omega'}(s_{t+1}, \pi'_{\theta'}(s_{t+1})), \tag{15}$$

With a random mini-batch sample $\langle s_t, a_t, r_t, s_{t+1}\rangle, t \in \{1, \dots, X\}$ from replay memory, the parameter ω in the primary Critic network is updated through gradient:

$$\nabla_\omega J(\omega) = E\left[2\left(y_t - Q_\omega\left(s_t, a_t\right)\right) \nabla_\omega Q_\omega\left(s_t, a_t\right)\right]. \tag{16}$$

The ω is training as follows:

$$\omega = \omega - \frac{\alpha_Q}{X} \cdot \sum_{t=1}^{X} \left[2\left(y_t - Q_\omega\left(s_t, a_t\right)\right) \nabla_\omega Q_\omega\left(s_t, a_t\right)\right], \tag{17}$$

where α_Q is the primary Critic network's learning rate.

Finally, at each episode, the target networks' parameters are updated as follows:

$$w' = \tau \cdot w + (1 - \tau) \cdot w', \theta' = \tau \cdot \theta + (1 - \tau) \cdot \theta', \tag{18}$$

where τ is a momentum factor.

Algorithm 1: DDPG-RA

```
1  Initialize the primary Actor network $\pi_\theta(S)$ and the primary
     Critic network $Q_\omega(S, A)$;
2  Initialize the target Actor network $\pi'_{\theta'}(S)$ and the target Critic
     network $Q'_{\omega'}(S, A)$;
3  Create the replay memory and clear it;
4  for each episode do
5      Initialize the environment and gather state information;
6      for each time $t$ do
7          $a_t \leftarrow \pi_\theta(s_t) + \mathcal{N}$;
8          $ra_t \leftarrow refinement(s_t, a_t)$; //Algorithm 2
9          Execute action $ra_t$ through executor in Section 3.3;
10         Observe the next state $s_{t+1}$ and calculate reward in
             Eq. (9);
11         Store the tuple $< s_t, a_t, s_{t+1}, r_t >$ in the replay memory;
12         if $OU(t) = 0$ then
13             break;
14     Sample mini-batch from the replay memory;
15     Update primary Critic and Actor networks based on
         Eqs. (16) and (12), respectively;
16     Update the target network parameters based on Eq. (18);
```

In each episode of Algorithm 1, DDPG-RA gets the current resource state of the virtual instance (line 5), and the primary Actor network generates the corresponding action (line 7). Since the output of Actor network using `tanh` as the activation function ranges in $[-1, 1]$, which cannot directly reflect the meaning of actions, a ratio-based action refinement algorithm is designed in Section 4.4 (line 8). Through action refinement, allocated/reclaimed resources can be directly obtained, and our EVRM executes the action (line 9). Next, we obtain the resource state at time $t+1$, calculate the reward from Eq. (9) (line 10), and store $\langle s_t, a_t, s_{t+1}, r_t\rangle$ in the replay memory (line 11). When there is no lousy state virtual instance (line 12), DDPG-RA jumps out of this episode (line 13). Finally, to learn the complex relations between empirical data and non-stationary distribution, we adopt the replay memory method to randomly sample from the previous state transition experience for training (lines 14–16). In this case, the empirical data can be fully utilized, and the correlation of continuous samples can also be reduced, avoiding significant variance during parameter updating.

In order to provide random explorations to improve the learning coverage during the training process, we add an extra noise $\mathcal{N}$ to a_t in line 7. Thereby, the equation of action a_t that ultimately interacts with the environment is as follows:

$$a_t = \pi_\theta(s_t) + \mathcal{N}, \tag{19}$$

where $\mathcal{N}$ possesses a normal distribution with mean μ and variance σ^2, written as $\mathcal{N} \sim N\left(\mu, \sigma^2\right)$. We initially set $\mu = 0$ and $\sigma = 0.8$. With the number of episodes increasing, the σ minus 0.05 happens every 100 episodes until $\sigma = 0.2$.

4.4. Action refinement

In the last section, the proposed framework uses the Actor network to directly output the ratio-based action result a_t, an action refinement algorithm is proposed to refine the ratio-based action into refined action, denoted as ra_t. The information of ra_t contains the specific size of configuration resources of each virtual instance. Then the host allocates virtual resources for each virtual instance according to ra_t:

$$ra_t = \left\{ra_t^i | i \in [1,n]\right\}, ra_t^i = \left\{ra_m^i(t), ra_c^i(t), ra_b^i(t)\right\}, \tag{20}$$

where ra_t^i is the configuration action for virtual instance v_i at time t. The $ra_m^i(t)$, $ra_c^i(t)$, and $ra_b^i(t)$ are the configuration values to configure the memory, CPU, and bandwidth of each virtual instance v_i, respectively.

During action refinement, the actions are continuous values in $[-1,1]$, whereas the negative and positive values are considered to reduce and increase resources, respectively. The refinement process is presented in Algorithm 2.

4.4.1. Memory refinement

For the memory action $a_m^i(t) \in [-1,0)$, we transform it as $1 + a_m^i(t)$ and then reclaim the memory of each virtual instance v_i by $m_i \cdot \left(1 + a_m^i(t)\right)$, which means $ra_m^i(t) = -m_i \cdot a_m^i(t)$ (line 4). For the memory action $a_m^i(t) \in [0,1]$, we allocate the reclaimed memory to these virtual instances as per their action values. First, we calculate the amount of available memory size M (line 5), which will be assigned to instances with positive memory action values $ratio$ (line 7). Then, our EVRM increases instance memory according to the ratio between their action values. Specifically, our EVRM configures the memory size by $ra_m^i(t) = m_i + M \times a_m^i(t)/ratio$ for each instance v_i (line 10).

4.4.2. CPU refinement

The vCPUs get executed on physical cores as tasks and their number can exceed the number of physical cores. However, multiprocessors possess unpredictable architectures by adding interference and communication delays between different tasks executed on different cores [40]. Without considering the task scheduling strategy on multiprocessors, each vCPU cannot obtain a fixed timeslice on physical cores to execute tasks.

Therefore, the CPU resource is difficult to quantify, and CPU scheduling is generally minor. We adopt the fine-grained scheduling of one vCPU. Specifically, we suggest to separate the continuous action values in $[-1,1]$ into values in $[-1,-\epsilon)$, $[-\epsilon,\epsilon]$, and $(\epsilon,1]$. More specifically, when the CPU action value $a_c^i(t)$ ranges in $[-1,-\epsilon)$, $[-\epsilon]$, and $(\epsilon,1]$, our EVRM deals with virtual instance v_i through decreasing by one vCPU, remaining unchanged, and increasing by one vCPU, which corresponds to $ra_c^i(t) = c_i - 1$, $ra_c^i(t) = c_i$, and $ra_c^i(t) = c_i + 1$, respectively (lines 12–17).

Algorithm 2: Refinement

```
   Input : state $s_t$, action $a_t$;
   Output: the refined action $ra_t$;
1  Initialize $M$ as the remaining available host memory, $ratio \leftarrow 0$;
2  for each $a_t^i \in a_t$ do
3      if $-1 \le a_m^i(t) < 0$ then
4          $ra_m^i(t) \leftarrow -m_i(t) \cdot a_m^i(t)$;
5          $M \leftarrow M + m_i(t) \cdot \left(1 + a_m^i(t)\right)$;
6      else
7          $ratio \leftarrow ratio + a_m^i(t)$;
8  for each $a_t^i \in a_t$ do
9      if $0 \le a_m^i(t) \le 1$ then
10         $ra_m^i(t) \leftarrow m_i(t) + M \cdot a_m^i(t)/ratio$;
11 for each $a_t^i \in a_t$ do
12     if $-1 \le a_c^i(t) < -\epsilon$ then
13         $ra_c^i(t) \leftarrow c_i(t) - 1$;
14     else if $-\epsilon \le a_c^i(t) \le \epsilon$ then
15         $ra_c^i(t) \leftarrow c_i(t)$;
16     else
17         $ra_c^i(t) \leftarrow c_i(t) + 1$;
18     $ra_b^i(t) \leftarrow b_i(t) \cdot \left(1 + a_b^i(t)\right)$;
19     if $ra_c^i(t) < \underline{c}_i$ (OR $ra_c^i(t) > \bar{c}_i$) then
20         $ra_c^i(t) \leftarrow \underline{c}_i$ (OR $ra_c^i(t) \leftarrow \bar{c}_i$);
21     if $ra_m^i(t) < \underline{m}_i$ (OR $ra_m^i(t) > \bar{m}_i$) then
22         $ra_m^i(t) \leftarrow \underline{m}_i$ (OR $ra_m^i(t) \leftarrow \bar{m}_i$);
23     if $ra_b^i(t) < \underline{b}_i$ (OR $ra_b^i(t) > \bar{b}_i$) then
24         $ra_b^i(t) \leftarrow \underline{b}_i$ (OR $ra_b^i(t) \leftarrow \bar{b}_i$);
25 if $\sum_{i=1}^n c_i(t) < HC$ OR $\sum_{i=1}^n m_i(t) < HM$ then
26     return $ra_t$;
```

4.4.3. Bandwidth refinement

As we described in Section 3.3, we allocate the bandwidth by limiting the maximum upload and download bandwidth of each vNIC. Here, we configure the same for both upload and download bandwidth and transform the bandwidth action value as $ra_b^i(t) = b_i \cdot \left(1 + a_b^i(t)\right)$ (line 18).

4.4.4. Action filtering

To ensure the regular operation of each virtual instance, we set the minimum amount of resource configuration for the memory, CPU, and bandwidth of each v_i, denoted as $\underline{m}_i, \underline{c}_i, \underline{b}_i$. During the scheduling process, each virtual instance cannot be lower than these minimum resources. Before performing action, it is necessary to determine how many resources each virtual instance obtains after executing it. Furthermore, if it is lower than the minimum resource or exceeds the maximum resource configuration, the operation is invalid, and we choose the minimum or maximum amount of resource instead (lines 19–24). Finally, we obey the available virtual resource constraints (lines 25–26).

5. Evaluations

We use several hosts to build an Ocata-version OpenStack platform to evaluate our EVRM. Each host possesses the Intel(R) Xeon(R) Gold 6226R CPU @ 2.90 GHz with 64 cores, with a 512 GB memory and disk size of 878 GB. The physical host possesses close to 10000 Mbps of bandwidth resource. Its OS is *Ubuntu 22.04.4 LTS 64*. We mainly use a nova (KVM) and a nova-docker (Docker) compute nodes in the OpenStack. We create a plurality of virtual instances in KVM and Docker nodes to assess multiple types of resource management,

Table 3
Initial configurations of VMs and containers.

Resource		Configuration
Memory:	Memory configuration m_i	2 GB
	Maximum memory size $\bar{m}_i$	8 GB
	Minimum memory size $\underline{m}_i$	512 MB
CPU:	CPU configuration c_i	2 vCPUs
	Maximum CPU size $\bar{c}_i$	12 vCPUs
	Minimum CPU size $\underline{c}_i$	1 vCPUs
Bandwidth:	Bandwidth configuration b_i	2 MB/s
	Maximum bandwidth size $\bar{b}_i$	10 MB/s
	Minimum bandwidth size $\underline{b}_i$	1 MB/s

including memory, CPU, and bandwidth. Our EVRM is written in *Python* and deployed on each compute node. Each virtual instance is booted with an initial configuration, as in Table 3.

5.1. *Experiment setup*

Our EVRM aims to improve the efficiency of virtual nodes by dynamically coordinating multi-type virtual resources, we compare our EVRM with two comparison schemes by benchmark performance and resource utilization. The experiment setting is as follows.

5.1.1. Comparison schemes

To evaluate the proposed EVRM, we compare with Default case, where applications run under bare virtualization. Furthermore, most existing resource management approaches focus on scheduling a single type of resources, here we also compare with two hybrid resource management prototypes in KVM and Docker-based technologies, respectively.

- Adares [15], a VM-based baseline, leverages a RL approach, named contextual bandits, to control the CPU and memory adaptations for VMs.
- Thoth [17], a container-based baseline, is a dynamic Q-learning resource management system to adjust appropriate amount of resource for containers.
- Escra [41], a method that does not use ML, is a threshold based approach adjusting the CPU quota and maintaining a memory pool for containers.

The Thoth employs a Q-learning approach to decide whether to increase or decrease the number of application docker containers. We have adopted this model and redefined its rewards to focus on resource management instead of changing container numbers. The rationale for Thoth's decision to adjust the number of containers stems from the application's varying resource needs or excessive resource usage. Thus, we transform the container number change to resource scaling. Furthermore, we have extended the Escra methodology from container management to VM management, which is feasible because the core principle involves adjusting instance resources and transitioning from dockers to VMs is straightforward.

5.1.2. Benchmark applications

We evaluate the performance overhead with *DaCapo* [42], a Java benchmark suite. It includes different types of test applications, including processor-, memory-, and disk-intensive applications. In this paper, we choose several benchmark applications to evaluate performance overhead, including memory-intensive (i.e., *h2*), processor-intensive (i.e., *jython*, *pmd*, *avrora*, *sunflow*, *fop*, *xalan*, *lusearch*, *lusearch-fix*, *batik*), disk-intensive (i.e., *eclipse*, *luindex*) benchmarks, which are regarded as light-weight CPU intensive tests. In all evaluations, each selected *DaCapo* application runs ten times. The *Httpload* is a Linux-based webserver testing tool. Given a fixed number of concurrency p and fetch f, which illustrates the amount of data obtained in each test, the network performance can be estimated based on the completion time of the test. In network load evaluation (Section 5.3), we set dynamic values for p and f. While in other evaluations, we set default values $p = 50$ and $f = 1000$.

In order to create resource competition scenarios, we also adopt numerous workload applications to run together with benchmarks during evaluation, involving the following:

- Memory workload, is a real-world memory-consuming test application. Given a memory load size x, it occupies x amount of memory resources in a virtual instance.
- CPU workload, is an endless loop of continuous addition operations. Given the number y, it creates the corresponding y processes, and each process occupies one vCPU.
- *Mono*, is a microkernel benchmark [21]. Given a workload range $[low, high]$, *Mono* works in two phases. In the first phase, *Mono* initially requests a low-size memory, increasing its memory requests monotonically to $high$. In the second phase, *Mono* gradually reduces its memory requests from $high$ to low. *Mono* is executed with *h2* as a benchmark suite in our experiments, observing the performance of *h2* under dynamic memory workloads of *Mono*.

According to the Ref. [43], we extracted the workload characterization by monitoring the resource usage. We run all *Dacapo* applications and *Httpload* test tool sequentially and bind these tasks to 2 CPU thread cores (i.e., 200% CPU). We observe the CPU and memory usages and extract the receiving and transmission data size in Fig. 4. As shown in Figs. 4(a) and 4(b), *Dacapo* applications exhibits periodic and sinusoidal characteristics. Fig. 4(c) shows network workload characteristics. At first the receiving traffic rises sharply and then decrease quickly to a relatively steady state. Finally, it falls to 0 rapidly.

By combining these different workload and benchmark, we obtain various test suites that include static and fluctuation workloads.

5.1.3. Initial settings of parameters

In our proposed DDPG-RA, environment parameters are set by empirical analysis or actual requirements. We set scheduling period as 5 s, and set the upper and lower thresholds as 80% and 20% as system settings. The parameters in Eqs. (4) and (6), i.e., λ_1, λ_2, λ_3, γ_m, γ_c, γ_b, are all set as 1. The evaluation values (i.e., from a_1 to a_6) in Fig. 3 are set as −1,−0.1,−5,−1,−10,−10, respectively. Furthermore, momentum factor $\tau = 0.05$.

The training process of our DDPG-RA is performed by a simulator, where the Actor and Critic networks' input layers are $6n$ and $9n$ dimensional vectors, respectively. The output layers are a $3n$ and 1 dimensional vectors, respectively. In light-loaded evaluation, we choose a four-layer fully connected network as the structure of the DNN, wherein the Actor and Critic networks both possess two hidden layers with 128 and 64 dimensions. In heavy-loaded evaluation, a five-layer fully connected network is used as the structure of the DNN, wherein the Actor networks possess three hidden layers with 400, 300, and 100 dimensions, and the Critic networks possess three hidden layers with 600, 300, and 100 dimensions. The simulator configures the virtual nodes with the same initial configuration in Table 3. At the same time, to thoroughly learn the knowledge of diverse application loads, we have generated 1000 different initial load vectors for the subsequent light-loaded and heavy-loaded experiments, respectively. A load vector represents initial load information of virtual instances, include CPU, memory, and bandwidth loads. At the beginning of each episode, it randomly selects a load vector and executes lines 6–13 in Algorithm 1, wherein the load vector remains unchanged. When there is no lousy state virtual instance, it then jumps out of this episode and adopts the replay memory method to randomly sample from the previous state transition experience for training the agent. We trained 50,000 episodes for the agents used in the subsequent light-loaded and heavy-loaded

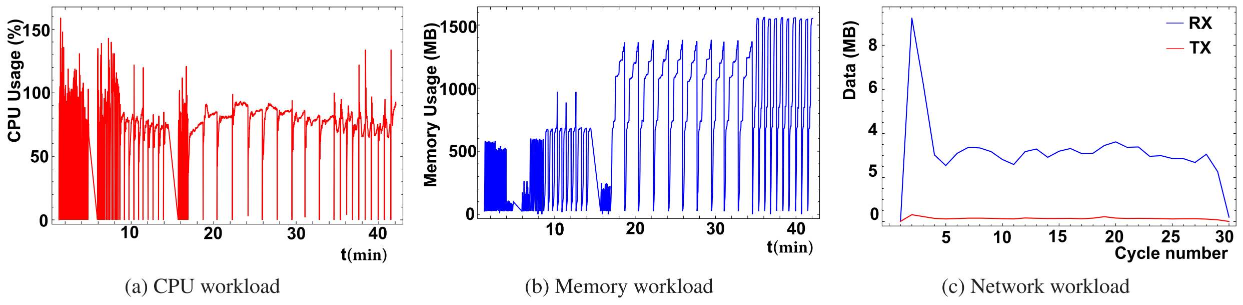


Fig. 4. CPU, memory and network workload characterization.

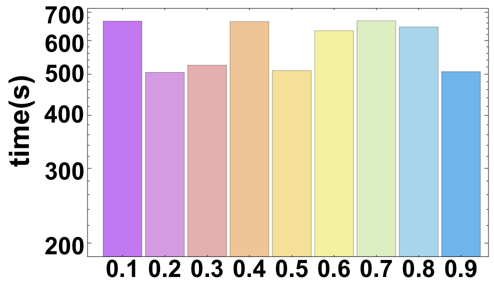


Fig. 5. The impact of ϵ values under heavy-loaded KVM experiment.

Table 4
Evaluation suites each with the same memory workload $x = 400$ MB and CPU workload $y = 1$.

Suite ID	Benchmark Name	Suite ID	Benchmark Name
1	*h2*	8	$Httpload + lusearch\_fix$
2	*h2*[*mono*]	9	$Httpload + pmd$
3	*eclipse*	10	$Httpload + luindex$
4	*avrora*	11	$Httpload + batik$
5	*jython*	12	$Httpload + fop$
6	*sunflow*	13	$Httpload + xalan$
7	*lusearch*	14	$Httpload + eclipse$
		15	$Httpload + jython$
		16	$Httpload + h2$

experiments. During the training process, we set $\alpha_\pi = 10^{-4}$, $\alpha_Q = 10^{-3}$, and $\gamma = 0.6$, based on the observation in numerous experiments.

We set 0.1 to 0.9 at the interval of 0.1 as candidate range for CPU mapping parameter ϵ in Algorithm 2. Fig. 5 presents the completion time of heavy-loaded KVM experiment (introduced in Section 5.4) under different ϵ values. The case under $\epsilon = 0.2$ shows the best performance, which is our experimental setting.

5.2. *Validation of light-loaded evaluation*

The EVRM aims to schedule virtual resources between multiple virtual instances reasonably while avoiding performance degradation. We compare the benchmark performance of the EVRM, Adares, and Default cases. We create 5 virtual instances to demonstrate whether the EVRM can shorten the benchmarks' completion times, including DaCapo and *Httpoad* applications.

We try to verify the performance improvement brought by EVRM under heavy workload states. We choose to run each suite of benchmarks in one virtual node, and the other four nodes are idle. Each suite of benchmark applications is shown in Table 4. Firstly, we start our resource scheduling program. Secondly, each testing suite includes load applications and various benchmarks to reach a heavy-loaded state in the virtual instance. During the evaluation, each *DaCapo* application in test suites runs ten times. Finally, we get the benchmark results, including average results, with error bars indicating the standard deviation of *DaCapo* applications to quantify the variance of their overall performance.

5.2.1. **KVM experiments**

Each evaluation suite will be tested under EVRM, Adares, and Default in the KVM-based environment, respectively. From the results of average running time in Figs. 6, 7, 8, we can obtain that benchmarks get lower running times under resource scheduling (based on EVRM or Adares) than Default. The benchmark performance is inversely proportional to the running time, the low completion time, the better performance. Since the EVRM gets more significant performance than Default, we mainly discuss the comparison between the two resource scheduling schemes.

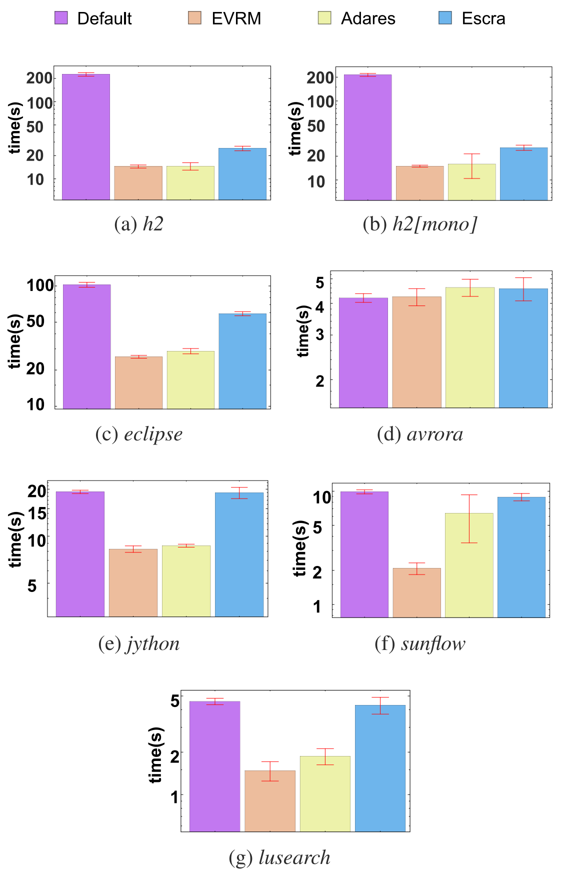


Fig. 6. The completion time of each single benchmark in a KVM instance.

First, we observe the comparison results (Fig. 6) of multiple benchmarks numbered 1 to 7 in Table 4. They get shorter completion time under EVRM than Adares and Escra, achieving 67.43% and 76.54% faster

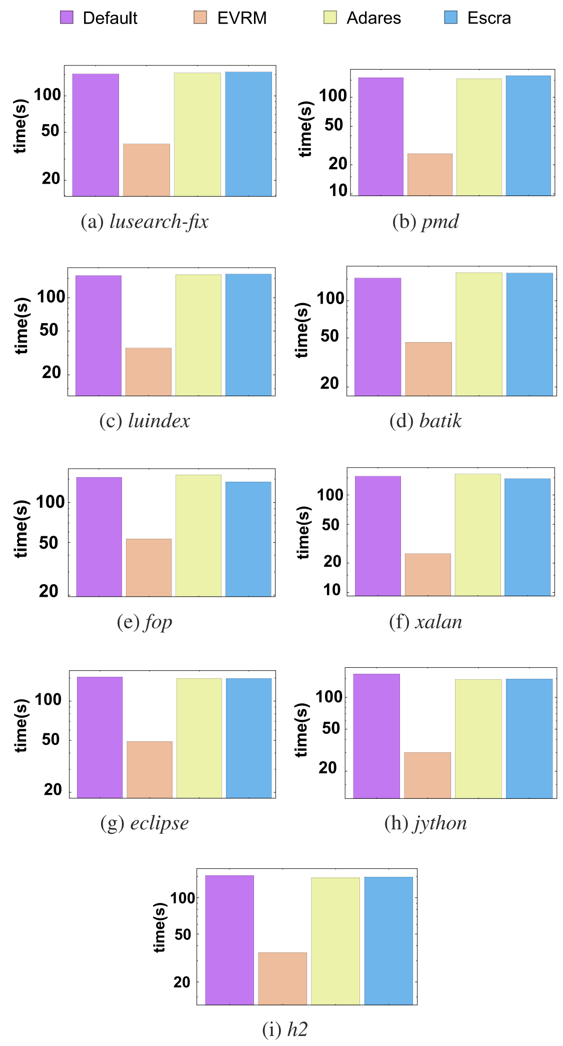


Fig. 7. The completion time of the *Httpload* task under diverse benchmarks in a KVM instance.

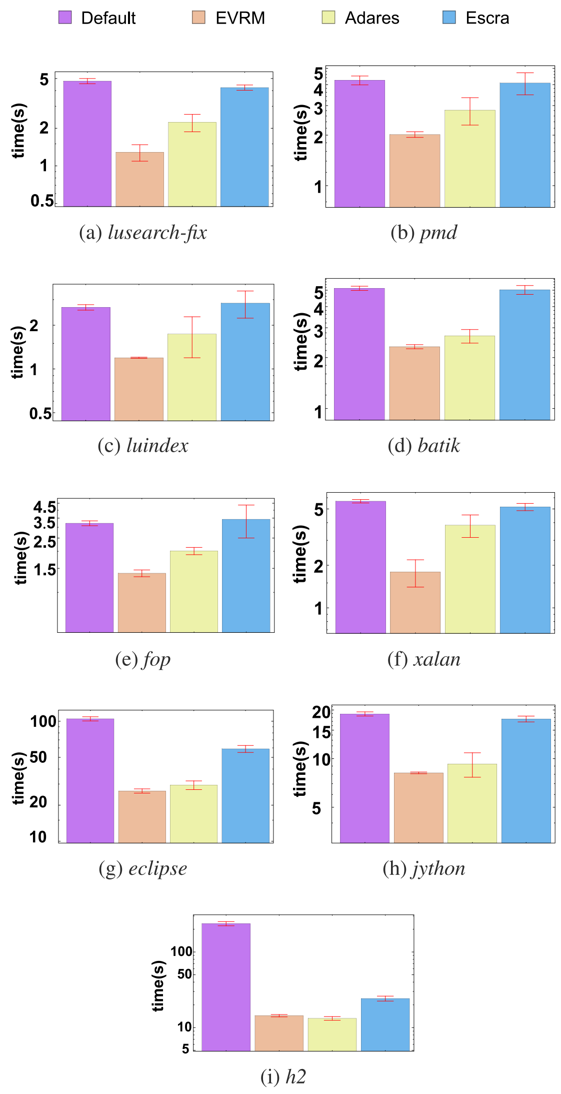


Fig. 8. The completion time of each single benchmark under the *Httpload* task in a KVM instance.

in *sunflow* experiment compared to Adares and Escra respectively. Although EVRM only shows light advantage over Adares in *h2+mono* experiment, EVRM presents significant improvement compared to Adares and Escra in other experiments.

Second, we compare the performance of network-intensive test suites numbered 8 to 16 in Table 4, which are suites combining network- and compute-intensive benchmarks. Fig. 7 illustrates the considerable reduction in completion times for network-intensive tasks (*Httpload*) under our EVRM compared to the Default, Adares and Escra, showcasing an average improvement of 49.86%. In addition to optimizing network-intensive tasks, EVRM also enhances the results of corresponding compute-intensive tasks, as demonstrated in Fig. 8. Specifically, EVRM achieves a 53.26% reduction in completion times for the *xalan* compared to Adares and a 69.65% reduction in the *lusearch* compared to Escra. While EVRM exhibits a slight performance degradation in the *h2* compared to Adares, it outperforms both Adares and Escra in other experiment suites.

5.2.2. Docker experiments

We have verified the promotions of our EVRM resource management scheme in the KVM environment through the above discussion. Then the same steps test the performance in container nodes. Finally, we compare the benchmark performance of the EVRM, Thoth, and Default cases. The results are presented in Figs. 9, 10, 11, which are similar to the results of KVM experiments. With diverse test suites arising in a container sequentially, the EVRM performs better results than the Thoth scheme as a whole between 5 containers.

Like the KVM experiments, we can conclude that our proposed EVRM and the baseline Thoth scheme perform better than the Default case. Thus, we analyze the comparison between EVRM and Thoth schemes.

First, we observe the results in Fig. 9. The benchmarks executed under EVRM present enhanced performance in terms of running speed compared to Thoth, with reductions of up to 77.48%. Similarly, when compared to Escra, EVRM demonstrates a significant speed improvement of 77.48%. Our EVRM outperforms others in all test suites.

Second, we compare the performance of network-intensive test suites numbered 8 to 16 in Table 4, which are suites combining network- and compute-intensive benchmarks. The completion times of network-intensive task (i.e., *Httpload*) are presented in Fig. 10. The results demonstrate that EVRM outperforms Default, Adares, and Escra for network-intensive tasks (*Httpload*), showcasing a significant reduction in completion times by an average of 52.39%. Finally, our EVRM performs better results of the corresponding compute-intensive tasks (Fig. 11) compared to Thoth and Escra, reducing up to 79.12% and 90% in *fop* test, respectively.

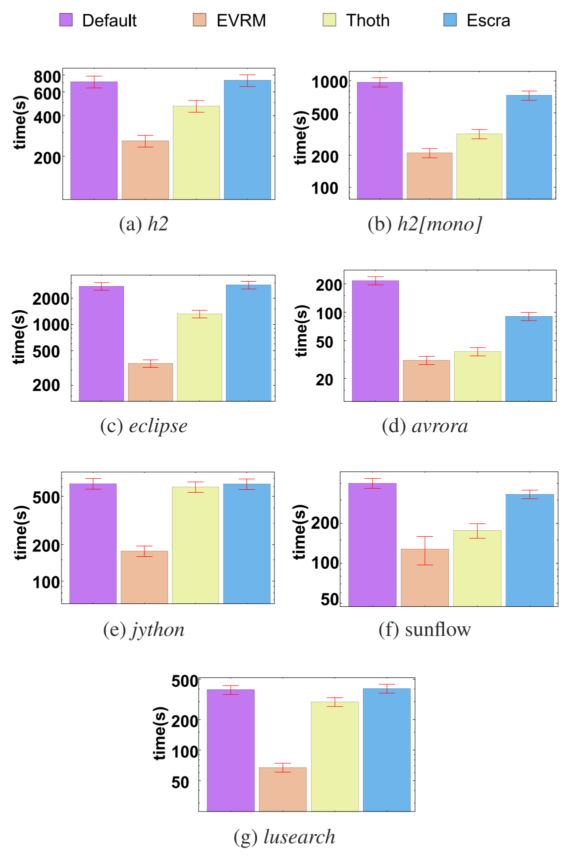


Fig. 9. The completion time of single benchmark in a Docker instance.

Since EVRM coordinates multiple resources, our EVRM improves memory-, CPU-benchmark applications and outperforms the Adares and Escra by 16.93% and 49.32% in VM and the Thoth and Escra by 61.43% and 55.96% in docker on average respectively. In light-loaded experiments, EVRM shows the best performance since it benefits from fine-grained continuous space compared to Adares and Thoth. Moreover, EVRM transforms the continuous values to discrete values through pre-training coverage, making models fit to current environment. Escra's performance is relatively poor due to its limited capability of adjusting only the CPU quota. This limitation becomes evident when dealing with heavy workloads, as Escra is constrained by the constant CPU number.

Compared to Docker experimental results in Fig. 9, KVM performs better in Fig. 6. Comprehensive explanations are taken as follows. Docker's shared-kernel architecture introduce contention between containers, especially under heavy-loaded conditions. While KVM provides stronger resource isolation by leveraging full virtualization, where each VM runs its own kernel. This reduces resource contention, particularly in CPU-bound or I/O-intensive tasks. The stronger isolation in KVM ensures better performance when the system is heavily loaded, which explains its superior performance in such conditions.

5.3. Validation of dynamic network load evaluation

To better simulate the real web workload, we change the p and f values of *Httpload* testing tool dynamically and observe the EVRM performance. We generate 20 sets of p and f values through a two-dimensional normal distribution, and then we run *Httpload* in 5 instances under different schemes. Fig. 12 presents the completion time of the *Httpload* in each instance.

Figs. 12(a) and 12(b) present the results under KVM and Docker. Our EVRM performs the best in both KVM and Docker experiments

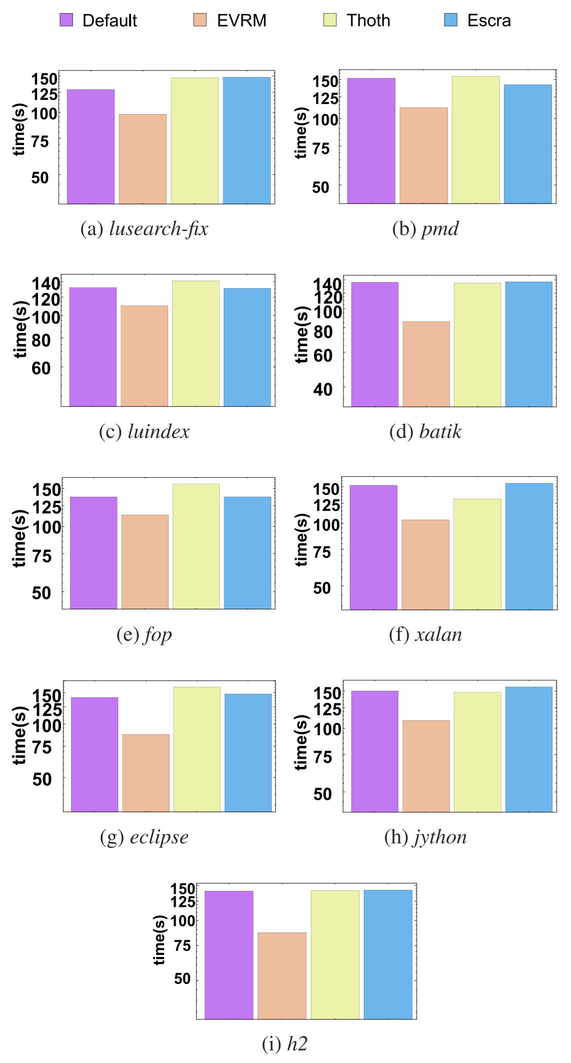


Fig. 10. The completion time of the *Httpload* task under diverse benchmarks in a Docker instance.

where the EVRM outperforms others by 10.47% and 16.24% on average, due to the fact that our EVRM well adapt to network resource scheduling.

5.4. Validation of heavy-loaded evaluation

We investigate 15 virtual instances to compare EVRM with comparison schemes in term of benchmarks' completion times. In Section 5.2, only one virtual instance runs benchmarks. In this section, we deploy diverse benchmark suites in all 15 virtual instances synchronously. Each suite in Table 5 runs on one virtual instance. Furthermore, we run the *DaCapo* benchmarks for ten times and average the results with error bars indicating the standard deviation of *DaCapo* applications. Based on the initial configuration in Table 3, the 15 virtual instances would be competitive for the CPU resource of the compute node.

5.4.1. KVM experiments

Fig. 13 displays the completion results of the benchmarks on KVM. Similarly, we find that the results under the EVRM circumstance perform the best. EVRM outperforms Default, Adares and Escra by reducing 51.16%, 51.31% and 58.04% time on average.

In Fig. 13(a), the benchmark under EVRM perform the best as a whole. The *sunflow* gets the most significant speedup, by achieving 2.2×

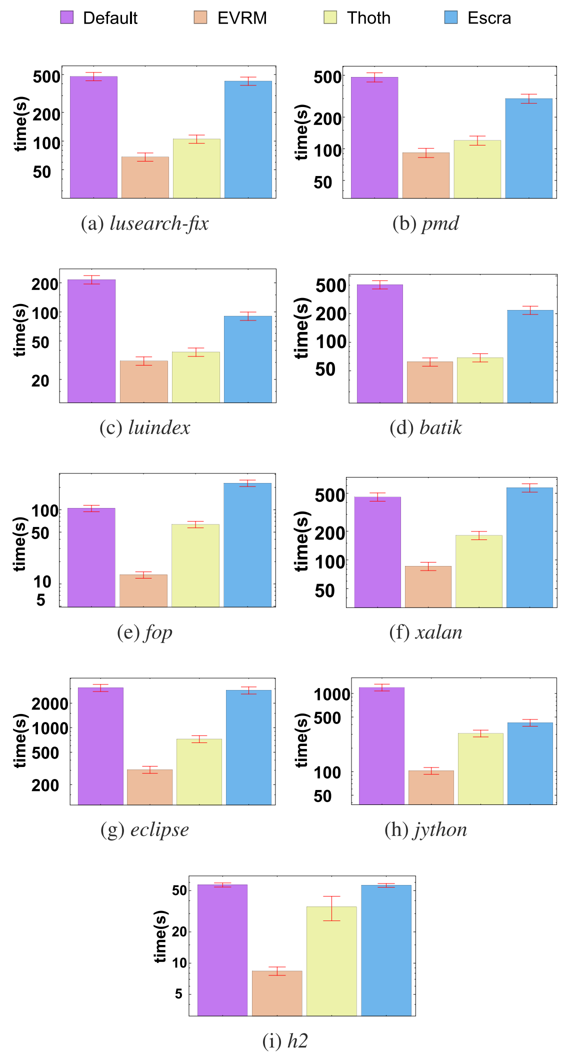


Fig. 11. The completion time of each benchmark under *Httpload* task in a Docker instance.

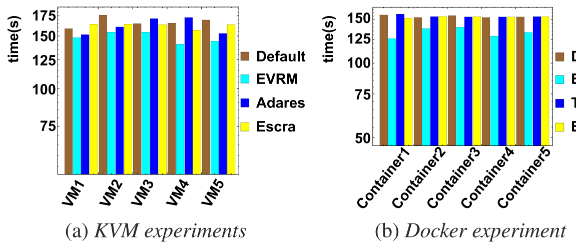


Fig. 12. Dynamic *Httpload* completion time in 5 instances.

speedup on average. However, the running speed of *avrora* drops by within 10%, which is a minor performance penalty.

When hybrid suites of the network- and compute-intensive benchmarks are tested, the proposed EVRM obtains the best *Httpload* performance by more than 3× speedup on average than other methods, under various CPU workloads (as shown in Fig. 13(b)). This result is mainly caused by the absence of bandwidth management when Default, Adares and Escra deal with multiple resource management.

As for the corresponding compute-intensive benchmarks (Fig. 13(c)), our EVRM also presents the best results as a whole. The *lusearch-fix* and *eclipse* get the best result compared to Default by obtaining 50.18%× and 66.52%× speedup, respectively. Additionally, the *xalan* also gets a slightly 4.65%× higher speed.

Table 5
Information about each evaluation suite.

Suite ID	Benchmark name	Memory load x	CPU load y
1	$h2$	300 MB	–
2	$h2[mono]$	200 MB	1
3	$eclipse$	100 MB	2
4	$avrora$	300 MB	2
5	$jython$	100 MB	1
6	$sunflow$	100 MB	–
7	$lusearch$	–	–
8	$http\_load + lusearch\_fix$	200 MB	1
9	$http\_load + pmd$	200 MB	–
10	$http\_load + luindex$	200 MB	1
11	$http\_load + batik$	200 MB	1
12	$http\_load + fop$	–	–
13	$http\_load + xalan$	–	–
14	$http\_load + eclipse$	100 MB	–
15	$http\_load + jython$	200 MB	–

5.4.2. Docker experiments

Similarly, we conduct the same experiments on containers. Comparison results are presented in Fig. 14. In Fig. 14(a), our EVRM outperforms Thoth and Escra. For example, the eclipse results present 2.1× and 4.3× speedup in completion time compared to Thoth and Escra. Although Thoth shows 1.22× and 1.74× improvements compared to EVRM in *avrora* and *jython*, EVRM achieves 56.47% improvements on average over Thoth, and EVRM still performs the best as a whole.

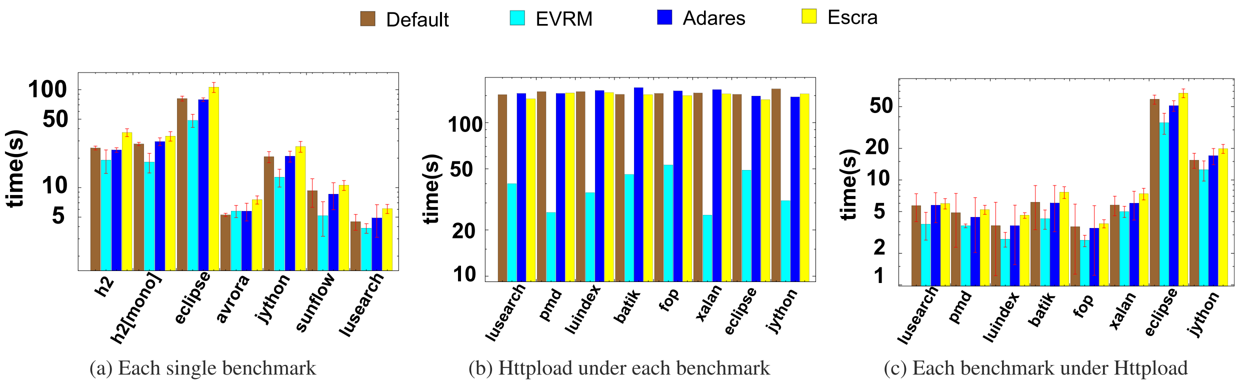
As for the results of the hybrid suites of the network- and compute-intensive benchmarks in Figs. 14(b) and 14(c), the proposed EVRM presents at least 25.04% and 26.55% time reduction than Thoth and Escra scheme respectively, under various DaCapo benchmarks (as shown in Fig. 14(b)). Furthermore, the EVRM also obtains the best results of corresponding compute-intensive benchmarks (Fig. 14(c)). Compared to Thoth and Escra scheme, EVRM reduces the completion time by 61.59% and 66.13% respectively.

These results presented in Figs. 13 and 14 verify that EVRM can shorten the completion times of diverse applications under heavy workload circumstances. While slight performance drops in some virtual instances can be observed under peak resource contention, this is a challenging situation where it may be difficult to meet the SLO requirements of all instances. In such cases, horizontal scaling becomes an essential strategy to ensure SLO compliance by distributing workloads across additional resources when vertical scaling reaches its limits.

EVRM performs the best under heavy-loaded experiments since we define Eq. (6) and reward parameters a_1 to a_6 to model the system behavior. In the pre-training process the model learns the intricate relationship between resources and system performance, and studies how to minimize Eq. (6) to optimize the system accordingly. In the experiments, EVRM dynamically adjusts resources to minimize the occurrence of bad states instances. Although Escra maintains a memory pool and it actually benefits from CPU quota adjust and memory pool under light-loaded experiments, it presents limited performance in heavy-loaded experiments due to the fixed CPU number. Moreover, EVRM outperforms Adares and Thoth due to fine-grained output.

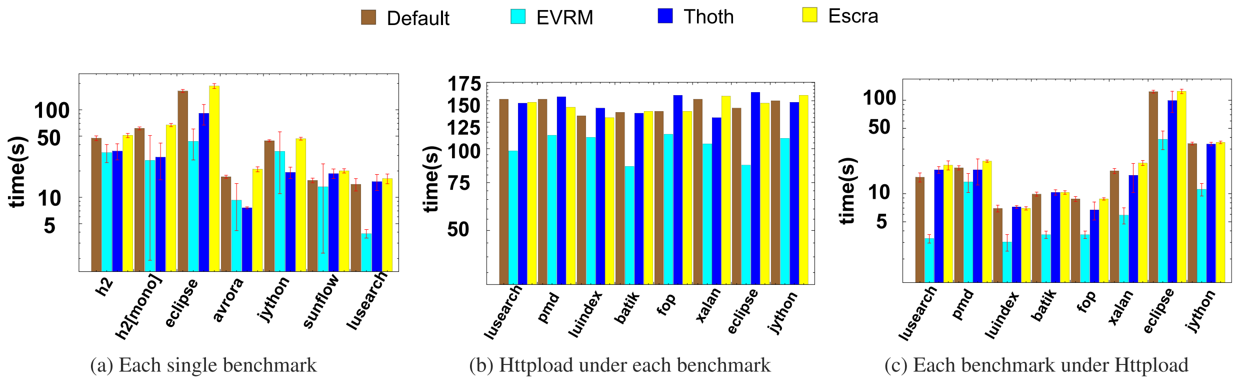
5.4.3. Resource violation

In our experiments, we also compare EVRM with other existing CPU-memory scheduling schemes on CPU and memory violations in heavy-loaded KVM and Docker experiments, as presented in Fig. 15. As for resource violation, Fig. 3 illustrates that instances violate the threshold when their resource usage exceeds the upper threshold or falls below the lower threshold. For CPU and memory, we set the lower threshold at 0.2 and the upper threshold at 0.8.



(a) Each single benchmark (b) Httpload under each benchmark (c) Each benchmark under Httpload

Fig. 13. Comparisons of the completion time in VMs.



(a) Each single benchmark (b) Httpload under each benchmark (c) Each benchmark under Httpload

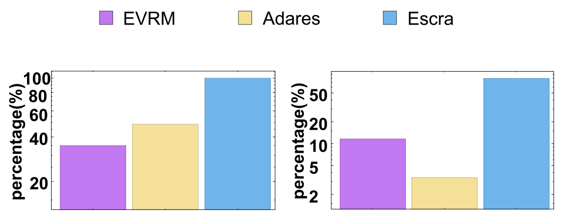
Fig. 14. Comparisons of the completion time in containers.

To better explain resource violation, we present the 14th VM's resource log which is extracted in a heavy-loaded VM experiment under the EVRM scheme. Figs. 16(a) and 16(b) shows the CPU and memory information, respectively. Memory usage is in violation condition at first, and then EVRM enlarge the memory to avoid the violation demands. CPU usage is low at the beginning, thus EVRM adjusts the CPU from 4 to 3. However, the workload grows gradually and it is under violation condition later in the 16th iteration. As a result, EVRM then adjusts CPU from 3 to 4 to better cope with the surge workload. We record the violation time proportion as violation result.

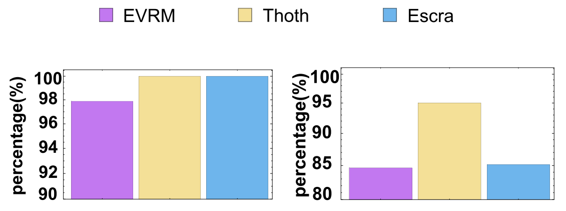
In our KVM experiments, EVRM demonstrates the lowest CPU violation rate and the second-lowest memory violation rate among all tested methods. Conversely, Escra consistently exhibits the highest violation rates due to its limited approach of adjusting only the CPU quota, resulting in consistently high CPU usage. EVRM's superior performance in CPU violations can be attributed to the accurate training of the DDPG module to match system behavior. Regarding memory violations, EVRM shows second-lowest violation rate. While memory violation shows limited impacts in our heavy-loaded KVM experiments, our EVRM possesses the best benchmark running performance (Fig. 13) at the expense of memory violation.

Similarly, our EVRM demonstrates the best violation rate for both CPU and bandwidth in Docker experiments. Notably, the CPU violation rate is relatively high in Docker environments due to the heavy nature of the Dacapo and *Httpload* test suites for Docker. In memory violation experiments, EVRM consistently strives to maintain resource utilization within an appropriate range, resulting in the lowest violation rate. Thoth exhibits suboptimal behavior under heavy-loaded conditions because it can only adjust one instance at a time. On the other hand, Escra performs well by managing a memory pool to control memory utilization.

In summary, faster completion of tasks reflects more efficient resource utilization, highlighting that our EVRM consistently achieves

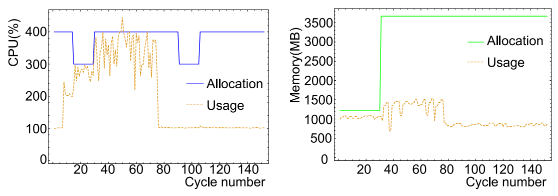


(a) CPU violation in VM (b) Memory violation in VM



(c) CPU violation in Docker (d) Memory violation in Docker

Fig. 15. CPU and memory violations.



(a) CPU usage (b) Memory usage

Fig. 16. CPU and memory information in VM14.

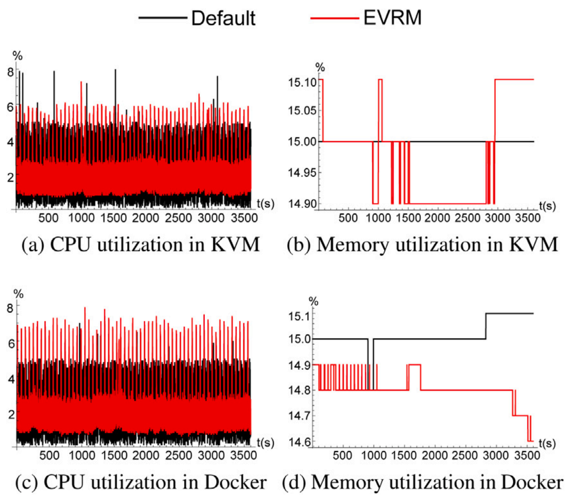


Fig. 17. System overhead comparison.

faster completion times with efficient resource usage. Consequently, EVRM optimally utilizes physical resources and enhances the running performance of virtual instances.

5.5. System overhead

We evaluate the system overhead, or in other words the self-adaptive overhead of EVRM in terms of CPU and memory utilization. We boot 15 bare VMs and containers on a nova node and a nova-docker node, called Default case, and observe CPU and memory utilization within an hour. Subsequently, we adopt EVRM to schedule CPU, memory, and bandwidth for all virtual instances once a minute, at the range of [1 vCPU, 2 vCPU], [1024 MB, 2048 MB], and [1 MB/s, 2 MB/s], respectively. Furthermore, we also observe CPU and memory utilization within an hour. In this way, we can evaluate the CPU and memory overhead of our EVRM that executes resource scheduling actions.

In Figs. 17, the red lines show the CPU and memory overhead incurred during frequent scheduling on CPU, memory, and bandwidth for VM and container nodes. The black lines are default CPU and memory overhead. Although many unknown system processes interfere with the results during the observation, we can find that our EVRM appears similar CPU and memory overhead to the default KVM or Docker host. Our EVRM does not generate significant overhead, which demonstrates the availability of our EVRM.

Except the resource overhead, we also observe the latency caused by EVRM. We have compared two situations in both KVM and Docker experiments with heavy-loaded evaluation setting. One with maximum resource configuration called No-limit case and the other uses EVRM with resource restriction at the beginning. In No-limit case, we allocate 12 CPU, 8 GB memory and 10 MB/s bandwidth for VMs and containers. Finally, we get the application completion time under two cases in Fig. 18.

Figs. 18(a) and 18(b) present the results under KVM and Docker. Compared to No-limit case, the applications running under EVRM exhibit close completion time. Our EVRM shows negligible latency expenditure because EVRM adjusts the resource properly and there is no downtime when modifying resource, which we have clarified in Section 3.3.

5.6. Replay memory effect

We record the pre-training overheads to evaluate the number of adaptations required to reach the target threshold. We train the model in 1000 episodes and each episode includes 100 iterations to find the target optimized reward. We compare the training rewards with and without replay memory. We find that EVRM with replay memory needs around 200 episodes while EVRM without replay memory requires 150 episodes, as shown in Fig. 19. These results suggest that training EVRM without replay memory involves fewer episodes compared to training with replay memory because training EVRM with replay memory takes the previous data to train to mitigate overfitting problem.

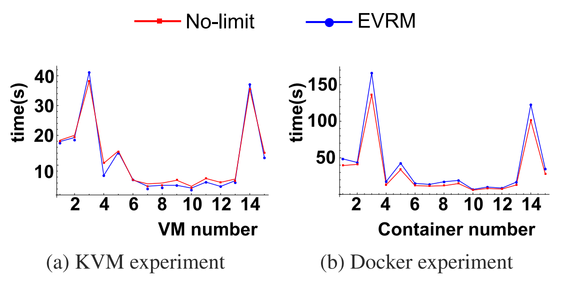


Fig. 18. System latency observation.

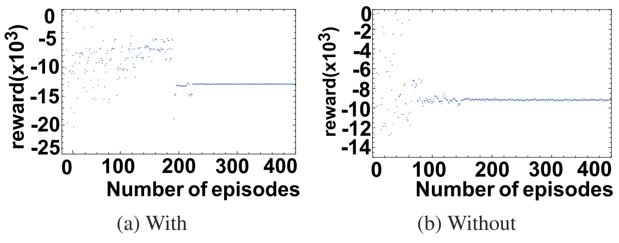


Fig. 19. Convergence rate with vs without replay memory.

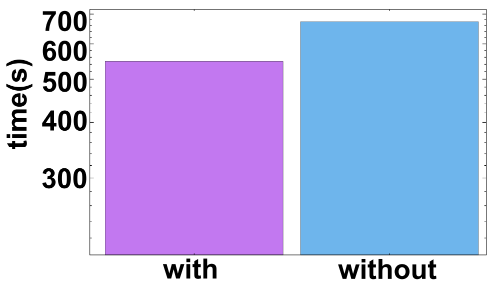


Fig. 20. Completion time of heavy-loaded KVM experiment with vs without replay memory.

To compare the effect of replay memory on model performance, we conduct heavy-loaded KVM experiments under two circumstances and the results are shown in Fig. 20. The results present that the EVRM trained with replay memory runs faster, because replay memory helps mitigate overfitting effects.

6. Conclusion

In this paper, we have developed the EVRM, which integrates monitoring, analysis, planning, and execution modules to achieve effective resource consolidation under both KVM and Docker virtualization. Subsequently, we have introduced a novel model for the elastic virtual resource management problem and proposed the DDPG-RA, a comprehensive resource allocation algorithm that manages multiple resources—CPU, memory, and bandwidth. DDPG-RA leverages the DDPG algorithm to derive ratio-based action decisions, which are further refined into resource allocation choices through an action refinement algorithm in the subsequent phase.

We have compared the performance of EVRM against two existing resource management frameworks through light- and heavy-load evaluations in both VM and container instances. The results demonstrate that

EVRM significantly outperforms these approaches in benchmark application completion times, achieving reductions of 64.4% and 41.37% on average in KVM and Docker during light-loaded evaluations, and 39.11% and 43.38% during heavy-loaded evaluations. Additionally, EVRM maintains lower CPU and memory violations compared to other methods in both environments, while exhibiting negligible self-adaptive overhead. These findings confirm that EVRM effectively adapts to workload variations for virtual nodes on a consolidated server.

Looking ahead, a promising area for future research is the joint optimization of local resource management and horizontal virtual instance management to enhance server consolidation in cloud data centers.

CRediT authorship contribution statement

Desheng Wang: Writing – original draft, Visualization, Validation, Software, Methodology, Formal analysis, Conceptualization. **Yiting Li:** Writing – review & editing, Methodology, Conceptualization. **Weizhe Zhang:** Writing – review & editing, Supervision, Project administration, Conceptualization. **Zhiji Yu:** Writing – review & editing, Conceptualization. **Yu-Chu Tian:** Writing – review & editing. **Keqin Li:** Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments and suggestions. This work was supported in part by the Shenzhen Colleges and Universities Stable Support Program No. GXWD20231129102636001, in part by the Guangdong Basic and Applied Basic Research Foundation (Grant No. 2023A1515110271), in part by the Shenzhen Colleges and Universities Scientific Research Startup Foundation for Newly Introducing Advanced Scholars (entitled Research on Hybrid Deployment for Deep Learning Jobs in Intelligent Clusters), in part by the Joint Funds of the National Natural Science Foundation of China (Grant No. U22A2036), in part by the Shenzhen Colleges and Universities Stable Support Program Nos. GXWD20220817124251002 and GXWD20231130110352002, in part by the Scientific Research Startup Foundation of Harbin Institute of Technology Shenzhen, in part by the Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies (2022B1212010005), and in part by the Australian Research Council (ARC) through the Discovery Projects Scheme (Grant No. DP220100580). Professor Weizhe Zhang is the corresponding author.

Data availability

No data was used for the research described in the article.

References

[1] S. Chouliaras, S. Sotiriadis, Towards constrained optimization of cloud applications: A hybrid approach, Future Gener. Comput. Syst. 151 (2024) 100–110.
[2] P. Osypanka, P. Nawrocki, Qos-aware cloud resource prediction for computing services, IEEE Trans. Serv. Comput. 16 (2) (2022) 1346–1357.
[3] J.N. Acharya, A.C. Suthar, Docker container orchestration management: A review, in: International Conference on Intelligent Vision and Computing, Springer, 2021, pp. 140–153.
[4] B. Jeong, J. Jeon, Y.-S. Jeong, Proactive resource autoscaling scheme based on SCINet for high-performance cloud computing, IEEE Trans. Cloud Comput. (2023).
[5] D. Saxena, A.K. Singh, A high availability management model based on VM significance ranking and resource estimation for cloud applications, IEEE Trans. Serv. Comput. 16 (3) (2022) 1604–1615.
[6] Y. Gao, L. Wang, Z. Xie, Z. Qi, J. Zhou, Energy-and quality of experience-aware dynamic resource allocation for massively multiplayer online games in heterogeneous cloud computing systems, IEEE Trans. Serv. Comput. 16 (3) (2022) 1793–1806.
[7] C. Lu, H. Xu, K. Ye, G. Xu, L. Zhang, G. Yang, C. Xu, Understanding and optimizing workloads for unified resource management in large cloud platforms, in: Proceedings of the Eighteenth European Conference on Computer Systems, 2023, pp. 416–432.
[8] E. Zeydan, J. Mangues-Bafalluy, J. Baranda, R. Martínez, L. Vettori, A multi-criteria decision making approach for scaling and placement of virtual network functions, J. Netw. Syst. Manage. 30 (2) (2022) 32.
[9] J. Dogani, R. Namvar, F. Khunjush, Auto-scaling techniques in container-based cloud and edge/fog computing: Taxonomy and survey, Comput. Commun. 209 (2023) 120–150.
[10] X. Chen, F. Zhu, Z. Chen, G. Min, X. Zheng, C. Rong, Resource allocation for cloud-based software services using prediction-enabled feedback control with reinforcement learning, IEEE Trans. Cloud Comput. 10 (2) (2020) 1117–1129.
[11] G. Sheganaku, S. Schulte, P. Waibel, I. Weber, Cost-efficient auto-scaling of container-based elastic processes, Future Gener. Comput. Syst. 138 (2023) 296–312.
[12] D. Wang, W. Zhang, H. He, Y.-C. Tian, Efficient hybrid central processing unit/input–output resource scheduling for virtual machines, IEEE Trans. Ind. Electron. 68 (3) (2020) 2714–2724.
[13] Z. Ye, Y. Wang, S. He, C. Xu, X.-H. Sun, Sova: A software-defined autonomic framework for virtual network allocations, IEEE Trans. Parallel Distrib. Syst. 32 (1) (2020) 116–130.
[14] E. Makridis, K. Deliparaschos, E. Kalyvianaki, A. Zolotas, T. Charalambous, Robust dynamic CPU resource provisioning in virtualized servers, IEEE Trans. Serv. Comput. 15 (2) (2020) 956–969.
[15] I.A. Cano, Optimizing Distributed Systems using Machine Learning (Ph.D. thesis), University of Washington, Seattle, USA, 2019.
[16] P.-J. Maenhaut, B. Volckaert, V. Ongenae, F. De Turck, Resource management in a containerized cloud: Status and challenges, J. Netw. Syst. Manage. 28 (2) (2020) 197–246.
[17] A. Sangpetch, O. Sangpetch, N. Juangmarisakul, S. Warodom, Thoth: Automatic resource management with machine learning for container-based cloud platform, in: International Conference on Cloud Computing and Services Science, vol. 2, SCITEPRESS, 2017, pp. 103–111.
[18] B. Bermejo, C. Juiz, C. Guerrero, Virtualization and consolidation: A systematic review of the past 10 years of research on energy and performance, J. Supercomput. 75 (2) (2019) 808–836.
[19] E.G. Radhika, G.S. Sadasivam, A review on prediction based autoscaling techniques for heterogeneous applications in cloud environment, Mater. Today: Proc. 45 (2021) 2793–2800.
[20] T.P. Lillicrap, Continuous control with deep reinforcement learning, 2015, arXiv preprint arXiv:1509.02971.
[21] W.-Z. Zhang, H.-C. Xie, C.-H. Hsu, Automatic memory control of multiple virtual machines on a consolidated server, IEEE Trans. Cloud Comput. 5 (1) (2015) 2–14.
[22] G. Lahmann, T. McCann, W. Lloyd, Container memory allocation discrepancies: An investigation on memory utilization gaps for container-based application deployments, in: 2018 IEEE International Conference on Cloud Engineering, IC2E, IEEE, 2018, pp. 404–405.
[23] C.H.Z. Nicodemus, C. Boeres, V.E.F. Rebello, Managing vertical memory elasticity in containers, in: 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing, UCC, IEEE, 2020, pp. 132–142.
[24] T. Melissaris, K. Nabar, R. Radut, S. Rehmtulla, A. Shi, S. Chandrashekar, I. Papapanagiotou, Elastic cloud services: Scaling snowflake's control plane, in: Proceedings of the 13th Symposium on Cloud Computing, 2022, pp. 142–157.
[25] J. Jang, J. Jung, J. Hong, An efficient virtual CPU scheduling in cloud computing, Soft Comput. 24 (8) (2020) 5987–5997.
[26] T. Wang, S. Ferlin, M. Chiesa, Predicting CPU usage for proactive autoscaling, in: Proceedings of the 1st Workshop on Machine Learning and Systems, 2021, pp. 31–38.
[27] Z. Wang, S. Zhu, J. Li, W. Jiang, K.K. Ramakrishnan, Y. Zheng, M. Yan, X. Zhang, A.X. Liu, DeepScaling: Microservices autoscaling for stable CPU utilization in large scale cloud systems, in: Proceedings of the 13th Symposium on Cloud Computing, 2022, pp. 16–30.
[28] Y. Zhao, A. Uta, Tiny autoscalers for tiny workloads: Dynamic CPU allocation for serverless functions, in: 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing, CCGrid, IEEE, 2022, pp. 170–179.
[29] K. Karmakar, R.K. Das, S. Khatua, Bandwidth allocation for communicating virtual machines in cloud data centers, J. Supercomput. 76 (2020) 7268–7289.
[30] L. Chen, Y. Feng, B. Li, B. Li, Efficient performance-centric bandwidth allocation with fairness tradeoff, IEEE Trans. Parallel Distrib. Syst. 29 (8) (2018) 1693–1706.
[31] W. Li, D. Guo, A.X. Liu, K. Li, H. Qi, S. Guo, A. Munir, X. Tao, CoMan: Managing bandwidth across computing frameworks in multiplexed datacenters, IEEE Trans. Parallel Distrib. Syst. 29 (5) (2017) 1013–1029.

[32] L. Zeng, Y. Wang, X. Fan, C. Xu, Raccoon: A novel network i/o allocation framework for workload-aware VM scheduling in virtual environments, IEEE Trans. Parallel Distrib. Syst. 28 (9) (2017) 2651–2662.
[33] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, et al., Autopilot: Workload autoscaling at google, in: Proceedings of the Fifteenth European Conference on Computer Systems, 2020, pp. 1–16.
[34] C. Prakash, P. Prashanth, U. Bellur, P. Kulkarni, Deterministic container resource management in derivative clouds, in: 2018 IEEE International Conference on Cloud Engineering, IC2E, IEEE, 2018, pp. 79–89.
[35] S. Chouliaras, S. Sotiriadis, Auto-scaling containerized cloud applications: A workload-driven approach, Simul. Model. Pract. Theory 121 (2022) 102654.
[36] D.-D. Vu, M.-N. Tran, Y. Kim, Predictive hybrid autoscaling for containerized applications, IEEE Access 10 (2022) 109768–109778.
[37] M. Maurer, I. Breskovic, V.C. Emeakaroha, I. Brandic, Revealing the MAPE loop for the autonomic management of cloud infrastructures, in: 2011 IEEE Symposium on Computers and Communications, ISCC, IEEE, 2011, pp. 147–152.
[38] D. Wang, W. Zhang, X. Wang, Y. Xiang, Y.-C. Tian, Kalman prediction-based virtual network experimental platform for smart living, Comput. Commun. 177 (2021) 156–165.
[39] G. Somma, C. Ayimba, P. Casari, S.P. Romano, V. Mancuso, When less is more: Core-restricted container provisioning for serverless computing, in: IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops, INFOCOM WKSHPS, IEEE, 2020, pp. 1153–1159.
[40] B.-A. Slim, C.-G. Liliana, D. Maxim, Worst-case response time analysis for partitioned fixed-priority dag tasks on identical processors, in: 2019 24th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA, IEEE, 2019, pp. 1423–1426.
[41] G. Cusack, M. Nazari, Escra: Event-driven, sub-second container resource allocation, in: IEEE International Conference on Distributed Computing Systems, 2022.
[42] S.M. Blackburn, R. Garner, C. Hoffmann, A.M. Khang, K.S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S.Z. Guyer, et al., The DaCapo benchmarks: Java benchmarking development and analysis, in: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, 2006, pp. 169–190.
[43] M.C. Calzarossa, L. Massari, D. Tessera, Workload characterization: A survey revisited, ACM Comput. Surv. 48 (3) (2016) 1–43.

Desheng Wang received the B.Eng degree in computer science and engineering from Harbin Engineering University, Harbin, China, in 2015, and the Ph.D. degree in Cyberspace Security from Harbin Institute of Technology, Harbin, China, in 2022. He is currently an assistant professor in the School of Computer Science and Technology at Harbin Institute of Technology, Shenzhen, China, His research interests include cloud computing and machine learning.

Yiting Li received the B.Eng degree in computer science from Harbin Institute of Technology, ShenZhen, China, where he is currently working toward the M.Eng degree in the School of Cyberspace Science. During the undergraduate period, he won the second prize of Guangdong Province in the National Competition of Mathematical Modeling in 2021, the first prize of the National Operating System Competition in 2022, and the third prize of the National Operating System Competition in 2023. Now his main research interest lies in serverless architecture in cloud computing.

Weizhe Zhang: received the B.Eng, M.Eng and Ph.D. degrees all in computer science and technology from Harbin Institute of Technology, China, in 1999, 2001 and 2006, respectively. He is currently a professor in the School of Computer Science and Technology at Harbin Institute of Technology, Shenzhen, China, and the director of the Cyberspace Security Research Center, Peng Cheng Laboratory, Shenzhen, China. His research interests are primarily in parallel computing, distributed computing, cloud and grid computing, and computer networks.

Zhiji Yu received the B.S degree in Data science and Big Data technology from Zhejiang Ocean University, Zhoushan, China. He is currently working toward the M.Eng degree at Harbin Institute of Technology, Shenzhen, China. His research interest includes cloud computing.

Yuchu Tian: received the Ph.D. degree in computer and software engineering from the University of Sydney, Sydney NSW, Australia, in 2009, and the Ph.D. degree in industrial automation from Zhejiang University, Hangzhou, China, in 1993. He is currently a professor with the School of Computer Science, Queensland University of Technology, Brisbane QLD, Australia. His research interests include big data computing, cloud computing, computer networks, smart grid communications, optimization and machine learning, networked control systems, and cyber–physical system security.

Keqin Li is a SUNY Distinguished Professor of computer science with the State University of New York. He is also a National Distinguished Professor with Hunan University, China. His current research interests include cloud computing, fog computing and mobile edge computing, energy-efficient computing and communication, embedded systems and cyber–physical systems, heterogeneous computing systems, big data computing, high-performance computing, CPU–GPU hybrid and cooperative computing, computer architectures and systems, computer networking, machine learning, intelligent and soft computing. He has authored or coauthored over 810 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He holds over 60 patents announced or authorized by the Chinese National Intellectual Property Administration. He is among the world's top 10 most influential scientists in parallel and distributed computing based on a composite indicator of Scopus citation database. He has chaired many international conferences. He is currently an associate editor of the ACM Computing Surveys and the CCF Transactions on High Performance Computing. He has served on the editorial boards of the IEEE Transactions on Parallel and Distributed Systems, the IEEE Transactions on Computers, the IEEE Transactions on Cloud Computing, the IEEE Transactions on Services Computing, and the IEEE Transactions on Sustainable Computing. He is an IEEE Fellow.