IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 2 January 2023, accepted 8 February 2023, date of publication 13 February 2023, date of current version 16 February 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3244388

RESEARCH ARTICLE

A Configurable Model-Based Reinforcement Learning Framework for Disaggregated Storage Systems

SEUNGHWAN JEONG AND HONGUK WOO, (Member, IEEE)
Department of Computer Science and Engineering, Sungkyunkwan University, Suwon 16419, South Korea

Corresponding author: Honguk Woo (hwoo@skku.edu)

This work was supported in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grant by the Korea Government through MSIT under Grant 2022-0-01045 and Grant 2022-0-00043, and in part by Samsung Electronics.

ABSTRACT With the rapid growth of data-intensive jobs and the use of different hardware in storage, disaggregated storage architecture systems are being used to improve the operational cost efficiency of data centers. The hardware heterogeneity and mixed configurations of disaggregated storage systems, along with the diversity of workloads, often make it difficult for administrators to operate them optimally. In this work, we investigate model-based reinforcement learning (RL) schemes to develop automated system operations and maintain the storage performance across various system settings and workloads in self-managed storage systems. Specifically, we propose a novel configurable model structure in which a system environment is abstracted with a two-level hierarchy of storage devices and a platform and thus the environment can be reconfigured according to a given system specification. Using that novel model structure, we implement a configurable model-based RL framework CoMoRL by which RL agents are trained through model variants that represent a variety of storage system specifications; thus, their learned management policy can be highly robust to the diverse operation conditions of real-world storage systems. We evaluate our CoMoRL framework using a storage cluster that relies on NVMe-oF devices and demonstrate that the framework can be adapted to different scenarios such as volume placement scenarios with Kubernetes and primary affinity control scenarios with Ceph. The learned management policy outperforms an IOPS-based heuristic method and a model-based method by 0.7%∼5.1% and 11.8%∼29.7%, respectively, for various Kubernetes system specifications, and by 1.6%∼5.6% and 8.2%∼16.5%, respectively, for various Ceph system specifications, without requiring model and policy retraining. This zero-shot adaptation superiority of our framework makes it possible to realize RL-based self-managing storage systems in data centers with frequent system changes.

INDEX TERMS Model-based reinforcement learning, configurable model, meta learning, policy adaptation, data placement, disaggregated storage, heterogeneous storage.

I. INTRODUCTION

The technology trend of disaggregated storage architectures has the benefits of flexibility and high efficiency in storage system operation, allowing for fine-grained, device-level upgrades and mixed configurations of heterogeneous devices with different I/O capabilities in data centers [1], [2]. This trend, which involves both disaggregation and heterogeneity in storage systems, is quite desirable from a total cost of ownership perspective. However, it can inherently cause performance issues. When the layout of data is managed by a conventional storage platform that does not account for the heterogeneous capabilities of storage devices, applications often experience lower-than-expected storage performance [3], [4]. Hot data, frequently requested in large amounts, and latency-sensitive data are preferably served by high-performance enterprise-grade storage devices; however, when storage size is limited and data access patterns vary

The associate editor coordinating the review of this manuscript and approving it for publication was Zhe Xiao.

dynamically, data can be misplaced. Furthermore, storage workloads become increasingly complicated, as various data-intensive applications run with different I/O through requirements and latency limits.

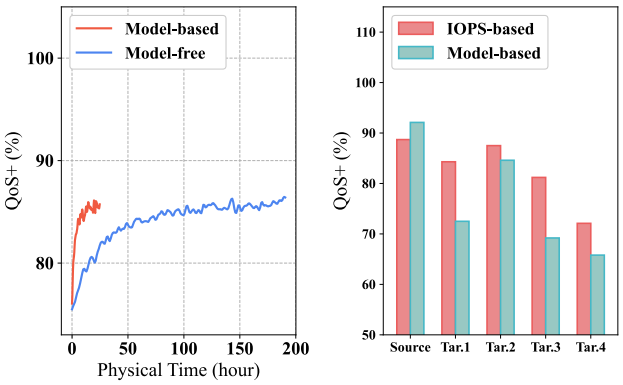
Reinforcement learning (RL)-based approaches integrated with deep neural networks have proved their applicability in automated system operation and resource management, e.g., on the single device level such as I/O merging [5], caching [6], and garbage collection [7], [8] and on the system level such as cluster resource management [9], object placement [10], network traffic engineering [11], [12], [13], and database index selection [14]. These applications of RL formulate system operation tasks as Markov decision processes (MDPs), by which an optimal policy for sequential management decisions can be learned from experiences with or operation logs from the target system.

In the context of self-managing storage systems, recently, there were several research works using RL algorithms [10], [15]. In Databot+ [10], a Q-learning-based management agent was trained through the Mininet Simulator [16] to determine server locations of I/O requests and reduce their latency. In ARM [15], an RL agent was learned to select an effective algorithm among several predefined heuristic algorithms for load balancing on a Ceph storage cluster. These works were either evaluated only in simulation environments or were limitedly tested in certain management scenarios with a small action space. The limitations of these prior works are attributed to the sample inefficiency of existing RL-based approaches, which can arise when it is difficult to aggregate operation log data sufficiently for RL training through direct interaction, such as a storage system. Even for ARM with a small discrete action space (i.e. a set of predefined heuristic management algorithms), RL training took 83 hours when learning by continual interaction with the target Ceph testbed, as reported in [15].

Figure 1(a) shows the learning curves of RL training with our Kubernetes testbed that is composed of 4 storage nodes. For this test, we implemented a data placement policy by which the state of a Kubernetes testbed is observed and inferred actions in data migration commands are performed every second. The learning curves show that more than 200 hours were needed for RL training where Model-free (the blue-colored curve in Figure 1(a)) corresponds to learning via direct interaction with the target system. In the RL literature, model-based RL methods, by which a model provides a simulation environment where the dynamics of the target system are abstracted according to its MDP, have been investigated for improving sample efficiency [17], [18], [19].

Our implementation with model-based RL (the red-colored curve in Figure 1(a)) indicates such sample efficiency, in contrast to the Model-free method.

In this paper, we explore model-based RL methods for disaggregated, heterogeneous storage systems to allow learned RL policies to adapt to continual system changes. In model-based RL, an MDP is defined and learned specifically for



(a) Sample inefficiency of model-free (b) Limited adaptation of model-based

FIGURE 1. Limitation of conventional (a) model-free and (b) model-based RL methods for storage system management. In (a), the x-axis denotes training time in hours and the y-axis denotes the achieved performance explained in Section IV-A. In (b), the x-axis denotes different storage systems where a model for model-based RL is learned on a specific (Source) system and an RL agent learned through the model is evaluated with different Tar(get) systems.

a given system, so if the system is changed, adaptation or retraining issues might arise in practice. The operation conditions of a data center with disaggregated storage systems vary continually due to frequent configuration updates such as storage node scale-in and -out and device upgrades and replacements as workload patterns change and devices fail. However, such changing conditions have not been fully investigated in the prior works. Figure1(b) demonstrates that the performance of model-based RL degrades more than the other heuristic method when the system is changed (i.e., from the source to other targets 1∼4), although it achieves higher performance than the heuristic method for the source system where it is learned. The detailed implementation is described in Section IV.

Existing model learning techniques in RL rarely account for configuration updates to the target system. The techniques normally rely on a monolithic model structure, so they are inherently unsuited to flexible reconstruction and rapid adaptation. In the context of disaggregated, heterogeneous storage systems, yet, a model that can readily accommodate changes without retraining from scratch is desirable. This limitation of existing RL approaches motivates us to investigate a configurable model structure in the context of self-managing storage systems.

To address the limitation and enable model adaptation for target storage systems, we develop a configurable model-based framework with a two-level hierarchy structure: at the lower level, device models are learned to represent the dynamics of individual storage devices and then a higher-level storage platform model is constructed on top of those device models. Using the composition of the learned dynamics, the resulting high-level dynamics model can be thus flexibly constructed to match with various system scales and mixed configurations. With the configurable structure that

provides numerous model variants, we leverage the meta-learning capability of RL agents so that the agents' learned policy (the management strategy) is able to adapt to different system specifications. Our learning procedure includes sampling a set of model variants from the configurable model and using them as environment models that represent storage system specifications, hence meta-training RL agents. We call this novel framework for configurable model-based RL, CoMoRL.

Our work is the first to propose model-based RL with configurability in the domains of self-managing storage in which the system configuration can be changed over time (i.e., different compositional settings of storage devices). The CoMoRL framework allows management policies learned through a set of model variants to adapt to system changes without retraining models and policies, thus enabling to facilitate RL-based zero-touch self-managing systems.

Through experiments, we show that an RL policy trained in CoMoRL achieves robust performance in various storage operation conditions compared to other baseline methods, e.g., with an average performance gain of 0.89%∼5.7% over an IOPS-based heuristic method and 11.8%∼29.7% over a model-based method in volume placement scenarios with various Kubernetes cluster specifications (as demonstrated in Section IV-B2).

The main contributions of this paper are as follows.

- We propose a novel model-based RL framework CoMoRL to support flexible RL adoption for automated operations in dynamic, disaggregated, and heterogeneous storage systems.
- We devise a configurable model structure with a two-level device and platform hierarchy and a set of robust management policies learned through reconfigured model variants for different storage operation conditions.
- We demonstrate several applications of CoMoRL such as volume placement optimization for a container-based virtual cluster and primary affinity control for an object storage cluster, verifying its superiority in zero-shot adaptation to given operating conditions.

The rest of this paper is organized as follows. Section II explains the architecture of a disaggregated storage system and its performance issues with mixed configurations of heterogeneous devices. Section III presents CoMoRL, our proposed model-based RL framework with configurability, and describes how to achieve a robust management policy by using model variants in the framework. Sections IV, V, and VI describe our experiment settings and results, related research works, and conclusions, respectively.

II. DISAGGREGATED STORAGE SYSTEMS

In this section, we present the architecture of a disaggregated storage system for which we adopt RL-based management strategies to establish robust storage performance for system changes and various workloads.

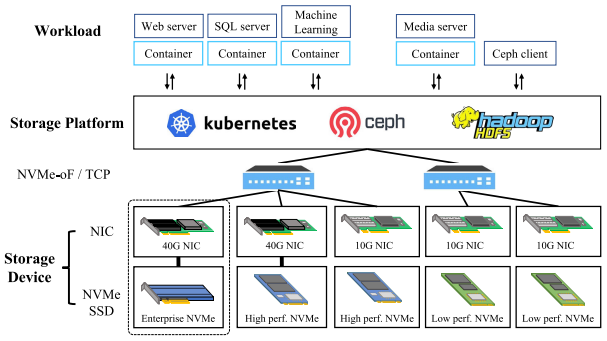


FIGURE 2. Disaggregated storage systems.

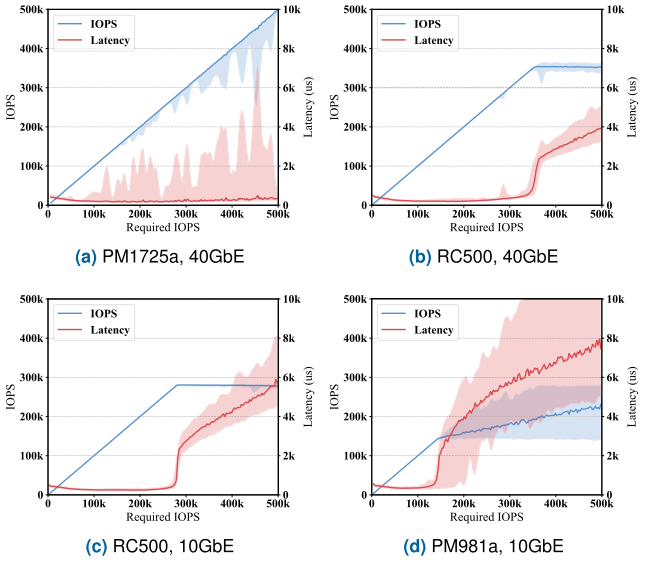


FIGURE 3. Performance patterns of different storage devices. Each corresponds to a specific NVMe-oF node containing different SSDs and network adapters; (a) an enterprise-level device with a Samsung PM1725a SSD and Intel 40GbE XL710-QDA2 NIC shows high throughput (on the left-y-axis) and low latency (on the right-y-axis) over an increasing IOPS request burden (on the x-axis). The examples in (b)-(d) show different performance patterns, e.g., having the bottleneck point on serviced latency at 360K, 270K, and 160K IOPS, respectively. The detailed specifications for these 4 devices are in Table 3.

Figure 2 illustrates the architecture of a disaggregated storage system of NVMe-oF (non volatile memory express over fabrics) devices in which numerous NVMe SSDs are distributed and connected over a data center network. NVMe-oF technology enables NVMe SSDs to operate on top of a network fabric transport (e.g., Ethernet, RDMA) other than a conventional PCIe [20]. It facilitates the separation of computing and storage nodes in a data center, thus allowing for storage disaggregation that offers independent scaling and resource pooling, while enabling low-latency access on remote SSDs. Overall, a storage scale-out structure using NVMe-oF SSDs offers several advantages of storage disaggregation including better resource utilization, rapid system upgrades, cost-efficient maintenance, and flexible configurations. Furthermore, it renders data center operation more flexible and efficient [20], [21], [22].

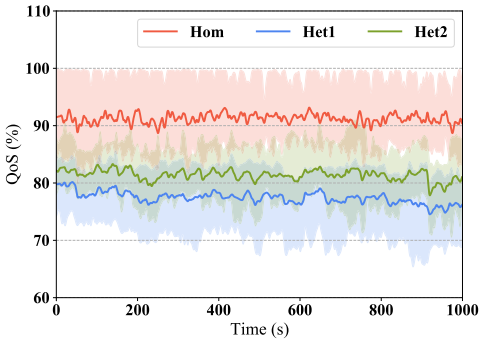


FIGURE 4. Performance of homogeneous and heterogeneous storage systems.

However, such a highly flexible operational strategy for independent storage scaling and upgrades in a heterogeneous storage system (i.e., a cluster of various storage devices with different I/O capabilities) often leads to management challenges in a data center. That is, a storage cluster of heterogeneous devices makes it difficult to optimize the overall performance of both the storage platform (e.g., Kubernetes volume manager, Ceph object storage, HDFS) and each application, because traditional storage platform architectures are rarely able to optimize the performance of a mixed configuration of heterogeneous storage devices with various application requirements [3], [4].

Figure 3 shows the I/O patterns of 4 individual storage nodes each of which has a specific NVMe/TCP SSD device. As shown, we obtain different patterns of IOPS and latency for the different nodes when the same workload is generated by the FIO storage performance benchmarking tool [23]. Figure 3(a) depicts a pattern of enterprise-level storage devices that maintains high serviced IOPS (on the left-y axis) and low latency (on the right-y axis) across increasing workloads (on the x-axis), but the others demonstrate different bottleneck patterns in which the latency increases suddenly at different required IOPS.

Furthermore, in Figure 4, we compare the overall performance, in terms of the application-level quality of service (QoS), achieved by different storage systems. The systems share all the same storage cluster settings except for the individual NVMe-oF nodes. The homogeneous system (Hom) is configured with 4 identical devices (RC500, 10GbE in Figure 3(c)) of mid-range performance. The heterogeneous systems (Het1, Het2) are configured with different devices, but their total I/O capacity is set to be no less than that of the homogeneous system. We intentionally generated intensive workloads for a clear comparison. The two heterogeneous systems yield lower performance than the homogeneous system with a gap of about 10%. This result indicates the unfavorable performance effects of storage heterogeneity, which is what has motivated us to investigate learning-based approaches for heterogeneous storage systems. The QoS metrics that we use are explained in Section IV-A.

III. A CONFIGURABLE MODEL-BASED RL FRAMEWORK

In this section, we describe our proposed framework, CoMoRL configurable model-based RL by which a management strategy based on experiences can be effectively learned for various operation conditions of disaggregated storage systems.

We formulate an RL-based management strategy in an MDP with a tuple $(\mathcal{S}, \mathcal{A}, p, r, \gamma)$. It consists of a state space $\mathcal{S}$, an action space $\mathcal{A}$, a state transition probability $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$, a reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R} \in [0, 1]$, and a discount factor $\gamma \in [0, 1]$. For a storage system, its management strategy is assumed to intend system performance optimization in some given QoS metric during the whole operation period. Accordingly, the reward function is designed based on that QoS definition, and the RL objective is to optimize the overall QoS (e.g., maximize $\sum_t \text{OoS}_t$) for timesteps t through the maximization of the accumulated rewards.

Figure 5 represents the entire structure of the CoMoRL framework with a hierarchy for the storage device model and platform model. The storage device model abstracts the performance patterns of different storage devices from the operation logs collected the FIO benchmark tool [23]. The storage platform model is constructed on top of those learned storage device models to abstract the behavior of the target storage system. The composition of the storage device model, the platform model, and the workload is used as an individual system setting (Conf. k in the figure) that corresponds to a specific simulation environment for some scale and configuration. In CoMoRL, a wide range of different environments can be generated for RL training and they are referred to as model variants ($\mathcal{M}$ in the figure). The management policy achieved by the RL agent trained through the model variants can be highly robust to system changes in practice.

It is worth noting that the storage device and platform models are trained or implemented individually, but the model variants are rendered cost-efficiently without retraining. That model configurability makes RL agents robust against a variety of operating conditions, especially for a target storage system that does not allow for online RL training.

In the following subsections, we explain the configurable model structure of CoMoRL and then describe how it is built using two layered component model types, (1) storage device models in Section III-A1 and (2) storage platform models in Section III-A2. With those, we also present (3) the meta-training procedure for an RL agent in Section III-B, which can establish a robust management strategy.

A. CONFIGURABLE MODELS

In the RL context, model-based approaches are considered promising for system optimization because of their sample-efficient structure and limited interaction with the target system [24], [25]. For a storage system, it is normally not feasible to apply RL algorithms online via direct interaction, but offline operation logs can be leveraged to build

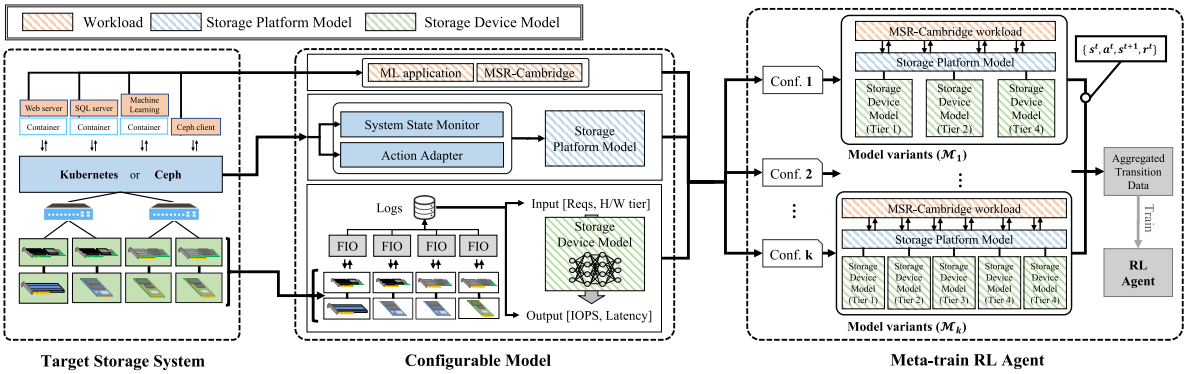


FIGURE 5. The CoMoRL framework.

an environment model that estimates the system dynamics. In general, a dynamics model is formulated as given below where a next state S^{t+1} is yielded for an input pair of state S^t and action A^t at timestep t according to a probability distribution $p(S^{t+1} \mid S^t, A^t)$ conditioned on the input pair. Thus, model learning in model-based RL tends to approximate the distribution $p(\cdot)$,

$$(S^t, A^t) \xrightarrow{p(S^{t+1}|S^t,A^t)} (S^{t+1}). \tag{1}$$

Once a model is established, we can train an RL agent through the model according to the desired management objective of the storage system. That is, a policy or management strategy learned by the agent can be optimized for each specific storage system.

In conventional model-based RL, a model is generally monolithic in that it is fully associated with a specific target system, because it is learned on a dataset of operation logs. In the procedure for model learning, the need to adapt to system changes is not taken into account. This limits the use of model-based RL for disaggregated storage systems, i.e., mixed configurations of heterogeneous devices, in which system changes (scale-in and -out, device upgrades and replacements, etc.) occur frequently. Continuously collecting operation logs for model learning and updating after each sequence of system changes would be a time-consuming and challenging job. As discussed in Figure 1(b), system changes can degrade the performance of model-based RL agents in the absence of retraining procedures.

To address the limitation of model-based RL for storage systems with frequent system changes and configuration updates, we explore a compositional model structure with a two-level hierarchy. Device models are trained on operation logs to represent the dynamics of individual devices, and a storage platform model is constructed based on an aggregation of the device models to represent the dynamics at the application level. This composition enables us to incorporate configuration updates to the target system into a single learned model, thereby facilitating the rapid adaptation of trained RL agents to system changes.

To that end, we have formulated a disaggregated storage system as a combination of individual devices and a platform running on top of those devices. Specifically, we represent a model for a storage system of N objects (an N-sized object set $O_{1:N}$ in a state $S^t = \{O^t_{1:N}\}$ at timestep t) in $\mathcal{D}_{sys}(\cdot)$ using the N-production of an object-wise model $\mathcal{D}_{obj}(\cdot)$. That is,

$$\begin{aligned}\mathcal{D}_{sys}(S^{t+1} \mid S^t, A^t) &= \prod_{i=1}^{N} \mathcal{D}_{obj}(O_i^{t+1} \mid A^t, S^t) \\ &= \prod_{i=1}^{N} \mathcal{D}_{obj}(O_i^{t+1} \mid O_i^t, A^t, O^t_{[\neq i]}).\end{aligned} \tag{2}$$

Given a storage system of M individual devices, we represent a subset of objects $O_{1:N}$ in a partition P_j that corresponds to a specific group of objects that is located and serviced in the jth device, i.e.,

$$P_j = \{O_i \mid O_i \in P_j\}, i \in \{1, 2, \ldots, N\}, j \in \{1, 2, \ldots, M\} \tag{3}$$

Considering that objects located on the same storage device have a larger performance effect on each other than those on different storage devices, we rewrite the model in Eq. (2) as

$$\begin{aligned}&\prod_{i=1}^{N} \mathcal{D}_{obj}(O_i^{t+1} | O_i^t, A^t, O^t_{[\neq i]}) \\ &\quad= \prod_{i=1}^{N} \mathcal{D}_{obj}(O_i^{t+1} | O_i^t, A^t, \{O_k^t | O_k^t \in P_j^t\}) \\ &\quad= \prod_{i=1}^{N} \mathcal{D}_{obj}(O_i^{t+1} | P_j^t, A^t) = \prod_{j=1}^{M} \mathcal{D}_{str}(P_j^{t+1} | P_j^t, A^t)\end{aligned} \tag{4}$$

where $\mathcal{D}_{str}(\cdot)$ corresponds to the dynamics model of a storage device, which is explained in Section III-A1 below. By Eq. (2)-(4), we establish that the overall system dynamics $\mathcal{D}_{sys}(\cdot)$ can be modeled based on the implementation and combination of $\mathcal{D}_{str}(\cdot)$.

In the following, we describe how to achieve the storage device model $\mathcal{D}_{str}(\cdot)$ and how to combine it with a known

platform model so that the overall system model $\mathcal{D}_{sys}(\cdot)$ can be implemented.

1) STORAGE DEVICE MODEL

Given the dynamics representation of a storage device $\mathcal{D}_{str}(\cdot)$ in Eq. (4), we decompose it into two individual parts, the dynamics of workloads and the dynamics of actions. Accordingly, we rewrite the model for storage devices as

$$\begin{aligned}
&\prod_{j=1}^{M} \mathcal{D}_{str}(P_j^{t+1} | P_j^t, A^t) \\
&\quad = \prod_{j=1}^{M} \sum_{\tilde{P}^t \subset S} \mathcal{D}_{str}(P_j^{t+1} \mid \tilde{P}^t) * \mathcal{D}_{act}(\tilde{P}^t \mid P_j^t, A^t) \\
&\quad = \prod_{j=1}^{M} \mathcal{D}_{str}(P_j^{t+1} \mid \tilde{P}_j^t) \qquad (5)
\end{aligned}$$

where $\tilde{P}^t$ denotes a random variable for partition after an action A^t is applied, and $\mathcal{D}_{act}(\tilde{P}^t | P_j^t, A^t)$ denotes the partial dynamics influenced only by actions. We assume that the actions we consider for storage management scenarios, such as migrating data and setting system parameters, are guaranteed to execute. This is because given a target management scenario, we consider only valid actions in its RL context that can be interpreted and performed as a sequence of executable system commands.

Next, we explain how to implement $\mathcal{D}_{str}(P_j^{t+1} \mid \tilde{P}_j^t)$. For each device, we first collect operation logs including the pair sets of $X = \tilde{P}_j^t$ and $Y = P_j^{t+1}$, where X represents the required IOPS on objects $i \in P_j$ and Y represents the respective serviced IOPS and latency for X. Then, with a sufficient dataset of logs, it is possible to learn a regression model that can predict the serviced IOPS and latency of an object set associated with partition P_j. In our notations, the followings are used for ith object state o_i^t at timestep t.

- $o_i^t.TI$ and $o_i^t.TL$ denote the required IOPS and latency of ith object, respectively.
- $o_i^t.SI$ and $o_i^t.SL$ denote the serviced IOPS and latency of ith object, respectively.

Algorithm 1 represents how to train such a regression-based storage device model $\mathcal{D}_{str}(P_j^{t+1} \mid \tilde{P}_j^t)$, where $j \leq M \leq$. In lines 3-7, operation logs are collected using FIO running on an individual storage device j. In doing so, each object $o_i \in P_j$ in the device is specified with its IOPS request $o_i.TI$. To generate FIO flows over multiple objects with different IOPS requests simultaneously, we use a single FIO job description file that contains a set of options commonly used for multiple flows. The options include block size, IO depth, and request type. In addition, to adjust each flow for an individual object, we use a configurable option `iops_rate` that represents the IOPS request for the object, which is randomly set in line 5. As a result of FIO execution in line 6, the logs of serviced IOPS and latency $(IOPS_{fio}, LAT_{fio})$ are collected and added to data buffer $\mathcal{B}$. This data collection iterates until $\mathcal{B}$ is full.

Algorithm 1 Storage Device Model Training

```
/* P_j : Set of objects stored in jth
   device, 1 ≤ j ≤ M,
   o_i : State of ith object,
   {TI, TL, SI, SL},
   B : Buffer to store operation
   logs,
   size : Maximum size of B,
   max_iops : Maximum IOPS request
   for an object,
   θ : Parameters of storage device
   model D_str,
   e : Number of epochs                */
1  Initialize B, θ
2  for |B| < size do
3      Device ID j ← randomInt(1, M)
4      for object o_i ∈ P_j do
5          o_i.TI ← randomInt(0, max_iops)
6      IOPS_fio, LAT_fio ← FIO (P_j, j)
7      B ← B ∪ {(P_j, j, IOPS_fio, LAT_fio)}
8  for epoch ∈ [1, e] do
9      for (P_j, j, IOPS_fio, LAT_fio) ∈ B do
10         IOPS, LAT ← InferenceStorage (P_j, j)
11         L ←
             ||LAT − LAT_fio||_1 + ||IOPS − IOPS_fio||_∞
12         θ ← θ - ∇_θ L
```

In lines 9-12, a regression model is trained on the logs in $\mathcal{B}$ using `InferenceStorage()` in Algorithm 2 and two losses, the L1 loss for serviced latency and the L-infinity loss for serviced IOPS. Here, for different device specifications, we use a single deep neural network (DNN) for model learning rather than a set of individual DNNs. In our experiments, a single model trained on various specification datasets turned out to be robust against the difference of learning and target systems.

Algorithm 2 represents how to use the storage device model $\mathcal{D}_{str}(\cdot)$ to infer the next state P_j^{t+1} including the serviced IOPS and latency, upon an input P_j that is the current state $\tilde{P}_j^t$ after action execution. We represent the input features in the form of histograms in which objects are grouped according to the range of required IOPS using the interval l in lines 3-4. For example, an object of 13,000 requests is represented in the $(13000/l)$th region. In our implementation, the maximum IOPS is set to 15000 and l is set to 1000. Accordingly, the input features are represented as a histogram with 15 regions. In line 5, the $\mathcal{D}_{str}(\cdot)$ model itself infers the

Algorithm 2 Storage Device Model Inference

```
/* P_j : Set of objects stored in jth device,
   o_i : The state of ith object,
   {TI, TL, SI, SL},
   l : IOPS interval in a single range,
   Count : Histogram list for o_i.TI */
def InferenceStorage (P_j, j):
    Initalize Count
    for object o_i ∈ P_j do
        Count[o_i.TI/l] ← Count[o_i.TI/l] + 1
    IOPS, LAT ← Predict (Count, j)
    return IOPS, LAT
```

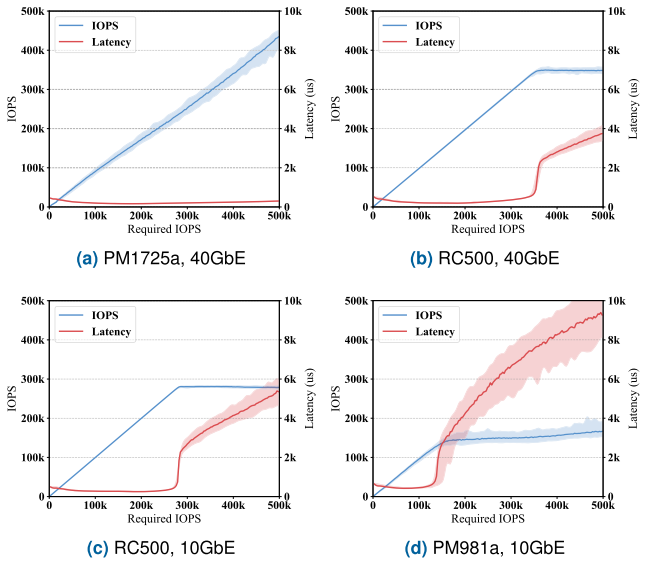


FIGURE 6. Performance patterns estimated by our learned storage device model. Each figure shows the predicted performance pattern for each specific device specification in Figure 3 and Table 3.

predicted serviced performance in IOPS and latency. The serviced IOPS (*IOPS*) is a vector of the same size as the input, representing the ratio for each required IOPS. The serviced latency (*LAT*) is a single value, as the average latency of all requests sent to the same device is assumed to be equal.

Figure 6 shows the inference outputs of our learned storage model for several devices, confirming that they are consistent with real measurements in Figure 3.

2) STORAGE PLATFORM MODEL

A storage platform model is used to integrate the state information of individual partitions in a unified state, as illustrated in Eq. (6).

$$\begin{array}{l} (P_1^t, A^t) \rightarrow \tilde{P}_1^t \xrightarrow{\mathcal{D}_{str}(P_1^{t+1}|\tilde{P}_1^t)} P_1^{t+1} \\ (S^t, A^t) \rightarrow (P_j^t, A^t) \rightarrow \tilde{P}_j^t \xrightarrow{\mathcal{D}_{str}(P_j^{t+1}|\tilde{P}_j^t)} P_j^{t+1} \rightarrow S^{t+1} \\ (P_M^t, A^t) \rightarrow \tilde{P}_M^t \xrightarrow{\mathcal{D}_{str}(P_M^{t+1}|\tilde{P}_M^t)} P_M^{t+1} \end{array} \tag{6}$$

The storage platform model first decomposes the whole object set into a set of partitions $P_j (j = 1, \ldots, M)$ based on the object location for the state, and then it aggregates the inference outputs by the storage device model for all j. Algorithm 3 implements this procedure, where the next state of M partitions is inferred through Algorithm 2.

Algorithm 3 Storage Platform Model Inference

```
/* S : Entire object set,
   a : Action from agent,
   P_j : Set of objects stored in jth device,
   o_i : The state of ith object,
   {TI, TL, SI, SL}                          */
/* ExecuteAction(), GetState(),
   GetReward() are scenario-specific functions */
def Inference (a):
    ExecuteAction(S, a)
    for Device ID j ∈ [1, M] do
        IOPS, LAT = InferenceStorage(P_j, j)
        for object o_i ∈ P_j do
            o_i.SI = IOPS[o_i.TI/l], o_i.SL = LAT
    State s, Reward r = GetState(S),
     GetReward(S)
    return s, r
```

Note that `ExecuteAction()` is responsible for action executions whose implementation is not part of our framework specification. Their implementation depends on a given management scenario and target platform. For instance, a specific data relocation action between NVMe-oF devices can be translated into appropriate platform commands and executed. Similarly, `GetState()` and `GeReward()` correspond to the transition and reward functions in conventional RL formulations, and they are also implemented according to the management scenario. Several examples of these scenario-specific functions are discussed and their implementation is presented in Section IV.

In line 2, `ExecuteAction()` updates the state of each partition, i.e., $(P_j^t, A^t) \rightarrow \tilde{P}_j^t$ in Eq. (6), in the storage platform model. Then, in line 4, `InferenceStorage()` (in Algorithm 2) performs the inference of the next state of each partition, i.e., $\tilde{P}_j^t \rightarrow P_j^{t+1}$ in Eq. (6). In lines 5-6, for each object $o_i \in P_j$, its serviced IOPS ($o_i.SI$) and latency ($o_i.SL$) are updated by the storage device model using the inference outputs. This iteration updates the state of the entire object set $S = \{P_{1:M}\} = \{O_{1:N}\}$ and aggregates the state of each partition P_j. Once S is updated, in lines 7-8, the next state and reward values are calculated and returned according to scenario-specific state representation and reward shaping functions.

As such, Algorithm 3 can be seen as a conventional step function in RL, e.g., next state s, reward r = `step` (action a), that takes an action as input and returns a next state and a reward as output.

B. TRAINING AN RL AGENT

Using our configurable model to provide a flexible simulation environment, we train an RL agent to establish learning-based storage management strategies. In particular, we consider the temporal changes and variants of a target storage system and so devise a meta-training procedure with a set of differently configured model variants.

Algorithm 4 Training an RL Agent

```
/* M : Model variant,
   B_agent : Replay Buffer,
   θ_agent : Parameters of RL,
   π_θagent : Policy of RL,
   T : total timesteps,
   Conf : System configuration */
1  Initialize θ_agent, B_agent
2  for t ∈ [1, T] do
3      if system is changed then
4          Model variant M ← Configure (Conf)
5          Initialize s^t from M
6      Action a^t ~ π_θagent(s^t)
7      s^{t+1}, r^t ← M.Inference(a^t)
8      B_agent ← B_agent ∪ {s^t, a^t, r^t, s^{t+1}}
9      for {s^t, a^t, r^t, s^{t+1}} ∈ B_agent do
10         Loss L ← RLLoss(π_θagent)
11         θ_agent ← θ_agent - ∇_θagent L
```

As specified in Algorithm 4, the meta-training procedure uses domain randomization (DR) [26], [27] by which an agent is trained through model variants $\mathcal{M}$. Each variant is created according to a system specification that is configured by the device group, storage platform, and workload type. In line 4, `Configure()` is implemented to render a model variant for a specific configuration. For example, for a Ceph storage platform configured to have a group of 4 different devices with the MSR workload, `Configure()` produces the specific model variant for the device group setting, platform and workload type. For simple algorithm representation, we assume that a system continuously varies with different configurations and each one is specified internally in *Conf*. Given a model variant, in lines 5-6, the platform model inference `Inference()` in Algorithm 3 is used to predict the next state and reward when the agent's action a^t is applied. This prediction generates transitions (s^t, a^t, r^t, s^{t+1}) for training the RL agent $\pi_{\theta_{agent}}$.

In our framework implementation, Transformer [28] is used to train the agent ($\pi_{\theta_{\text{agent}}}$ in Algorithm 4), so variable system scales can be handled. An M-length vector for partitions $\{P_1, \ldots, P_M\}$ is used for input to the transformer encoder, and the transformer decoder returns actions, where M is a variable for dynamic scaling. Algorithms 1-3 together establish model variants $\mathcal{M}$ that can be readily configured for training the RL agent in Algorithm 4. That model configurability facilitates the meta-learning of the agent, thereby allowing the agent to adapt to different target system settings.

IV. EVALUATION

In this section, we evaluate the CoMoRL framework. Specifically, we adopt the framework for two storage application scenarios such as virtual object placement for Kubernetes volume management and primary affinity control for Ceph object storage, evaluating the performance of RL policies learned in the framework across various system operation conditions.

A. EXPERIMENT SETTINGS

For comparison, we implement the following baseline algorithms in addition to our CoMoRL.

- **IOPS-based:** this algorithm continuously adjusts the total amount of IOPS on each storage device to ensure that the required IOPS remains under some threshold. Several works for storage performance optimization have used IOPS-based heuristics [3], [10], but they did not consider application-level QoS and system changes. Thus, we test our own simple IOPS-based heuristics for comparison purposes. For each device, we first establish the bottleneck point at which the latency starts to increase dramatically based on its operation logs (Figure 4), and then use that point to specify the threshold, such as e.g., a 10% margin from the bottleneck point.
- **DR:** this algorithm is based on domain randomization techniques, in which randomly generated system configurations are used to train an RL agent. It is intended to make RL agents robust to system changes.
- **Model-based:** this is a conventional model-based RL algorithm that learns a model from operation logs for a specific system. Unlike our proposed configurable model scheme, it uses the monolithic architecture. Using

TABLE 1. The hyper-parameter settings of RL.

Hyper-parameter	Value
Learning algorithm	SAC
Optimizer	Adam
Discount factor	0.95
Steps per episode	1000
Total training episodes	10,000
Learning rate	0.0005
Training frequency	1
Replay buffer size	500,000
Mini-batch size	256
Num. of Transformer layers	3
Average required latency	2500

the collected operation logs of each target system, we generate a respective model and train an RL agent for each system.

- **Autotiering** [3]: this is a state-of-the-art heuristic method to allocate virtual machine disk files in a multi-tier all-flash data center. It is intended to maximize the performance and utility of a data center by estimating the gain of specific file relocation plans through regression. Autotiering considers IOPS throughput as performance metrics and focuses on maximizing IOPS and minimizing latency, but our framework focuses on specific user requirements in QoS such as Eq. (7) and (8) which need to consider the required latency specification.

In RL implementation, we use SAC [29] with the Adam optimizer. Table 1 lists the hyper-parameter settings for training RL agent of DR, Model-based and ours.

For workload emulation on storage systems, we use two datasets.

- **MSR workload:** this contains a set of object-based workloads based on the MSR-Cambridge I/O trace dataset [30], which uses 7 days of operation logs of request time, logical address, object size, and type information from 13 servers of the MS data centers.
- **ML workload:** this contains various machine learning (ML) workloads generated by a few well-known ML models, which are considered to represent the majority of modern data centers. Table 2 lists the ML models with their datasets, batch sizes, and measured I/O throughput (TP(MB/s)).

As performance metrics, we consider the application-level QoS, which is averaged for all requests on objects $i = 1, \ldots, N$. Rather than focusing on latency minimization, we intend to handle various data-intensive applications with different requirements in I/O throughput and latency-limit. Therefore, we formulated our QoS metrics by compositing the throughput and latency requirements in the following form.

$$\mathrm{QoS}_{sys}(t) = \frac{1}{N}\sum_{i=1}^{N}\mathrm{QoS}(i,t)$$
$$\mathrm{QoS}(i,t) = \begin{cases} 1, & if SI_i^t >= \alpha * TI_i^t and SL_i^t <= \beta * TL_i^t \\ 0, & otherwise \end{cases} \quad (7)$$

TABLE 2. Configuration of ML workloads.

Dataset	Model	Batch	TP(MB/s)
VisDrone [31]	yolov5s [32]	8	17.02
		16	18.15
		32	16.68
	yolov5n	8	17.52
	yolov5m	8	10.43
	yolov5l	8	7.27
Coco [33]	yolov5s	8	42.26
		16	50.91
		32	58.35
	yolov5n	8	52.44
	yolov5m	8	26.11
	yolov5l	8	17.65
ImageNet [34]	EfficientNet -B0 [35]	8	21.65
		16	39.23
		32	45.65
	EfficientNet -B2	8	10.67
		16	14.18
		32	19.44
	EfficientNet -B4	8	4.106
		16	4.69
		32	4.84

Here, the per-object $\mathrm{QoS}(i,t)$ yields 1 when both the IOPS and latency requirements, TI_i^t and TL_i^t, respectively, are satisfied at timestep t and is 0, otherwise. Note that α and β represent the weights for IOPS and latency requirements, respectively, and SI_i^t and SL_i^t denote the serviced IOPS and latency, respectively.

Regarding the generality of our framework, we seek to render the framework agnostic to particular QoS specifications, because it is feasible to incorporate user-custom QoS metrics into the learning objectives of RL, as long as they can be measured online. Therefore, the following metric in a generalized form is also tested.

$$\mathrm{QoS}^{+}_{sys}(t) = \frac{1}{N}\sum_{i=1}^{N}\mathrm{QoS}^{+}(i,t)$$
$$\mathrm{QoS}^{+}(i,t) = \alpha^{+} * \frac{SI_i^t}{TI_i^t} + \beta^{+} * \mathrm{clip}(1 - \frac{SL_i^t - TL_i^t}{2 \times TL_i^t}, 0, 1) \quad (8)$$

The per-object $QoS^{+}(i,t)$ can be differently defined in terms of the respective strictness of IOPS and latency requirements. The hyperparameters α^{+} and β^{+} are used to enable a higher α^{+} setting for low-latency applications and a higher β^{+} setting for IOPS-intensive applications.

B. VOLUME PLACEMENT SCENARIO

Using a Kubernetes server cluster running in our lab, we tested a data placement scenario in which each container was associated with its persistent volume. In this scenario, the management strategy accounts for the optimal placement of volumes on a set of storage tiers, each of which consists of NVMe/TCP (NVMe-over-TCP) SSDs of the same specification and capability. This scenario is similar to Autotiering [3] in terms of its management mechanism, in which virtual machine disk files are relocated to a tiered storage system

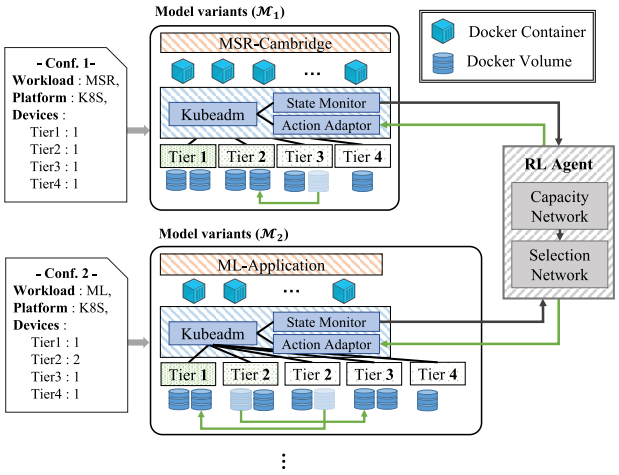


FIGURE 7. Volume placement scenario in a Kubernetes cluster.

TABLE 3. The specifications of storage tiers.

Tier	SSD	Read(MB/s)	TP(IOPS)	NIC	GbE
1	PM1725a	6400	1080K	XL710-QDA2	40
2	RC500	1700	290K	XL710-QDA2	40
3	RC500	1700	290K	X710-DA2	10
4	PM981a	3500	460K	X710-DA2	10

in the all-flash storage data center. In particular, Autotiering uses heuristic algorithms for VMDK placement based on several I/O performance features, thereby optimizing the I/O throughput and latency. Although our work uses a similar mechanism for tiered storage management, the volume management by CoMoRL optimizes a given specific QoS defined in the form of application-specific performance constraints on both throughput and latency (i.e., Eq. (7) or (8)).

Figure 7 illustrates the volume placement function with our Kubernetes cluster where multiple NVMe/TCP devices are categorized as different tiers, e.g.,, Tier 1,..., Tier 4, and used to manage container volumes. In addition to the CoMoRL framework, we implement several modules including a state monitor and an action adaptor. The former aggregates the state information of average IOPS throughput and latency via Kubeadm, sending it to the RL agent for sequential decision-making on volume placement. The latter translates the actions of the RL agent into volume migration commands for different storage tiers.

1) IMPLEMENTATION

We set up our small testbed with Kubernetes version 1.24, where 3 storage nodes and 1 worker node operate. In the testbed, each storage node containing some specific SSDs and a network interface is categorized in one of the storage tiers listed in Table 3 according to its I/O capability. The storage tiers categorize volumes hierarchically according to a given volume management strategy. For workload generation, we execute the FIO container [36] with 4KB-sized blocks on the worker node, which runs on a system of an AMD Threadripper 2995WX with 32 cores and 128GB RAM.

For training an RL agent for volume placement, we develop the two-staged algorithm with (1) a capacity network that determines the throughput threshold of each storage device and (2) a selection network that determines the candidate volumes that will be migrated when the IOPS request exceeds the throughput threshold. In the following, we explain the scenarios-specific function implementation for `GetState()`, `GetReward()` and `ExecuteAction()`, which are specified in Algorithm 3. Note that the implementation of these functions allows the CoMoRL framework to be used for a specific management scenarios.

Algorithm 5 ExecuteAction (in Volume Placement)

```
/* S : Entire volume set,
   a : Action from agent (capacity
   and selection networks),
   P_j : Volumes in jth device        */
def ExecuteAction(S, a):
    capaAction, selAction = a
    for j ∈ [1, M] do
        k ← 0
        while capaAction_j < P_j.TI and k < ρ do
            if selAction_k > 0 then
                Move o_k to High-perf. tier
            else
                Move o_k to Low-perf. tier
            k ← k + 1
            P_j.TI ← P_j.TI − o_k.TI
```

a: STATE

`GetState()` is implemented to produce state information for both capacity and selection networks. Specifically, the state for the capacity network $capaState^t$ includes integrated information for the M-sized partition set,

$$capaState^t = \sum_{j=1}^{M} \{capaState_j^t\} = \sum_{j=1}^{M} \{|P_j|,\ P_j^t.TI,\ P_j^t.SI,\ P_j^t.TL,\ P_j^t.SL\} \quad (9)$$

where the number of objects in partition P_j, the total required IOPS $P_j^t.TI = \sum_{i \in P_j} o_i^t.TI$, the total serviced IOPS $P_j^t.SI = \sum_{i \in P_j} o_i^t.SI$, the average maximum required latency $P_j^t.TL = \frac{1}{|P_j|} \sum_{i \in P_j} o_i^t.TL$, and the average serviced latency $P_j^t.SL = \frac{1}{|P_j|} \sum_{i \in P_j} o_i^t.SL$ are calculated individually.

The state of the selection network contains a ρ-length list in which each element specifies the information related to P_j,

$$selState_j^t = \{capaState_j^t, capaAction_j^t\} + \sum_{o_i \in P_j, o_i.TI \geq TI_\rho} \{o_i.TI, o_i.SI, o_i.TL, o_i.SI\}. \quad (10)$$

The first term contains $capaState_j^t$ which is used as input to the capacity network and the output $capaAction_j^t$ which is produced by the capacity network. Note that $capaAction_j^t$ specifies the throughput threshold of jth device (the device for partition P_j) and it is calculated by the capacity network. The second term refers to the state of ρ volumes for each P_j, which involves required IOPS $o_i.TI$, serviced IOPS $o_i.SI$, maximum required latency $o_i.TL$, and serviced latency $o_i.SL$ of ith volume. For efficient processing, the selection network considers only the top-ρ volumes with high required IOPS as the candidates to be migrated. In Eq. (10), TI_ρ denotes the required IOPS (TI) of the volume with the ρth highest required IOPS. In our implementation, ρ is set to 10.

b: REWARD

`GetReward()` is implemented to yield the average QoS in Eq. (7) or (8) which is based on performance data measured in the state monitor.

c: ACTION

`ExecuteAction()` conducts scenario-specific action executions, i.e., conducting volume migration across tiers. It takes the outputs of both the capacity and selection networks as input, and determines candidate volumes to be migrated. Algorithm 5 implements the volume placement procedure using the capacity and selection actions $capaAction^t$ and $selAction^t$. In lines 4-9, when the threshold for P_j ($capaAction_j \leq P_j.TI$) is not satisfied, some volumes in P_j are migrated to devices in tiers other than the tier of P_j. Specifically, ρ volumes are selected for migration using top-ρ-rankings, $selAction_j^t = selAction_{j,1}^t, \ldots, selAction_{j,\rho}^t$, where each value is set to $[-1, 1]$. In lines 6-9, this value is used to indicate the migration direction toward either high-performance tiers or low-performance tiers. The capacity and selection networks are jointly trained with this scenario-specific `ExecuteAction()` to maximize QoS over time.

2) EXPERIMENT RESULTS

Figure 8 represents the QoS in Eq. (8) achieved upon system changes and mixed configurations, where Conf. k denotes a specific system configuration of our testbed. The configurations, which have different tiered storage settings and object sizes, are listed in Table 4, where tiers are specified by NVMe/TCP SSD types in Table 3. The various Conf. k emulates temporal system changes, e.g., scale-in and -out, and evaluates the robustness of our framework. Conf.0 is a normal configuration in which the storage system is set to be sufficient to handle all given requests. As expected, no significant performance difference is observed under such normal circumstances. The others, Conf.1-11, are set to have excessive requests, and some of them (Conf.10* and Conf.11*) are unseen configurations that the agent did not experience in training. While Default supports no migration, the other baseline methods, IPOS-based, DR, and Model-based, use migration strategies. The IPOS-based method employs a heuristic rule as described in Section IV-A, and the DR and Model-based methods employ RL-based learned strategies.

As shown, under the MSR workload, the RL agent (**CoMoRL**) trained using our framework achieves higher QoS consistently for all Conf.1-11, e.g, 0.7∼5.1%, 1.7∼8.1%, and 11.8∼29.7% higher than the IOPS-based, DR, and Model-based methods, respectively. Specifically, for Conf.1-9, in Figure 8(a), **CoMoRL** shows 2.82%, 4.04%, and 17.68% higher average QoS than the IOPS-based, DR, and Model-based methods, respectively. For unseen Conf.10*-11*, furthermore, **CoMoRL** shows 0.75%, 2.09%, and 22.9% higher average QoS than those baseline methods, respectively. Under the ML workload, in Figure 8(b), **CoMoRL** shows 1.91%, 2.60%, and 9.78% higher average QoS than the IOPS-based, DR, and Model-based methods, respectively. For unseen Conf.10*-11*, **CoMoRL** shows 0.8%, 3.14%, and 17.35% higher average QoS than those baseline methods, respectively. These results demonstrate that our approach is robust to system changes under different workloads. The superiority is achieved because **CoMoRL** provides an efficient mechanism for restructuring models specific to different system scales and configurations, which enables meta-training of the RL agents with model variants. Interestingly, the Model-based method shows lower performance than the others. Conventional model-based RL techniques require sufficient samples to learn the model for each system configuration. In our case where it is difficult to collect sufficient samples due to a wide variety of configurations, model learning easily ends up with underfitting performance. This clarifies the benefit of our configurable model.

To discuss the robustness of **CoMoRL** in a statistical way, we present the average QoS with 95% confidence intervals of each baselines methods in Figure 9 and Table 5. For this comparison, we perform iterative tests on the Conf.1 setting in Table 4 with MSR and ML workloads. As shown, our **CoMoRL** achieves not only a higher average QoS than the other methods but also maintains a lower variance (i.e., 95% confidence intervals of ±1.02% for MSR workload and ±1.29% for ML workload).

In Figure 10, we test different QoS specifications for the evaluation metric, where Eq. (7) is used in (a), Eq (8) is used in (b), and the reciprocal of average latency is used in (c). In using such specific QoS metrics, we reformulate the reward function accordingly. As shown, our approach outperforms the others for all cases, achieving higher QoS in (a) and (b) and lower latency in (c). These results indicate that **CoMoRL** is generalized and extensible to different objectives and optimization scenarios to some extent. In principle, several RL agents can be differently trained for given objectives with a single set of model variants. The decoupled structure of

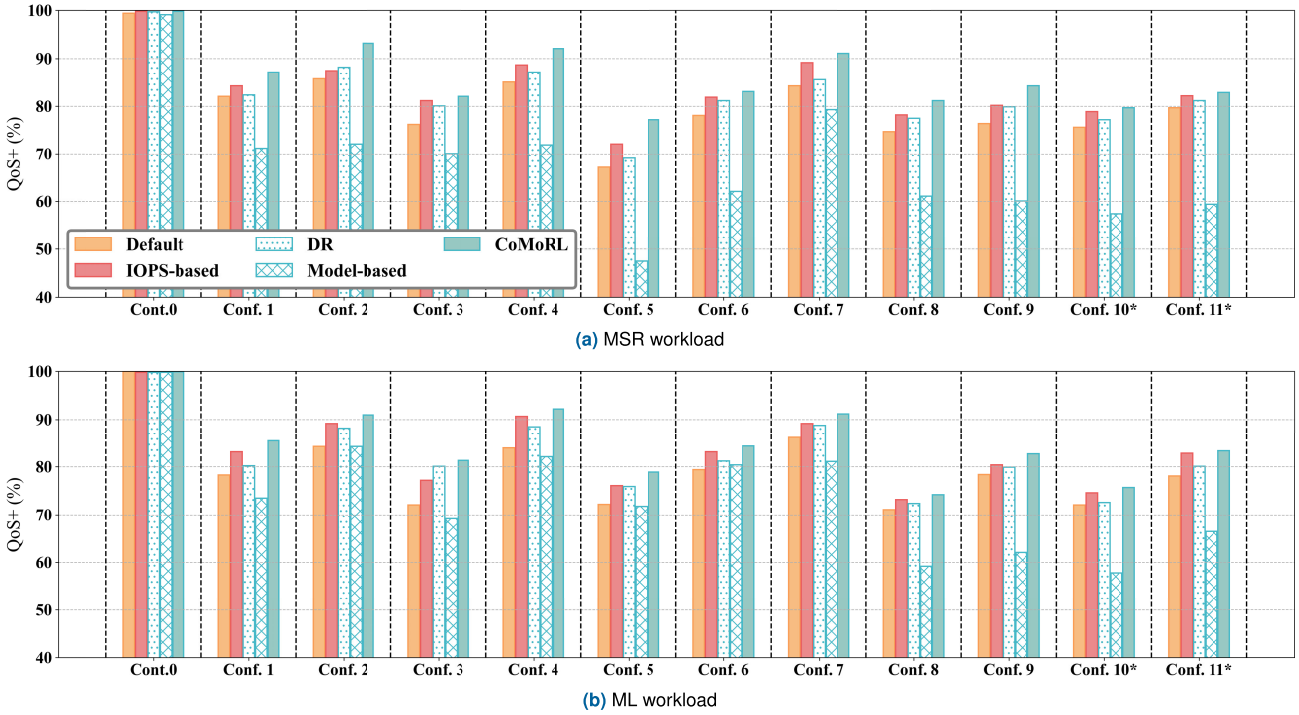


FIGURE 8. Performance in volume placement scenario.

TABLE 4. Storage configurations for volume placement scenario. Each Conf. k corresponds to a configuration tested in Figure 8 with different settings. For example, Conf. 4 is set to have 1, 1, 2, and 1 devices of tier 1, 2, 3, and 4, respectively. In addition, Volume (a) denotes the total number of objects handled in (a) the MSR workload and Volume (b) denotes the total number of objects handled in (b) the ML workload.

-	Conf. 0	Conf. 1	Conf. 2	Conf. 3	Conf. 4	Conf. 5	Conf. 6	Conf. 7	Conf. 8	Conf. 9	Conf. 10*	Conf. 11*
Tier 1	1	1	1	1	1	1	1	1	1	1	1	1
Tier 2	1	1	2	2	1	1	1	2	2	1	1	1
Tier 3	1	1	1	0	2	1	2	2	1	2	1	3
Tier 4	1	1	1	0	1	1	2	1	1	2	3	1
Volume (a)	300	500	500	500	500	750	750	750	1000	1000	1000	1000
Volume (b)	30	50	50	50	50	75	75	75	90	90	90	90

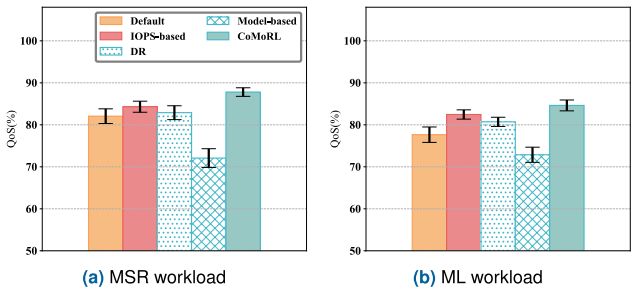


FIGURE 9. Statistical analysis of CoMoRL.

TABLE 5. Statistical analysis of CoMoRL.

MSR workload				
Default	IOPS-based	DR	Model-based	CoMoRL
82.5±1.74%	84.3±1.31%	82.89±1.65%	72.08±2.23%	**87.8±1.02%**
ML workload				
Default	IOPS-based	DR	Model-based	CoMoRL
77.65±1.83%	82.46±1.1%	80.71±1.09%	72.86±1.8%	**84.62±1.29%**

configuring environment models and RL training allows for multiple agents that are differently optimized without model retraining.

In Figure 11, to evaluate the predictability in performance provided by our approach, we check the serviced latency of different requests under some confidence, i.e., 90%. The requests are characterized as (a) Critical with a latency constraint of 500us and (b) Non-critical with no latency constraint. In (a), the red line represents the latency constraint. As shown, CoMoRL manages to keep the serviced latency (SL) much closer to the latency constraint than others in (a) Critical. CoMoRL is intended to meet the required latency of each object, without necessarily reducing the overall serviced latency. Accordingly, in (b) Non-critical, CoMoRL shows higher latency than the others.

In Figure 12, to confirm the stability of CoMoRL with respect to various user requirements, we evaluate the QoS

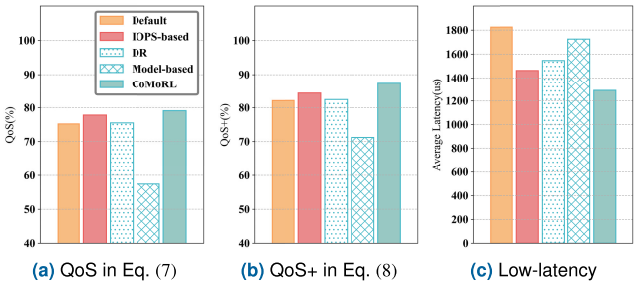


FIGURE 10. Performance with different objectives.

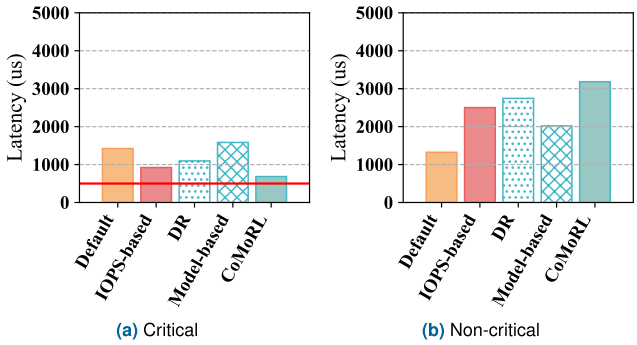


FIGURE 11. Serviced latency under confidence.

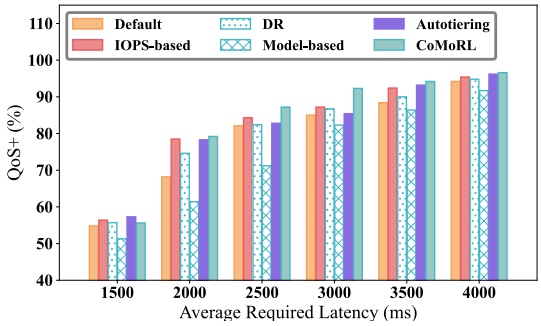


FIGURE 12. Performance with required latency specifications.

performance across different required latency specifications (i.e., 1500∼4000ms). In this experiment, we add Autotiering [3] to our baselines and observe that Autotiering experiences lower performance due to the fact that the required latency is not considered. Indeed, our previous experiment results do not include Autotiering, as it consistently shows poor performance similar to the IOPS-based method that also does not consider the required latency. In Figure 12, CoMoRL achieves 1.81%, 3.48%, 10.13%, and 1.98% higher QoS than IOPS-based, DR, Model-based, and Autotiering methods, respectively. When the required latency is too low or high (i.e., the requirement is too tight or loose), the performance gain of CoMoRL decreases. In the other range, i.e., 2000∼3500ms, the gain increases.

C. PRIMARY AFFINITY SCENARIO

To evaluate the effectiveness of CoMoRL in practice, we implement and test an autonomous management scenario

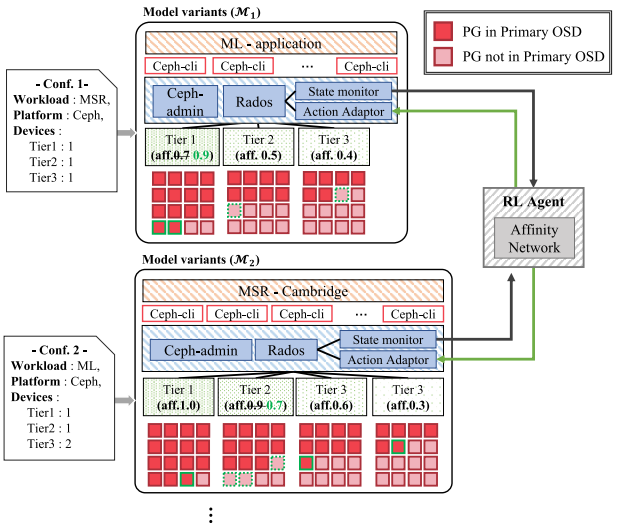


FIGURE 13. Primary affinity scenario in a Ceph cluster.

TABLE 6. The specification of OSD tiers.

Tier	Device	Read(MB/s)	TP(IOPS)	NIC	GbE
1	RC500	1700	290K	XL710-QDA2	40
2	RC500	1700	290K	X710-DA2	10
3	WD Blue	560	95K	X710-DA2	10

with a Ceph storage cluster in our lab. Figure 13 illustrates such a scenario in which the primary affinity value is adjusted continuously as part of the self-tuning operation in Ceph. In Ceph, a data distribution algorithm CRUSH [37] is responsible for managing placement groups (PGs) that are located on object storage daemons (OSDs). A set of objects is grouped as a PG, and a predetermined number of replicas of each PG are stored on several OSDs. Among several OSDs holding replica of each, CRUSH chooses one primary OSD to be responsible for handling requests to its associated PGs.

Specifically, the *primary affinity* value represents the probability that an OSD will be chosen as the primary OSD, and it is initialized as 1. By modifying the *primary affinity* manually, it is possible for a Ceph administrator to redistribute request loads over multiple OSDs. By default, all *primary affinity* values are set to 1, and thus a uniformly random distribution is normally expected. If the number of requests on an OSD becomes too large, it is desirable to lower its primary affinity value.

In this test, we adopt RL-based strategies to automate the primary affinity control. In doing so, we implement the state monitor and action adaptor modules to connect our framework to the Ceph cluster. The state monitor aggregates the state information about OSDs and PGs, and the action adaptor translates actions from the RL agent into respective RADOS commands for setting primary affinity values.

1) IMPLEMENTATION

To test the primary affinity scenario, we set our Ceph Octopus cluster, where librados [38]-based clients run on a system of

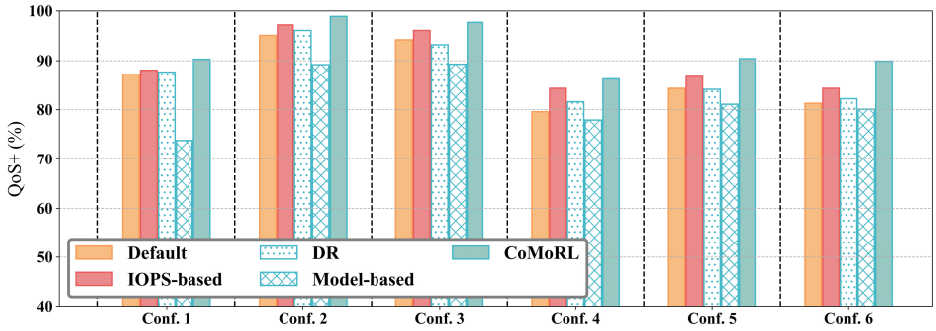


FIGURE 14. Performance in primary affinity scenario under the MSR-Cambridge workload.

an AMD Threadripper 2995WX with 32 cores and 128GB RAM. Each client sends I/O requests to the primary OSD and receives 4MB-sized objects. Similar to the volume placement scenario, OSDs are associated with tiers defined in Table 6. Next, we explain the implementation of scenario-specific functions.

a: STATE

`GetState()` is implemented to produce the state information for an RL agent's network, namely the affinity network. The state $affState^t$ includes

$$affState^t = \sum_{j=1}^{M} \{|P_j|,\ aff_j^t,\ P_j^t.TI,\ P_j^t.SI,\ P_j^t.TL,\ P_j^t.SL\}, \tag{11}$$

which is similar to Eq. (9), where aff_j^t denotes the primary affinity value of jth OSD. Unlike the two-staged agent with capacity and selection networks in the volume placement scenario, we use a single network for the RL agent, which produces the primary affinity values of M OSDs. Because CoMoRL provides a configurable model for training agents, RL algorithms and the agent structure can be used selectively. Compared with the former scenario that requires complex decision-making for volume migration, the primary affinity control is rather straightforward in terms of action representation.

b: REWARD

`GetReward()` is implemented to yield the average QoS.

c: ACTION

`ExecuteAction()` is implemented based on the M-sized vector output generated by the affinity network. That output renders the desired primary affinity value of M OSDs. In the action adaptor, the output (action) of the affinity network is converted into a sequence of RADOS commands to update the primary affinity value of OSDs.

2) EXPERIMENT RESULTS

Figure 14 shows the QoS achieved by different methods across configurations, where Conf. k corresponds to one of the specific mixed configurations of tiered OSDs in Table 7. Our agent (CoMoRL) demonstrates its superiority, consistently outperforming the others in terms of QoS achieved under the MSR workload. CoMoRL achieves 1.6∼5.6%, 2.6∼7.7%, and 8.2∼16.5% higher QoS consistently for all Conf.1-6 than the IOPS-based, DR, and Model-based methods, respectively. Specifically, it shows 5.28% higher average QoS than the Default which does not control the primary affinity, and it shows 2.78%, 4.75%, and 10.33% higher average QoS than the IOPS-based, DR, and Model-based methods, respectively.

TABLE 7. The storage configuration of primary affinity scenario. Each Conf. k corresponds to a configuration tested in Figure 14.

-	Conf.1	Conf.2	Conf.3	Conf.4	Conf.5	Conf.6
Tier 1	1	1	1	1	1	1
Tier 2	1	2	1	1	2	1
Tier 3	1	1	2	1	1	2
Obj.	500	500	500	750	750	750

V. RELATED WORK

In the area of data center management and automatic operation, numerous research works using RL algorithms have been introduced, e.g., RL-based job scheduler [39], network traffic optimization [40], [41]. As storage system operation pertains to the problems of sequential decision-making for automatic operation and performance optimization, interest in RL-based automation for storage management has been raised recently. In Databot+ [10], the object placement on many SSDs was formulated in the RL context, similar to our volume placement scenarios, and in ARM [15], the problem of primary affinity control in Ceph was addressed using RL algorithms. Our test scenarios in Section IV, volume placement and primary affinity, follow the same structure as those works, but unlike them, we demonstrate the benefits of CoMoRL's configurable architecture which enables model adaptation without retraining to tackle continual storage system changes.

Regarding storage system optimization, only a few research works have considered the storage system heterogeneity or used heuristic management algorithms for different storage configurations. For example, in [3], the Autotiering

technique was used to maximize the I/O performance of virtual machines that run on a multi-tiered storage system in terms of throughput and latency. Several heuristic rules for making decisions about the optimal location of virtual disks were introduced to account for different I/O performance of storage tiers. For Ceph-based storage systems, a primary affinity control algorithm DLR was proposed in [42]. DLR enables the dynamic rebalancing of I/O loads by adjusting the primary affinity values, and it demonstrates a significant I/O throughput gain. In [4], the I/O pattern was investigated specifically for heterogeneous storage systems. Our work shares a similar purpose, i.e., performance optimization of multi-tier storage, as that prior research. However, our work enables zero-shot adaptation to continual system changes with heterogeneity and facilitates learning-based management strategies, by employing the model configurability.

In the RL research literature, numerous works have considered model-based approaches for training agents sample-efficiently [17], [18], [19], [43], particularly for the cases in which the target environment makes collecting sufficient training samples or online learning difficult. In [25], a model-based RL approach was investigated for low-level quadrotor flight controllers. In SOLAR [24], a linear quadratic regulator of model-based RL was developed for vision-based robot arm manipulation. Model learning for a complex and large-scale environment is considered to be particularly challenging in model-based RL. In O2P2 [43] and OP3 [17], the entity abstraction scheme was employed in a way that the dynamics model of each entity was learned individually and combined with others to build a complex environment, e.g., multiple block stacking. That work focused on the scale and complexity of a vision-based task using a per-object-level dynamics model, but it rarely considered temporal changes. While model-based RL can be a promising tool for the domains of large-scale system optimization, thanks to its sample efficient learning, the adaptation of learned models to continual system changes has rarely been investigated. Our CoMoRL is the first framework to adapt model-based RL with configurability for self-managing storage.

VI. CONCLUSION

In this paper, we proposed the configurable model-based RL framework CoMoRL for managing storage systems, which enables the establishment of zero-shot policy adaptation to continual storage system changes and various operation conditions. In the framework, storage management policies are achieved through meta-training on a set of model variants, and so they are able to adapt to unseen system specifications without retraining. Through experiments with the container volume placement and primary affinity control scenarios in our real storage cluster, we demonstrate that RL policies trained through CoMoRL are robust to different system specifications and outperform other baseline methods in terms of achieved QoS. That zero-shot adaptation of the RL policies is able to facilitate wide adoption of RL-based system automation in a data center, where the target system specification can be frequently changed during operation.

Our direction for future works is to adapt our framework to system areas other than storage such as task scheduling in a self-managing GPU cluster and network function management in a telecommunication infrastructure. The configurable model will be used to generate the model variants required for meta-training, hence allowing the learned management policies to adapt to operating conditions of different domains. We are also extending the model architecture in that the relational dynamics of heterogeneous components in a complex environment can be sample-efficiently learned. This will provide a technical foundation for facilitating RL-based zero-touch self-managing systems in various domains.

ACKNOWLEDGMENT

The authors would like to thank anonymous reviewers for their valuable comments and suggestions.

REFERENCES

[1] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar, "Flash storage disaggregation," in *Proc. 11th Eur. Conf. Comput. Syst.*, Apr. 2016, pp. 1–15.

[2] P. Stuedi, A. Trivedi, J. Pfefferle, R. Stoica, B. Metzler, N. Ioannou, and I. Koltsidas, "Crail: A high-performance I/O architecture for distributed data processing," *IEEE Data Eng. Bull.*, vol. 40, no. 1, pp. 38–49, Mar. 2017.

[3] Z. Yang, M. Hoseinzadeh, A. Andrews, C. Mayers, D. T. Evans, R. T. Bolt, J. Bhimani, N. Mi, and S. Swanson, "AutoTiering: Automatic data placement manager in multi-tier all-flash datacenter," in *Proc. IEEE 36th Int. Perform. Comput. Commun. Conf. (IPCCC)*, Dec. 2017, pp. 1–8.

[4] J. Zhou, Y. Chen, W. Xie, D. Dai, S. He, and W. Wang, "PRS: A pattern-directed replication scheme for heterogeneous object-based storage," *IEEE Trans. Comput.*, vol. 69, no. 4, pp. 591–605, Apr. 2020.

[5] C. Wu, C. Ji, Q. Li, C. Gao, R. Pan, C. Fu, L. Shi, and C. J. Xue, "Maximizing I/O throughput and minimizing performance variation via reinforcement learning based I/O merging for SSDs," *IEEE Trans. Comput.*, vol. 69, no. 1, pp. 72–86, Jan. 2020.

[6] S. Yoo and D. Shin, "Reinforcement learning-based SLC cache technique for enhancing SSD write performance," in *Proc. USENIX Workshop Hot Topics Storage File Syst.*, Jul. 2020, pp. 1–7. [Online]. Available: https://dl.acm.org/doi/10.5555/3488733.3488740

[7] W. Kang, D. Shin, and S. Yoo, "Reinforcement learning-assisted garbage collection to mitigate long-tail latency in SSD," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5s, pp. 1–20, Oct. 2017.

[8] W. Kang and S. Yoo, "Dynamic management of key states for reinforcement learning-assisted garbage collection to reduce long tail latency in SSD," in *Proc. 55th Annu. Design Autom. Conf.*, Jun. 2018, pp. 1–6.

[9] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proc. ACM Special Interest Group Data Commun. (SIGCOMM)*, Aug. 2019, pp. 270–288.

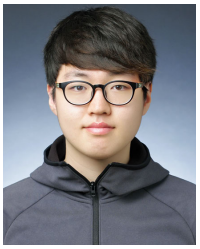
[10] K. Liu, J. Peng, J. Wang, B. Yu, Z. Liao, Z. Huang, and J. Pan, "A learning-based data placement framework for low latency in data center networks," *IEEE Trans. Cloud Comput.*, vol. 10, no. 1, pp. 146–157, Jan. 2022.

[11] J. Zhang, M. Ye, Z. Guo, C.-Y. Yen, and H. J. Chao, "CFR-RL: Traffic engineering with reinforcement learning in SDN," *IEEE J. Sel. Areas Commun.*, vol. 38, no. 10, pp. 2249–2259, Oct. 2020.

[12] S. Troia, F. Sapienza, L. Varé, and G. Maier, "On deep reinforcement learning for traffic engineering in SD-WAN," *IEEE J. Sel. Areas Commun.*, vol. 39, no. 7, pp. 2198–2212, Jul. 2021.

[13] P. Pinyoanuntapong, M. Lee, and P. Wang, "Delay-optimal traffic engineering through multi-agent reinforcement learning," in *Proc. IEEE Conf. Comput. Commun. Workshops (INFOCOM WKSHPS)*, Apr. 2019, pp. 435–442.

[14] J. Kossmann, A. Kastius, and R. Schlosser, "SWIRL: Selection of workload-aware indexes using reinforcement learning," in *Proc. Int. Conf. Extending Database Technol.*, Mar. 2022, p. 155.
[15] R. R. Noel, R. Mehra, and P. Lama, "Towards self-managing cloud storage with reinforcement learning," in *Proc. IEEE Int. Conf. Cloud Eng. (IC2E)*, Jun. 2019, pp. 34–44.
[16] *MiniNet*. Accessed: Nov. 1, 2022. [Online]. Available: http://mininet.org/
[17] R. Veerapaneni, J. D. Co-Reyes, M. Chang, M. Janner, C. Finn, J. Wu, J. Tenenbaum, and S. Levine, "Entity abstraction in visual model-based reinforcement learning," in *Proc. Conf. Robot Learn.*, Oct. 2020, pp. 1439–1456.
[18] L. Kaiser, M. Babaeizadeh, P. Milos, B. Osinski, R. H. Campbell, K. Czechowski, D. Erhan, C. Finn, P. Kozakowski, S. Levine, A. Mohiuddin, R. Sepassi, G. Tucker, and H. Michalewski, "Model-based reinforcement learning for Atari," 2019, *arXiv:1903.00374*.
[19] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, and T. Lillicrap, "Mastering Atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.
[20] Z. Guz, H. Li, A. Shayesteh, and V. Balakrishnan, "NVMe-over-fabrics performance characterization and the path to low-overhead flash disaggregation," in *Proc. 10th ACM Int. Syst. Storage Conf.*, May 2017, pp. 1–9.
[21] Y. Zhu, W. Yu, B. Jiao, K. Mohror, A. Moody, and F. Chowdhury, "Efficient user-level storage disaggregation for deep learning," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, Sep. 2019, pp. 1–12.
[22] A. Klimovic, H. Litz, and C. Kozyrakis, "Reflex: Remote flash ≈ local flash," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 1, pp. 345–359, 2017.
[23] *FIO*. Accessed: May 17, 2022. [Online]. Available: https://linux.die.net/man/1/fio
[24] M. Zhang, S. Vikram, L. Smith, P. Abbeel, M. Johnson, and S. Levine, "Solar: Deep structured representations for model-based reinforcement learning," in *Proc. Int. Conf. Mach. Learn.*, Jun. 2019, pp. 7444–7453.
[25] N. O. Lambert, D. S. Drew, J. Yaconelli, S. Levine, R. Calandra, and K. S. J. Pister, "Low-level control of a quadrotor with deep model-based reinforcement learning," *IEEE Robot. Autom. Lett.*, vol. 4, no. 4, pp. 4224–4230, Oct. 2019.
[26] R. Volpi, D. Larlus, and G. Rogez, "Continual adaptation of visual representations via domain randomization and meta-learning," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2021, pp. 4443–4453.
[27] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, Sep. 2017, pp. 23–30.
[28] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 1–11.
[29] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine, "Soft actor-critic algorithms and applications," 2018, *arXiv:1812.05905*.
[30] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *ACM Trans. Storage*, vol. 4, no. 3, pp. 1–23, Nov. 2008.
[31] P. Zhu et al., "VisDrone-DET2018: The vision meets drone object detection in image challenge results," in *Proc. Eur. Conf. Comput. Vis. Workshops*, Sep. 2018, pp. 1–30.
[32] G. Jocher et al., "Ultralytics/YOLOv5: V3.1—Bug fixes and performance improvements," Ultralytics, Los Angeles, CA, USA, Tech. Rep., Oct. 2020, doi: 10.5281/zenodo.4154370.
[33] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common objects in context," in *Proc. Eur. Conf. Comput. Vis.*, Sep. 2014, pp. 740–755.
[34] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2009, pp. 248–255.
[35] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *Proc. Int. Conf. Mach. Learn.*, Jun. 2019, pp. 6105–6114.
[36] *FIO Docker*. Accessed: May 17, 2022. [Online]. Available: https://hub.docker.com/r/xridge/fio
[37] S. Weil, S. Brandt, E. Miller, and C. Maltzahn, "CRUSH: Controlled, scalable, decentralized placement of replicated data," in *Proc. ACM/IEEE SC Conf. (SC)*, Nov. 2006, p. 31.
[38] *Librados*. Accessed: May 17, 2022. [Online]. Available: https://docs.ceph.com/en/latest/rados/api/python/
[39] F. Li and B. Hu, "DeepJS: Job scheduling based on deep reinforcement learning in cloud data center," in *Proc. 4th Int. Conf. Big Data Comput.*, 2019, pp. 48–53.
[40] C. Tessler, Y. Shpigelman, G. Dalal, A. Mandelbaum, D. Haritan Kazakov, B. Fuhrer, G. Chechik, and S. Mannor, "Reinforcement learning for datacenter congestion control," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 49, no. 2, pp. 43–46, Jan. 2022.
[41] L. Chen, J. Lingys, K. Chen, and F. Liu, "AuTO: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2018, pp. 191–205.
[42] R. R. Noel and P. Lama, "Taming performance hotspots in cloud storage with dynamic load redistribution," in *Proc. IEEE 10th Int. Conf. Cloud Comput. (CLOUD)*, Jun. 2017, pp. 42–49.
[43] M. Janner, S. Levine, W. T. Freeman, J. B. Tenenbaum, C. Finn, and J. Wu, "Reasoning about physical interactions with object-centric models," in *Proc. Int. Conf. Learn. Represent.*, May 2019, pp. 1–12.



SEUNGHWAN JEONG received the B.S. degree from the Department of Software, Sungkyunkwan University, in 2018. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, Sungkyunkwan University. His research interests include intelligent application, reinforcement learning, and storage system management.

HONGUK WOO (Member, IEEE) received the B.S. degree in computer science from Korea University, Seoul, in 1995, and the M.S. and Ph.D. degrees in computer science from The University of Texas at Austin, Austin, TX, USA, in 2002 and 2008, respectively. From 2008 to 2018, he worked at Samsung Research, Samsung Electronics, as a Principal Engineer and the Vice President. Since 2018, he has been working with the Department of Computer Science and Engineering, Sungkyunkwan University, Suwon, South Korea, and he has been working as an Associate Professor, since 2022. His research interests include intelligent cyber-physical systems, self-managing systems, and machine learning.