Metis: Learning to Schedule Long-Running Applications in Shared Container Clusters at Scale

Luping Wang*†, Qizhen Weng*†, Wei Wang†, Chen Chen†‡, Bo Li†
†Hong Kong University of Science and Technology
‡Huawei Technologies Ltd
{lwangbm, qwengaa, weiwa}@cse.ust.hk, cchenam@connect.ust.hk, bli@cse.ust.hk
*Equal Contribution

***Abstract*—Online cloud services are increasingly deployed as *long-running applications* (LRAs) in containers. Placing LRA containers is known to be difficult as they often have sophisticated resource interferences and I/O dependencies. Existing schedulers rely on operators to manually express the container scheduling requirements as *placement constraints* and strive to satisfy *as many constraints as possible*. Such schedulers, however, fall short in performance as placement constraints only provide *qualitative* scheduling guidelines and minimizing constraint violations does not necessarily result in the optimal performance.**

In this work, we present Metis, a general-purpose scheduler that *learns* to optimally place LRA containers using *deep reinforcement learning* (RL) techniques. This eliminates the complex manual specification of placement constraints and offers, for the first time, concrete *quantitative* scheduling criteria. As directly training an RL agent does not scale, we develop a novel *hierarchical learning* technique that decomposes a complex container placement problem into a *hierarchy* of subproblems with significantly reduced state and action space. We show that many subproblems have similar structures and can hence be solved by training a *unified* RL agent *offline*. Large-scale EC2 deployment shows that compared with the traditional constraint-based schedulers, Metis improves the throughput by up to 61%, optimizes various performance metrics, and easily scales to a large cluster where 3K containers run on over 700 machines.

I. Introduction

Production clusters run two types of workloads, *long-running applications* (LRAs) for online cloud services [1]–[14] and *batch jobs* for offline data analytics [15]–[17]. Unlike batch jobs that run in short-lived task executors, LRAs run in *long-lived containers* with durations spanning hours to months [18]–[20]. This allows dynamic queries to be served in real time without the overhead of repeatedly launching containers upon their arrivals. Compared with batch jobs, LRAs run business-critical, user-facing services with stringent SLO (Service-Level Objective) requirements, and need to scale out to a large number of containers in response to load spikes [21]. LRAs are hence scheduled as *first-class citizens* in production clusters [18]–[20], [22]. Fig. 1 illustrates a typical scheduling process for LRA containers.

LRAs have sophisticated *performance interactions* that dramatically complicate container placement. LRAs commonly have I/O dependencies in that one's input depends on the output of another, e.g., a Spark streaming instance [23] running business analytics service reads streaming data prepared by a Kafka instance [3]. In the meantime, LRA containers routinely

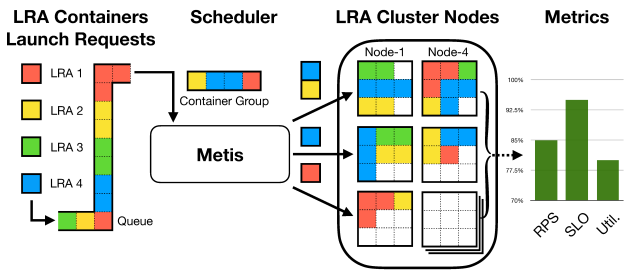


Fig. 1: An illustration of LRA scheduling: the container launching requests, received in a queue buffer, are scheduled in groups to optimize various performance metrics.

face interference from the co-located containers contending shared resources such as CPU cache, network, I/O and memory bandwidth. Existing LRA schedulers define various *placement constraints* [18], [24]–[28], such as *affinity*, in which I/O-dependent containers should be co-located on one machine to avoid the communication overhead, and *anti-affinity*, in which contending containers should be placed on separate machines to avoid resource interference. The scheduler then places containers to satisfy *as many constraints as possible*, using simple heuristics [18], [29]–[35].

However, such *constraint-based scheduling* has three fundamental problems. First, it requires cluster operators to identify complex container interactions based on operational experience and manually specify them as placement constraints. This requires onerous human efforts yet can still be inaccurate. Second, placement constraints only provide a *qualitative* scheduling guideline, but *do not quantify* the actual performance impact (e.g., to what degree can the throughput be harmed if violating a certain constraint). Consequently, when the scheduler cannot satisfy all constraints, it may mistakenly choose to violate those with more significant impacts. Third, complex placement constraints often formulate an *intractable* optimization problem in large clusters.

In this paper, we show that optimal LRA placement can be *automatically learned* using *deep reinforcement learning* (RL) techniques [36], without the need of specifying placement constraints. We present Metis[1], an intelligent, general-purpose LRA scheduler capable of optimizing various scheduling ob-

[1]In Greek mythology, Metis is the goddess of wisdom, prudence, and deep thought.

jectives, such as throughput, SLO satisfaction, and cluster utilization. Metis learns the sophisticated performance interactions between LRA containers from past workload logs or from lightweight offline profiling. Based on that information, Metis learns to schedule containers by encoding the scheduling policy into a neural network and training it with extensive *simulated experiments*, in which it places LRA containers, predicts their performance, and iteratively refines the policy.

The key to Metis is to build an RL model that scales to large clusters where tens of thousands of LRA containers run on thousands of machines. However, directly training an RL agent at such scale is *computationally infeasible*, as it mandates high-dimensional state representations (i.e., container placement on all machines). We find that the RL techniques recently developed for the other scheduling problems offer little help in addressing the scalability challenge posed to container placement (see §III). Most of these works learn to find the optimal *scheduling order* of batch jobs [37]–[39] or network flows [40], whereas LRA scheduling concerns the interactions between containers, and is essentially a *combinatorial placement optimization* problem. This fundamental difference invalidates their techniques in our problem.

We address the scalability challenge of container placement with three new techniques. First, unlike existing works [37]–[41] that train a generic scheduling policy *offline* for all possible input workloads, we train a *dedicated* RL model every time when a group of containers arrive. Having a dedicated model tailored to each container group results in high-quality placements and reduced training complexity. Although it takes more time to make a decision, LRAs are insensitive to the scheduling latency due to their long-running nature [18].

Second, to enable scalable learning, we propose a *hierarchical reinforcement learning* technique using a **divide-and-conquer** approach. Our key idea is to decompose a complex learning task into *a hierarchy of subtasks with significantly reduced state and action space*. Specifically, given a container, we learn to place it following a decision tree. At the root level, we divide the cluster into K partitions and learn to determine which sub-cluster provides the optimal placement. Following our choice, we descend to a lower level of the tree, where we subdivide the selected sub-cluster into K partitions, and learn to choose the one that contains the optimal placement. As we traverse down the decision tree, we recursively narrow our search for the container placement, until it pinpoints a single machine. In the end, the scheduler makes a sequence of K-choose-1 decisions that can be individually learned at each level by training simple RL models. Note that at one level all the decision-making tasks have the same learning structure, enabling a *shared model* to be reused by all tasks.

Third, to further accelerate the learning process, we *pretrain a sub-scheduler* for general workloads in a small sub-cluster. This allows the traverse down the decision tree to stop at a high level when reaching that sub-cluster, where the **pretrained sub-scheduler** is used to decide the container placement. This approach results in a "shallowed" decision tree, with the decision space reduced exponentially. In our evaluation, using pretrained sub-schedulers leads to $95\times$ training speedup (§VII-C, Fig. 11).

We have implemented Metis as a pluggable scheduling service in Docker Swarm [42]. We evaluate its performance in two Amazon EC2 [43] clusters (81 and 729 nodes) against seven real-world applications covering machine learning, stream processing, I/O and storage services. Compared with the state-of-the-art constraint-based schedulers [18], [29], Metis achieves up to 61% higher request throughput. Metis is also capable of optimizing various scheduling objectives that are otherwise not directly supported by the existing constraint-based schedulers (e.g., minimizing SLO violations). Metis easily scales to large clusters where thousands of containers run on over 700 machines.

II. Background

Long-Running Applications Online cloud applications are widely hosted in production clusters to provide interactive, latency-critical services, such as stream processing [1]–[4], interactive data analytics [44], [45], storage services [5]–[7], and machine learning [8]–[14]. These applications usually run in *long-lived containers*, enabling them to respond to dynamic queries in real time without repeatedly launching or warming up new containers at request arrivals. Previous study shows that in Microsoft clusters, the containers of online services typically run for hours to months [18]. Our communication with Alibaba Cloud confirmed the same phenomenon. Online cloud applications are thus referred to as *long-running applications* (LRAs) in the literature [18], [20].

LRAs run business-critical, user-facing services with stringent SLOs and are hence scheduled as first-class citizens in the clusters [18]–[20], [22]. In Microsoft, many large clusters are entirely dedicated to LRA workloads [18]. Analysis of the recently released Alibaba cluster trace [46] shows that 94.2% of the cluster CPUs were allocated to run LRA containers [19]. As LRAs dominate resource allocations in production clouds, they play a pivotal role in cluster scheduling.

Scheduling LRA Containers An LRA typically consists of multiple containers, each running a replica of the service instance. Over time, an LRA may launch more containers or destroy existing ones as the request load changes. Consider a cluster consisting of N machines, each of which can host a certain number of containers.[2] Given the dynamic arrivals of the container launching requests from M LRAs, the scheduler examines the current cluster state (i.e., the currently running containers on each machine) and determines the placement of each container, with the objectives of maximizing the overall container throughput, SLO satisfaction rate, cluster utilization, or any other performance metrics in combination (see Fig. 1). Once a container is deployed, it runs for a long time, during which no preemption or migration is allowed.

To achieve better performance, the scheduler batches the received container launching requests into *groups*, each con-

[2]For simplicity, we assume containers with homogeneous resource demands. Yet our approach can be extended to a more general case.

sisting of no more than T containers. The scheduler then makes placement decisions on a *per-group* basis. Compared with per-container placement [24], [29], [30], scheduling a group of containers in one go enables more combinatorial optimization opportunities, as it gives the scheduler a "global view" to account for the performance interactions between containers within a group [18], [35]. Configuring a larger group usually leads to a better placement. Yet, it significantly complicates the scheduling decision, plus accumulating a large number of arriving containers also takes time. In this work, we tune the group size for improved placement quality without causing overly long scheduling latency.

Interactions between Containers LRA containers have sophisticated performance interactions that substantially complicate the placement problem [18], [29], [30]. While containers effectively divide CPU cores and memory capacity among packaged applications, they offer poor isolation when contending for *shared resources* that are not managed by the OS kernel, such as network, CPU cache, disk I/O, and memory bandwidth. Therefore, co-locating many contending containers increases resource contention, resulting in significant *interference* that harms the performance of all the hosted containers.

On the other hand, co-locating LRA containers can sometimes be beneficial. In production clusters, many online services are structured as graphs of *dependent* LRA containers where the output of an upstream container is forwarded to a downstream instance for further processing [47], [48]. Co-locating two dependent LRA containers on one machine avoids transferring a large amount of data over the network, leading to faster responses to query processing.

Given the complex interactions between LRAs, judiciously scheduling their containers for optimal performance becomes critically important [18], [29], [30], [33], [34], [49]–[51].

III. Prior Arts and Their Limitations

In this section, we briefly review existing LRA schedulers that make scheduling decisions based on placement constraints. We discuss their inefficiency and motivate the need for an intelligent LRA scheduler driven by reinforcement learning (RL) techniques. We show that previous RL solutions developed for the other scheduling problems do not scale to container placement in large clusters.

A. *Inefficiency of Constraint-based Solution*

Constraint-based LRA Scheduling Existing cluster schedulers use various *placement constraints* to capture the complex interactions between LRA containers [18], [24]–[28]. Common constraints supported in popular cluster management systems [24], [25], [42] include *affinity*, which co-locates I/O-dependent containers to provide data locality, and *anti-affinity*, which schedules contending containers on separate machines to avoid resource interference. More expressive constraints have also been proposed to support more sophisticated requirements [18], [35]. These constraints are usually specified through scheduler-provided APIs [18], [24], [25]. The schedulers then make container placement decisions to satisfy *as many constraints as possible*, either with simple greedy strategies [29], [30] or by solving a constrained combinatorial optimization problem [18], [31]–[35].

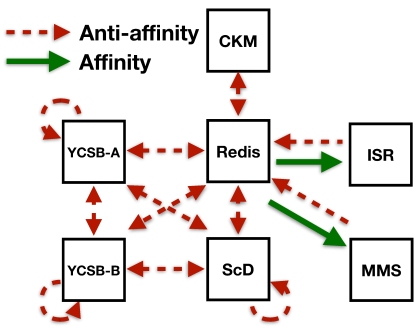


Fig. 2: An illustration of the performance interactions between seven applications: Redis [5], MXNet Model Server (MMS) [11], image super resolution (ISR) [52], file checksum (CKM) [53], video scene detection (ScD) [54], and two YCSB benchmarks [55]. Detailed descriptions are given in §VII-A.

TABLE I: Minimizing constraint violations does not necessarily optimize the performance in the example of Fig. 3.

Scheduler	# of Violated Constraints	Average RPS
Medea [18]	4 / 468 (minimum)	0.93
Metis	6 / 468	1.16

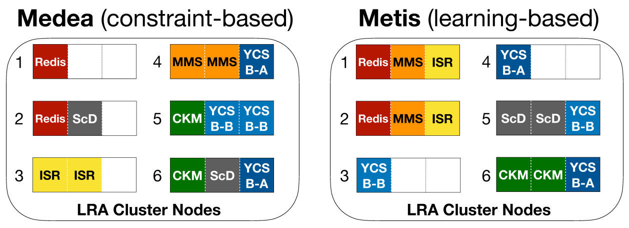


Fig. 3: Container placements given by Medea and Metis when running seven LRAs (Fig. 2) in a 6-node EC2 cluster.

However, simply relying on placement constraints for LRA scheduling can be highly inefficient, as we explain below.

Expensive to Specify Placement Constraints In production clusters, placement constraints are often specified *manually* by cluster operators based on operational experience, which requires expert knowledge and extensive human efforts. Many organizations do not have such expertise or find the required labor expense too costly to justify the benefits. Although prior works [29], [30] show that some placement requirements can be efficiently profiled by classification techniques, they can only identify resource contentions (i.e., anti-affinity), but not the affinity requirements (e.g., I/O dependency between containers) or more sophisticated interactions (e.g., the *cardinality* requirement as supported in Medea [18]).

Minimizing Constraint Violations Does Not Guarantee Optimal Performance Placement constraints do not *quantify* the significance of the underlying performance gain (or loss). Take the affinity constraint as an example. Beyond stating that co-locating two containers is desirable, it offers no clue about *how much* performance gains that co-location can provide.

Consequently, given *conflicting* constraints, the scheduler may unwisely choose to violate those with more significant impact.

To illustrate this problem, we run 14 containers of seven real applications (two containers each) in a 6-node EC2 cluster (details given in §VII-A). We manually profile their performance interactions and depict the results in Fig. 2. Note that Redis and MMS have *conflicting* placement requirements: Co-locating an MMS container with Redis greatly improves the RPS of the former by 147% (affinity) but reduces that of the latter by 13% (anti-affinity). We make a similar observation in co-locating ISR and Redis containers.

We schedule the 14 containers using Medea [18], a state-of-the-art constraint-based LRA scheduler, which formulates an ILP problem based on the profiled placement constraints. Medea produces a placement with 4 constraint violations, the *minimum* one can expect, and achieves 0.93 RPS on average. However, by violating more constraints, our solution Metis gives an even better placement with higher RPS (see Table I).

Fig. 3 compares the placement decisions made by the two schedulers, where the key difference is the placements of Redis, MMS, and ISR containers. Notably, Medea assigns Redis and MMS (or ISR) containers to different nodes. In contrast, Metis learns that co-locating Redis with MMS and ISR is more beneficial than separating them, i.e., the impact of affinity *outweighs* that of anti-affinity. The result is 25% higher RPS than that obtained with Medea.

Limited Support of Scheduling Objectives Constraint-based scheduling formulates an ILP problem [18], [35], which does not support optimizing the objectives that cannot be quantitatively specified by the placement matrix [56]. For example, high-level scheduling objectives that are not linearly correlated to the container allocation such as guaranteeing a specific level of QoS or balancing network traffics are not supported in Medea.

Intractable Optimization Formulation To make matters worse, solving ILP problems is time-consuming and does not scale to large clusters [18], [35]. Taking Medea [18] as an example, the ILP formulation uses a variable $X_{ijn} \in \{0,1\}$ to indicate if container j of LRA i is placed on node n. For each placement constraint (affinity and anti-affinity), a specific *ILP-constraint* is required for all machines and subjected containers. Consequently, the formulation involves $O(NM)$ ILP-constraints when placing M containers to an N-node cluster. As we will show in §VII-B, when placing thousands of containers to a large cluster of more than 700 nodes, the ILP formulation involves 1M variables and 10M ILP-constraints. Even finding a feasible solution takes more than 10 hours.

B. Learning-based Scheduling Solutions

The inefficiency of constraint-based scheduling calls for a more intelligent alternative that automatically learns to schedule LRA containers without pursuing an intermediate objective, such as meeting the placement constraints. We find that modern reinforcement learning (RL) techniques [36] offer a particularly appealing solution for LRA scheduling. *First*, over the years, production clusters have accumulated a significant amount of workload logs, providing rich operation data for learning container interactions. *Second*, LRA containers can tolerate *long scheduling latency* in exchange for high-quality placement [18], allowing sufficient time for the RL agent to learn the placement policy.

RL-based Solutions for Other Scheduling Problems A rich body of recent works have applied RL techniques to many other scheduling problems, such as batch job scheduling [37]–[39], network optimization [40], [57], and device placement [58], [59]. However, we find that their techniques are either *inapplicable* to container placement or limited to an optimization problem at a *small scale*.

One successful RL-based scheduling is to learn the *optimal scheduling order* of batch jobs [37]–[39]. For example, Decima [39] is a meticulously optimized scheduler for DAG-structured batch jobs. Decima automatically encodes the DAG information into feature vectors and trains a policy network to determine which tasks should be launched next and how many executors should be allocated. Spear [38] learns to schedule DAG-structured jobs with Monte Carlo Tree Search. DeepRM [37] trains an RL policy for packing batch jobs with multi-resource demands. Unlike LRA workloads, batch jobs run in executors *without complex interactions*, and their performance critically depends on the *scheduling order* rather than job placement. Therefore, the RL designs developed for job scheduling do not apply to container placement with a fundamentally different formulation. In fact, as the former mainly concerns the scheduling order of M jobs, the action space is $O(M!)$. In comparison, scheduling M containers on an N-node cluster formulates a far more complex *combinatorial placement optimization problem* with substantially larger action space of $O(N^M)$.

To our knowledge, there are only a few attempts of using RL techniques to solve combinatorial placement problems at a small scale. Notably, Google proposes to use RL to optimally partition the operations of a deep neural network onto different devices (CPUs and GPUs) for fast model execution [58], [59]. It encodes the information and dependencies of these operations into a sequence-to-sequence model, and learns to assign these operations onto *a few* devices. However, even at such a small scale, learning the optimal device placement may still take more than 10 hours [58].

IV. Learning to Schedule LRA Containers

In this section, we propose to learn the optimal LRA container placement using deep reinforcement learning (RL). We present our design choices for a practical RL-based scheduler and discuss the key scalability challenges posed to it.

A. Learning Optimal Placement with RL

LRA scheduling naturally formulates an RL problem. Given the existing container placement (*state*) in the cluster, the scheduler (*RL agent*) learns to place new LRA containers (*action*) based on its interactions with the cluster (*environment*). We use a neural network to encode the scheduling policy and train it with extensive *simulated experiments*: the scheduler places containers, observes the performance outcome (*reward*), and iteratively improves the policy. We visualize this training process in Fig. 7 (the dotted arrows on the left).

State Recall in §II that the scheduler group-schedules T containers at their arrivals. We treat each group scheduling as an *episode* consisting of T steps, where in each step, only one container is placed onto a machine. More specifically, consider an N-node cluster running M applications. Assume that in step t, the RL agent has already placed $t-1$ containers in the group and will next schedule container c_t. We embed container c_t into a *one-hot vector* $\mathbf{e} = \langle e_1, \ldots, e_M \rangle$, where each element e_i is 1 if c_t belongs to application i, and 0 otherwise. We further define the *state of a node* as the vector $\mathbf{v}_n = \langle v_{n1}, \ldots, v_{nM} \rangle$, where v_{ni} measures the number of containers it runs for application i. Concatenating container c_t and the states of all nodes in step t, we define the *cluster state* $s_t = \langle \mathbf{e}, \mathbf{v}_1, \cdots, \mathbf{v}_N \rangle$, which is observed by the RL agent.

Action and Reward Given a state s_t, the RL agent takes an *action* $a_t = n$ which schedules container c_t to node n, and transits the system to a new state s_{t+1} in the next step. The agent will evaluate the performance of the group placement in the final step T after *all* T containers are scheduled. More specifically, the agent receives no reward r_t at an intermediate step $t < T$, and the final reward r_T, which can be any performance measures such as the normalized average throughput of the scheduled container group, SLO satisfaction rate, cluster utilization, or their combinations, is *independent* of the actual container scheduling order within the group.

Training Policy Network with Policy Gradient We encode the scheduling policy into a neural network with parameters θ, known as *policy network* π_θ. It takes as input the cluster state and outputs a distribution over all possible actions [36]. We train the policy network using the REINFORCE algorithm [60] which performs gradient ascent on parameters θ using the rewards observed during training, i.e.,

$$\theta \leftarrow \theta + \alpha \sum_t \nabla_\theta \log \pi_\theta \Big(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} - b\Big). \tag{1}$$

Here, α is the learning rate, $\gamma \in (0, 1]$ a factor used to discount future rewards, and b the *baseline* used to *reduce the variance* of the policy gradient. A common choice of the baseline is the average reward observed during training [36], [61]. These hyper-parameters can be tuned for faster convergence. Intuitively, this gives the agent a high chance of choosing an action with a better-than-average reward, accelerating the training process.

Inspired by [39], we use *experience replay* [62], [63] to further speed up the training. The main idea is to store the performed actions leading to high rewards in a replay buffer. The agent will periodically sample those high-performance actions for policy updating. By replaying the good experience encountered in previous trainings, the agent can learn faster. More details can be found in the supplemental material [64].

Algorithm 1: Policy-gradient training routine of Metis

```
Input : An $N$-node cluster with node $n \in \{1, \ldots, N\}$, a group of
        $T$ containers $\{c_1, c_2, \ldots, c_T\}$ to allocate on these nodes.
Output: An allocation $\{a_1, a_2, \ldots, a_T\}$ of the batch of $T$
        containers ($a_t \in \{1, \ldots, N\}$ is allocating container $c_t$ on
        node $n = a_t$).
1  Initialize the environment and performance indicator $R$
2  Set replay buffer $B$ and best performance $R^*$ as empty
3  for episode=1,2,...,E do
4      Initialize the state $s = \{\mathbf{e},\ \mathbf{v}_1, \ldots, \mathbf{v}_N\}$
5      for t=1,2,...,T do
6          Choose an action $a_t$ with policy $\pi(a_t|s_t)$
7          Execute the action and observe a new state $s_{t+1}$
8      Collect all perf. indicators as reward $r = \sum_t R(c_t)$
9      if $r \geq \eta R^*$          // $\eta \in (0, 1]$: replay threshold
10      then
11         Store experience $\{s_1, a_1, \ldots, s_T, a_T, r\}$ in $B$
12         $R^* \leftarrow \max(r,\ R^*)$
13     Every $C$ episodes, use REINFORCE algorithm to update
         $\pi(a_t|s_t)$, with both recent $C$ experiences and a mini-batch of
         experiences sampled from $B$
14 Return the action $\{a_1, a_2, \ldots, a_T\}$ of the experience with the
     highest reward $r = R^*$ in replay buffer $B$.
```

To summarize, Algorithm 1 presents the pseudo-code of RL agent training run in both the vanilla and augmented versions (i.e., hierarchical reinforcement learning, to be elucidated under the "Scheduling Routine" in §V-A). Specifically, the input is a group of containers and the output is their corresponding allocations. Line 4 shows the state input, which is the concatenation of the LRA vector $\mathbf{e}$ of the current container to schedule and the states of all nodes $\langle \mathbf{v}_1, \ldots, \mathbf{v}_N \rangle$ (see the "State" above). Lines 5–7 are the allocation process of all T containers in the batch; the reward is issued after all of them are scheduled (Line 8) (see the "Action and Reward" above). Lines 9–12 describe the experience replay technique that stores experiences with high rewards in the replay buffer. In Line 13, the agent updates its policy with both recent experiences and the sampled ones from the replay buffer, using the REINFORCE algorithm (Eq. (1)). We refer interested readers to Sutton and Barto's textbook [36] for detailed explanations. In the end, the RL agent selects the action sequence with the highest reward in the replay buffer and returns it as the final placement decision (Line 14).

B. Cluster Environment Simulator

Training a neural network using policy gradient requires the RL agent to frequently interact with the environment, which is extremely time-consuming for LRA scheduling. In real systems, it takes at least minutes to deploy containers and measure their performance. As RL training typically requires tens of thousands of iterations to complete [37], [39], [58], [59], [61], [65], having the scheduler directly interacting with a real cluster is too slow to be practical.

Similar to prior work [39], [41], we develop a high-fidelity *cluster environment simulator* that can faithfully predict container performance given a placement. This allows the learning to be performed with *simulated experiments* without deploying containers in a real cluster.

Instead of modeling low-level resource interference based on contentions on CPU caches or memory bandwidth, which may not be available in production traces, we turn to high-level performance metrics such as container throughput and request latency. We then predict how those metrics may change under varying container placements. In particular, we log the machine-level container co-location vectors along with the observed RPS/latency of each resident container, and use them as the training samples for the simulator. Our simulator uses multivariate Random Forests (RF) [66] as the main regressor to characterize container interactions. RF method uses a combination of decision trees to perform the regression task, and can make accurate predictions with a small number of training data. It is also resilient to overfitting when supplied with a large number of repetitive samples. Both properties make the RF regressor an ideal choice for our simulator. We defer the implementation details to §VI.

C. Training Dedicated Model on the Spot

Previous RL-based schedulers [37]–[39], [41] train a *unified* RL model *offline* and use it *online* to make scheduling decisions. However, this approach falls short in LRA scheduling. As a container group comes with a large number of combinations, an LRA scheduler needs to handle *highly variant input*—in our previous experiments with seven applications (Fig. 2), scheduling a small group of 30 containers requires the scheduler to handle over one million possible container combinations in input. When it comes to a large cluster, offline training a scheduling policy for extremely variant workloads inevitably results in poor performance.

Instead, we train a *dedicated* RL model *on the spot* upon the arrival of a new group of containers. While training a dedicated model takes time, the long-running nature of LRA containers allows them to tolerate relatively long scheduling latency (e.g., tens of minutes) in exchange for better placements [18].

D. Scalability Challenge

However, directly training a dedicated model using standard RL techniques (§IV-A) suffers from two scalability problems.

Exponential State Space Given an input container group, for each container, the RL agent needs to choose the optimal placement from the whole cluster. This requires the agent to keep track of the container placement on *all* machines (see the definition of state s_t in §IV-A). Maintaining such a *high-dimensional* state results in *the curse of dimensionality* [67]–[71], in that the state space grows exponentially as the cluster size increases. In order to learn a stable policy, each state must be visited multiple times during training [72]. Having an exponential state space mandates prohibitive training efforts.

Exponential Scheduling Decision Space In a training episode, the RL agent takes a sequence of actions to place a group of containers, which together compose a *scheduling decision*. As the action space for each container placement is proportional to the number of machines in the cluster (i.e., choosing one machine out of all), the *scheduling decision*

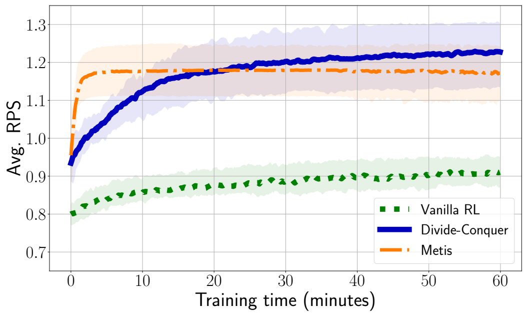


Fig. 4: An illustration of the scalability challenge posed to RL approaches: learning curves of vanilla RL and Hierarchical RL designs in an 81-node cluster.

space also grows exponentially as the cluster size increases. This, in turn, requires an extremely large number of training episodes in efforts to search the optimal placement combination, dramatically prolonging the training time.

To demonstrate these scalability problems, we launch an 81-node EC2 cluster and run the seven applications in Fig. 2. We randomly generate 30 container groups, each having 200 containers, and measure their average RPS. We depict the learning curve of vanilla RL in Fig. 4 (the green, dotted curve). We observe that the agent makes little progress throughout the training process. In fact, as there are so many machine candidates to choose from, the agent easily gets stuck at a local optimum, without a chance to explore a better placement.

V. Enabling Scalable Learning with Metis

In this section, we present Metis, an intelligent LRA scheduler that enables scalable learning in large clusters. Metis learns to place LRA containers using *hierarchical reinforcement learning* (HRL) techniques [67], [71], [73]–[76]. The main idea of HRL is to decompose an initially complex learning task into multiple layers of simpler subtasks. Each subtask is independently modeled as a single MDP (Markov Decision Process) with its own state, action, and reward.

We next present two key HRL designs for LRA scheduling that aim to answer two questions: (1) How to decompose the scheduling task into a hierarchy of subtasks with reduced state and action space (§V-A)? (2) How to reuse the building blocks in the learning hierarchy for generalization (§V-B)?

A. Divide-and-Conquer Placement

Divide-and-Conquer Throughout the RL training process, determining the placement for a given container is the most complex task. We decompose it into a sequence of simpler decision-making problems with *substantially reduced state and action space*. To facilitate this *divide-and-conquer* (D&C) strategy, we *recursively divide* a cluster into smaller ones following a *decision tree*. Specifically, at the root level (L0) of the tree, we conceptually divide the entire cluster into K equal-sized *sub-clusters*, each being a child of the root located at the next level (L1). For each L1 sub-cluster, we further divide it into K smaller partitions, each being a child at the

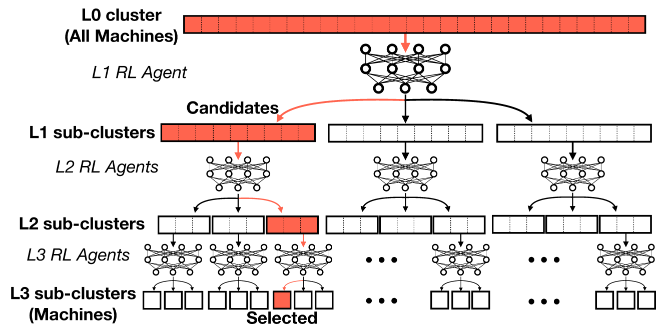


Fig. 5: An illustration of the divide-and-conquer placement. Instead of using a monolithic neural network to determine the container placement, we make a sequence of K-choose-1 decisions following a decision tree, each of which can be individually learned using a small neural network.

next level L2. Continuing this process, we recursively divide an upper-level sub-cluster into K smaller ones at a lower level. The decomposition stops when a sub-cluster contains only a single machine. Fig. 5 illustrates this process, where a 27-node cluster is recursively divided into three sub-clusters.

Following the decision tree, we gradually narrow down the search for a container placement by making a sequence of K-choose-1 decisions, each corresponding to an independent RL sub-policy. In particular, to determine the placement of a container in the cluster, we start from the root level L0, where we learn to find which one of the K sub-clusters at the next level L1 provides the optimal placement. Following our choice, we descend to L1 and narrow the search scope from the entire cluster down to the selected sub-cluster. We next learn to find which sub-cluster of its children at L2 contains the optimal placement. As we traverse down the decision tree, we recursively narrow the search scope for the container placement, until it pinpoints a single machine at the bottom level. Continuing our previous example in Fig. 5, we make three 3-choose-1 decisions (highlighted in red) to locate the host machine for a container in a 27-node cluster.

Compared with choosing the container placement from all N machines using a monolithic neural network (§IV-A), our divide-and-conquer approach makes a sequence of K-choose-1 decisions ($\log_K N$ decisions made in total). Each decision can be individually learned using a small neural network with significantly reduced state space (Fig. 5), as we explain below.

State Summarization for Sub-clusters For each sub-cluster p, we train a policy network (an RL agent) to determine which one of its K child sub-clusters provides the optimal placement for a container. We first embed the current container placement of each child sub-cluster into a *feature vector* which is a high-level *state summarization* of the included machines. Through careful feature engineering, we find that an informative state summarization is given by *the number of containers each application runs in a sub-cluster*. More formally, let $C(p)$ be the set of K child sub-clusters of p. For each child $k \in C(p)$, we define its feature vector as $\mathbf{f}_k = \langle f_{k1}, \ldots, f_{kM} \rangle$, where M is the number of LRAs and f_{ki} the number of containers LRA i runs in k. To choose the best child out of the K candidates for a given container, the RL agent takes as state input $s = \langle \mathbf{e}, \{\mathbf{f}_k\}_{k \in C(p)} \rangle$, where $\mathbf{e}$ is the one-hot LRA embedding vector of the container (see State in §IV-A).

Scheduling Routine We next elaborate the scheduling routine used to determine the placement for a given container. Starting from the root level L0, we summarize the state vectors of all K child sub-clusters in L1. Concatenating these summarizations with the LRA embedding vector, we have the L1 state $s_1 = \langle \mathbf{e}, \{\mathbf{f}_k\}_{k \in C} \rangle$, where C is the collection of L1 sub-clusters. With state input s, the L1 RL agent outputs an action k_1^*, which chooses an L1 sub-cluster k_1^* to place the container. Now in k_1^*, we summarize its K child sub-clusters in L2 and obtain the L2 state $s_2 = \langle \mathbf{e}, \{\mathbf{f}_c\}_{c \in C(k_1^*)} \rangle$. Taking s_2 as input, the L2 RL agent outputs a child sub-cluster $k_2^* \in C(k_1^*)$ to narrow the search scope. We recursively continue this process until the bottom RL agent outputs a single machine to host the container. We use this scheduling routine to place all the containers in the group. At the end of the episode, we evaluate the collective performance of the placed containers, and use it as the reward *shared* by all the involved $\langle \texttt{state}, \texttt{action} \rangle$ pairs to update the RL policies in the hierarchy.

Recall that Algorithm 1 gives the scheduling routine for vanilla RL in §IV-A. For hierarchical RL agents in sub-clusters, it only requires minor modifications to accommodate the idea of *state summarization*. In particular, (1) the "Input" becomes "a cluster with K sub-clusters; (2) each action in "Output" is "$a_t \in \{1, \cdots, K\}$", meaning allocating container c_t to sub-cluster $k = a_t$; (3) the state in Line 4 is redefined as $s = \langle \mathbf{e}, \{\mathbf{f}_k\}_{k \in C} \rangle$, where C is a collection of the corresponding level of sub-clusters.

Benefits Compared with the standard RL described in §IV-A, our divide-and-conquer approach offers three benefits.

1) *Manageable state size.* In each subtask that makes a K-choose-1 decision, the RL agent maintains a *concise state* consisting of K *fixed-sized* feature vectors. As the state structure is *independent* of the cluster size, scaling out the learning to a larger cluster with more machines suffers no curse of dimensionality: it only adds more subtasks but does not complicate any of them. Moreover, the concise state of each subtask results in a *manageable state space* with reduced environment variance during training, allowing high-quality container placement to be learned quickly.

2) *Tractable action space.* Compared with the N-choose-1 decision in standard RL, the action space in each subtask consists of only K actions ($K \ll N$). Because K is a constant that does not grow with the cluster size, the action space is reduced from $O(N)$ to $O(1)$.

3) *Reusable RL models among subtasks at the same level.* The decision-making tasks at the same level of the decision tree have exactly the same learning structure, i.e., selecting a child sub-cluster from the K candidates. Therefore, their RL agents can be *reused* by each other. The reusability of RL agents enables model generalization. That is, the knowledge

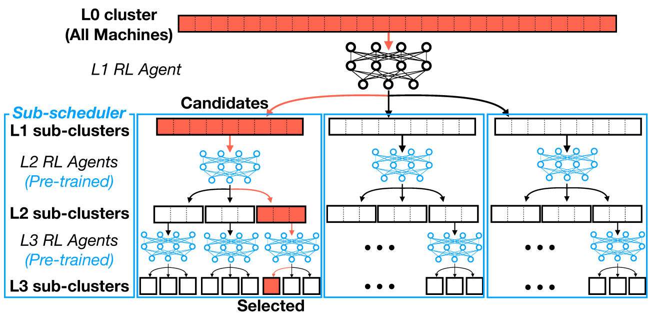


Fig. 6: Two-level hierarchical scheduling where a dedicated RL agent makes a high-level placement decision and gets it refined in a sub-cluster by the pretrained sub-scheduler.

learned in one subtask can be transferred to another, which, in turn, accelerates the learning process.

To demonstrate the benefits of our divide-and-conquer (D&C) approach, we refer back to Fig. 4, where we depict the learning curve of our approach in the 81-node cluster (the blue, solid curve). Our D&C approach with $K = 3$ easily outperforms vanilla RL, improving the average RPS by ~30% at the end of the training. Such a significant improvement is achieved due to the exponentially reduced state space and the enhanced exploration ability.

B. Augmenting D&C with Pretrained Model

Motivation Training a dedicated RL model using our D&C approach significantly improves the learning quality. However, the training process remains *time-consuming*, causing a long scheduling delay, e.g., in the previous experiment, it takes one hour to learn the placements of 200 containers (Fig. 4). The root cause of such a long training time is the *fine-grained* decision making. That is, for each container, the placement decision must narrow down to a specific machine. As the RL model learns the optimal *placement combination* for a batch of containers, the scheduling decision space grows *exponentially* with the cluster size (§IV-C). Note that our D&C approach does not address this problem, as it still makes a fine-grained placement decision for each container.

Combining Dedicated Model and Pretrained Model To reduce the scheduling decision space, we propose to train a *dedicated* RL model that makes high-level placement decisions *at the granularity of sub-clusters*. That is, the dedicated model only learns to find which sub-cluster a container should be scheduled, without narrowing down to a specific host machine in it. These coarse-grained placement decisions are then handed over to a *pretrained unified RL model* for refinement. Specifically, we *offline train* a unified scheduler in a sub-cluster for general workloads, and use it online to locate the host machines for containers in that sub-cluster.

Intuitively, this approach combines the benefits of both training a dedicated model for high-quality placement and using a pretrained unified model for fast online decision making. Our D&C approach naturally enables this combination. As illustrated in Fig. 6, we split the decision tree into two *vertical* parts. In the upper part, we train a dedicated RL model at the arrival of a container group. The dedicated RL model only makes high-level decisions to dispatch containers to sub-clusters. In the lower part, we refine their placements to host machines using a pretrained unified model. We refer to the entire design as the *core* of Metis, which essentially performs *hierarchical scheduling in two levels*.

Training Sub-scheduler Offline We stress that offline training a unified RL model for general container workloads in a small sub-cluster is *computationally feasible* using our D&C approach, as it only needs to handle a limited number of container combinations. The intuition behind is that those low-level decision makings, e.g., selecting a host machine for a container within a server rack, are inherently *atomic*, as they only perform some simple skills, episodically or cyclically. These atomic learning tasks can be solved by training a unified RL model in advance. The trained RL model can then be used to make *online* placement decisions for *any* container group in that sub-cluster, hence providing a plug-in scheduling service. We call such an offline-trained RL model a *sub-scheduler*. In a homogeneous cluster, a sub-scheduler trained for one sub-cluster can be directly *reused* by another as both sub-clusters have machines of the same configurations.

Learning Speedup vs. Performance Loss Sub-schedulers substantially reduce the scheduling decision space: instead of scheduling C containers onto N machines, the dedicated RL model now places those containers to K sub-clusters, reducing the scheduling decision space from $O(N^C)$ to $O(K^C)$, where $K \ll N$. The reduced scheduling decision space results in dramatic learning speedup. Continuing our experiments in an 81-node cluster, we offline train a sub-scheduler in a 27-node sub-cluster. We depict the learning curve of Metis using pretrained sub-schedulers in Fig. 4 (the orange, dashed curve), where we observe ~10× training speedup over the plain D&C.

On the other hand, we notice a slight RPS loss ($< 3\%$ in Fig. 4) of using sub-schedulers, as they are trained offline for *general workloads* and hence cannot achieve the optimal performance for a given container batch like a dedicated agent. In general, using a sub-scheduler in a larger sub-cluster results in higher model reusability and faster training of the high-level RL agent (hence shorter scheduling latency), at the expense of increased performance loss. Metis balances this tradeoff by configuring an appropriate size of sub-schedulers (§VII-C).

VI. Implementation

We have implemented Metis as a pluggable scheduler in Docker Swarm [42]. Our design can also be easily ported to the other container-orchestration frameworks.

Scheduling Workflow Fig. 7 illustrates the architecture overview and the system workflow. Metis receives the container launching requests and aggregates them in groups (①). For a container group, Metis trains an RL Agent to make a placement decision (②). The Container Launcher then executes the learned decision by placing containers onto the selected nodes (③ and ④). The performance of these containers are monitored by the Performance Profiler (⑤). The

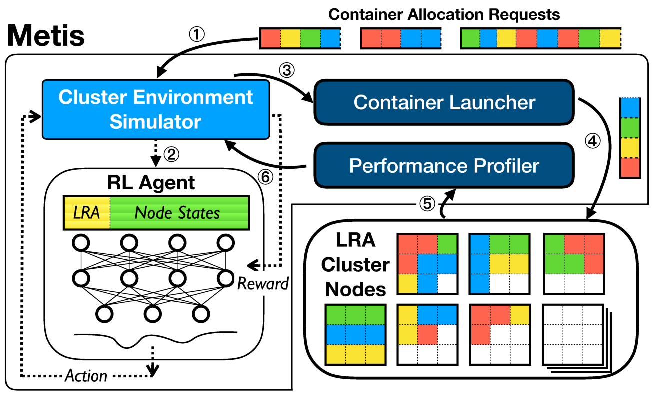


Fig. 7: Metis Overview.

collected performance data is then used as training samples by the Environment Simulator to further improve its fidelity (⑥).

Cluster Environment Simulator and RL Agent Our implementation exposes an interface to RL agents in the Cluster Environment Simulator similar to that of OpenAI Gym [77], [78]. To evaluate the outcome of a placement policy, the simulator predicts the performance (i.e., RPS) of containers on each node by taking the *node state* as input, which is a vector of the number of containers running on that node for each application (see §IV-A). In other words, it learns the *mapping* between the container co-location information (*input*) and their normalized throughput (*output*) from traces [46] or benchmark results. Specifically, we build a `RandomForestRegressor` with scikit-learn [79], which makes an ensemble of 100 regression decision trees with a maximum depth of 20.[3] Compared with the deep neural network-based simulators [41] which are compute-intensive and data hungry [80], our ensemble method is more efficient. It learns thousands of samples in tens of seconds and makes predictions in sub-seconds, enabling frequent model re-training with the latest traces. As an ensemble method, it also shows a strong resistance to overfitting even supplying with a large number of repetitive training samples [66]. We will validate these benefits in §VII-E.

Note that our simulator does *not* explicitly model per-container shared resources such as CPU cache, disk I/O, or memory bandwidth. Instead, it aggregates all resource contention effects for each container and predicts their performance impact using metrics like RPS. The reason for such "coarse-grained" simulation is the lack of detailed information. Unlike CPU cores or memory capacity, collecting per-container usage of shared resources requires hardware- and software-level monitoring mechanisms [29], [30], [32], [81], which are expensive to deploy in production clusters.

We have implemented the RL Agent in 500 lines of Python code and trained a 2-layer policy neural network with TensorFlow [8].

Container Launcher Our Container Launcher is compatible with many container-orchestration frameworks, provided that they support per-container placement specification to a single machine, e.g., through labels in Kubernetes [24] and Docker Swarm [42]. Containers with I/O-dependencies can be connected through overlay networks or by exposing their endpoints for service discovery.

[3]Having more or deeper trees barely improves accuracy in this setting.

VII. Evaluation

In this section, we evaluate Metis on EC2 clusters against container workloads of seven real applications[4]. Our evaluations aim to address the following three questions: (1) How does Metis perform compared to prior constraint-based LRA schedulers? (§VII-B) (2) Can Metis scale to large clusters, and how does each of our HRL designs described in §V contribute to its scalability? (§VII-C) (3) Can Metis support various scheduling objectives? (§VII-D)

A. Methodology

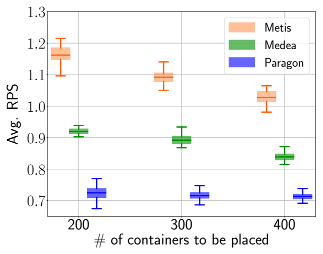
Cluster Setting Most of our evaluations are carried out in two clusters, a medium one with 81 nodes and a large one with 729 nodes. Each node is an `m5.4xlarge` instance with 16 vCPUs, 64 GB memory, and up to 10 Gbps network. We run applications in Docker containers, each with 2 vCPUs and 8 GB memory. Up to eight containers can run in one node.

Workloads We have implemented seven long-running applications covering machine learning, stream processing, I/O and storage services. Fig. 2 depicts their performance interactions.

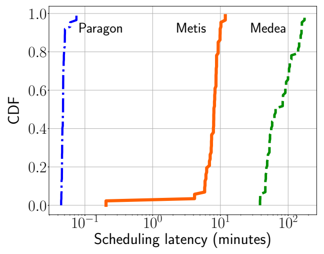
- **Redis** [5]: a stand-alone Redis server instance responding to the `redis-benchmark` [82] requests with 20K basic operations each.
- MXNet Model Server (**MMS**) [11]: an image classification service, using ResNet-152 [83] in MXNet.
- Image Super Resolution (**ISR**) [52]: a CPU service that super-scales low-resolution images using Residual Dense Network [84] with Tensorflow [8].
- File Checksum (**CKM**) [53]: a service that loads hundreds of files (several MB each) from disk, hashes each file with SHA-512, and verifies the checksum values.
- Yahoo! Cloud Serving Benchmark (**YCSB**) workload A and B [55]: the standard benchmarking suites for stream processing, which generate either read-write-balanced (YCSB-A) or read-heavy (YCSB-B) workloads [55] and feed them to the underlying Memcached [6] daemon.
- Video Scene Detection (**ScD**) [54]: a service that detects scene changes by comparing the difference between each two subsequent frames in videos of two minutes.

We configure a constant high rate (4 requests/sec) for all applications except YCSB—for which the client issues as many operations as it can. Admittedly, this may deviate from a more realistic setting where workloads change diurnally. Yet, faithfully synthesizing those patterns requires access to production traces, which we do not have. According to our contacts at Alibaba, core LRA services have peak loads overlapped in a few hours of a day. Our configuration hence targets

[4]All the benchmark workloads and baseline schedulers in our evaluations are available at `https://github.com/Metis-RL-based-container-sche/Metis`. The details are given in the Artifact Description.

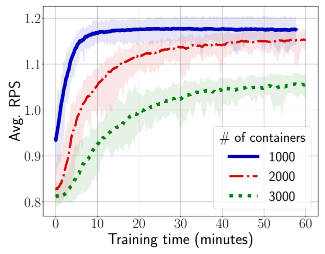


(a) Average RPS with various container group sizes.

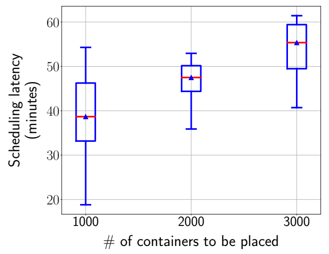


(b) Distribution of scheduling latency in all container groups.

Fig. 8: Comparison of Metis, Medea and Paragon in an 81-node cluster. Boxes depict the 25th, 50th, and 75th percentiles, and whiskers the 5th and 95th.



(a) Learning curve of the RL Agent.



(b) Scheduling latency.

Fig. 9: Metis's scalability in a large cluster with 729 nodes.

the most challenging scenario, where sustained high load stretches the processing capabilities of all LRA containers.

Metrics We adopt RPS (requests per second) as the main performance metric for containers [18], [30]. For equal treatment for all applications, we normalize the RPS of a container by its *stand-alone* RPS which is measured when the container *runs alone* on a machine.

Baselines We evaluate Metis against two constraint-based schedulers with state-of-the-art performance:

1) ***Medea*** [18] requires the explicit specifications of placement constraints. To this end, we predict the container performance in co-location for any two LRAs using our cluster environment simulator. We specify an affinity (anti-affinity) constraint for a container if co-location leads to an RPS improvement (reduction) over 10%.[5] This allows us to provide a high-quality constraint set to Medea, based on which it solves an ILP problem using the branch-and-bound heuristic [85]. Medea was originally implemented in YARN. We ported it to Docker Swarm, where we use the MATLAB ILP solver.

2) ***Paragon*** [29] automatically characterizes resource interferences between two LRAs by computing their sensitivity scores against each other. We use the same profiling method as mentioned above, and align the scheduling requirements with the 10% RPS variation threshold. Paragon schedules containers using a simple greedy algorithm. As Paragon is not open-sourced, we reimplemented it in Python.

B. Scheduling Performance

We first compare Metis and the two constraint-based schedulers in an 81-node cluster, with container group size varying from 200 to 400. For each group size, we repeat the experiment 30 times with random containers combinations. We run Metis in full functions using both D&C placement (§V-A) and a pretrained sub-scheduler in a 27-node sub-cluster (§V-B).

Fig. 8a compares the average RPS under three schedulers with different group sizes. Metis consistently outperforms Medea and Paragon, achieving up to 25% and 61% higher RPS, respectively. Compared with Metis, Medea only achieves sub-optimal performance as it simply minimizes constraint violations without differentiating their performance impacts (see §III-A). Paragon performs even worse for two reasons. First, it only profiles anti-affinity with interference scores, while leaving the affinity requirements unattended. Second, it employs a simple greedy scheduling policy that sequentially places containers to minimize interferences at each step. In comparison, Medea and Metis place containers in batches, enabling them a "global view" to make optimal decisions that account for the complex interactions between those containers.

Fig. 8b compares the scheduling latencies of the three schedulers with different group sizes. Owing to its simple greedy scheduling logic, Paragon has the shortest latency in only a few seconds. Metis, on the other hand, employs a more complex RL algorithm, but still finds the optimal placement in less than 10 minutes thanks to its highly optimized design. The longest latency is measured in Medea, as it needs to solve a complex ILP problem involving 100K constraints as discussed in §III-A. To summarize, Metis offers the best solution out of the three schedulers in that it achieves the highest RPS with only a modest scheduling latency.

C. Scalability

Metis Performance at Production Scale We next evaluate the scalability of Metis in a large cluster with 729 nodes. We trained a unified sub-scheduler offline in a 27-node sub-cluster. Experiments of each setting are repeated 30 times with randomly generated groups of 1000, 2000, or 3000 containers.

Fig. 9a depicts how the RPS improves during training under different groups sizes. Even in such a large cluster, Metis still provides placements of high performance (RPS ranging from 1.0 to 1.2 for different settings), which is comparable to that in previous medium-sized clusters (Fig. 8a). Concerning the scheduling latency, Fig. 9b shows Metis can still make *timely* scheduling decisions within one hour. Paragon gives undesirable performance similar to the previous; Medea takes over ten hours to solve the ILP problem with 10M constraints, which is unacceptable; their results are hence omitted.

A Deep Dive into the Hierarchical RL Design To quantify how our two designs, D&C placement and pretrained sub-scheduler, contribute to Metis' scalability, we revisit Fig. 4 for its performance in an 81-node cluster.

[5]We grid search the RPS variation for constraint specification and find that the 10% threshold results in the best performance for Medea.

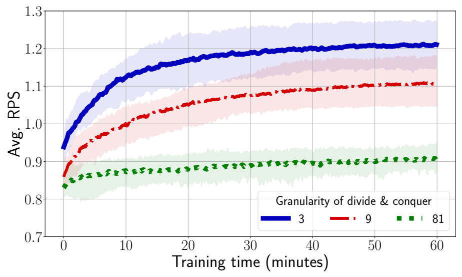
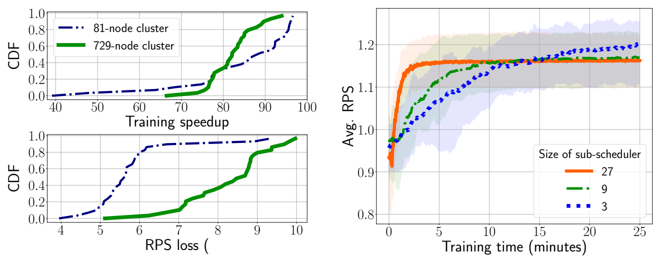


Fig. 10: Sensitivity analysis of K, the sub-cluster decomposition degree, in D&C (without sub-schedulers).



(a) Training speedup and RPS loss of agents with a 27-node sub-scheduler.

(b) Learning curve of RL agents with different sub-scheduler sizes.

Fig. 11: Performance-latency trade-off with sub-scheduler.

1) Higher-quality decisions with D&C placement. In Fig. 4, compared with vanilla RL, D&C makes better scheduling decisions by decoupling the cluster into 3 partitions recursively ($K = 3$) and improves the average RPS by 35.2% (from 0.91 to 1.23). It is because D&C shrinks both state and action space of the RL agents (§V-A), allowing them to take less various inputs, learn much fewer state and action mappings, and explore different placement strategies more efficiently.

2) Faster decisions with pretrained sub-schedulers. Fig. 4 confirms that employing sub-schedulers accelerates the training by $12\times$, as it substantially reduces the scheduling decision space (§V-B). Specifically, the $O(81^{200})$ possible outcomes of scheduling 200 containers in an 81-node drops to $O(3^{200})$ ones after deploying a 27-node sub-cluster scheduler.

Decomposition Degree K of Divide-and-Conquer To better illustrate the learning sensitivity to the hyper-parameter K—the sub-cluster decomposition degree of our D&C technique, we depict the training curves in an 81-node cluster with various K values (3, 9, and 81)[6] as Fig. 10. As we can see, D&C with a smaller decomposition degree provides placements with better performance. It is because smaller K indicates smaller state and action space in each subtask, which further augments the learning capability and avoids getting stuck in local optima.

Performance Impact of Sub-scheduler Design Admittedly, sub-scheduler design accelerates the training at the cost of lower decision quality (e.g., RPS loss). To quantify the impacts, we generate 90 container groups of various size for Metis to schedule, and compare the performance between the RL agents with sub-scheduler and those without such design. As shown in Fig. 11a, the sub-scheduler design (with 27-node sub-cluster) dramatically accelerates RL convergence by $40\times$–$95\times$, with less than 10% loss of RPS. Compared to sub-scheduler of smaller sizes (3 or 9 nodes) as shown in Fig. 11b, the larger sub-scheduler also leads to faster learning processes and slightly lower RPS in the end. We therefore think trading quality for latency with the sub-scheduler design is well justified in large clusters.

[6]In this case, D&C with $K = 81$ is *equivalent* to the vanilla RL. Unless otherwise specified, D&C adopts $K = 3$ in other parts of our evaluation.

D. Support of Various Scheduling Objectives

We next show that Metis is a general-purpose scheduler that can be used to optimize various objectives beyond RPS. The experiments are conducted in an 81-node cluster.

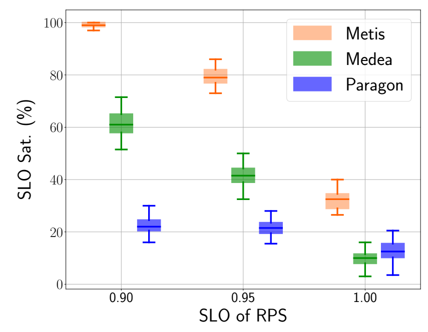
Case Study #1: Maximizing SLO Satisfactions Metis can directly maximize the SLO satisfactions—the percentage of containers whose RPS meets a specified requirement—by configuring the RL reward function (§IV-A) as the SLO satisfaction rate. Fig. 12a compares the SLO satisfaction rate with different schedulers. Metis consistently achieves much higher SLO satisfactions than Medea and Paragon. When the required RPS is set to 0.9, Metis can meet this SLO for almost all containers, outperforming Medea and Paragon by $1.6\times$ and $4.4\times$ on average, respectively. Even with a rather demanding SLO requirement (RPS $\geq$ 1.0), Metis still achieves notably higher SLO satisfaction rate (32%) than the two baselines.

To understand the advantage of Metis over constraint-based schedulers, we refer to Fig. 12b for the distribution of container throughput measured under Metis and Medea with target RPS $\geq$ 0.9. Compared with Medea, Metis has a curve with a much shorter tail and a close-to-zero "knee" at the target RPS—a clear sign of sacrificing high-RPS containers in exchange for more others to meet the SLO. For comparison, we also plot the RPS distribution (red dashed curve) when Metis is configured to maximize the average throughput, without concerning about the SLO compliance. This time, it sacrifices a small number of low-RPS containers for more to achieve higher throughput. Metis can thus adapt to various scheduling objectives by making precise placement decisions.

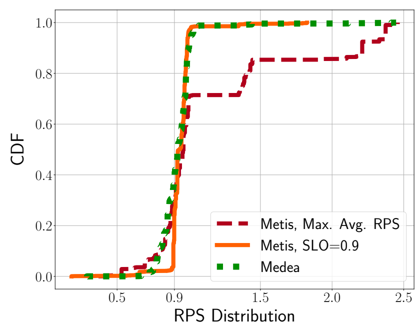
Case Study #2: Minimizing Resource Fragmentation Minimizing the resource fragmentation requires packing as many containers as possible into as few machines, leading to lower container performance due to resource contention. Metis easily balances the two conflicting objectives by setting the reward function as the weighted sum of RPS and vacant machines with a tunable knob β, as shown in Fig. 12c. Compared with maximizing RPS only ($\beta = 0.0$), setting the knob to 0.5 only sightly decreases the container performance while saving around 12% of machines. Continuing increasing β saves more machines at the expense of lower RPS.

E. Cluster Environment Simulator

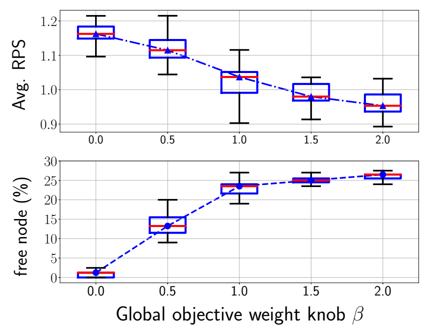
As RL training critically relies on the prediction given by the Cluster Environment Simulator (§VI), we validate its accuracy with varying training samples. In particular, we profile



(a) Maximizing SLO satisfactions.

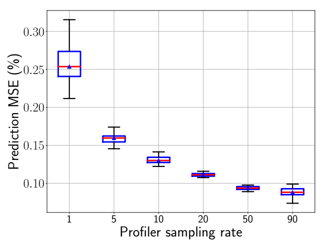


(b) RPS distribution in (a).

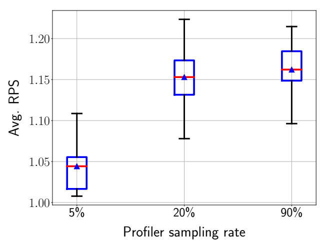


(c) Balancing average RPS and vacant nodes.

Fig. 12: Metis supports various scheduling objectives. Boxes depict the 25th, 50th, and 75th percentiles; whiskers depict the 5th and 95th percentiles.



(a) Prediction accuracy.



(b) Scheduling performance.

Fig. 13: Prediction accuracy of the simulator and the scheduling performance of Metis with varying data sampling rates in an 81-node cluster.

the resultant container RPS in all 6435 possible container co-location options of the seven LRAs on one machine. We randomly sample the training sets covering 1%–90% of the profiled results—the remaining data is used as the test set. We train the predictor of our simulator, a multivariate Random Forests regression model (an ensemble of 100 decision trees with maximum depth of 20), and evaluate its accuracy in terms of mean square error (MSE) over the test set. Fig. 13a shows a box plot of the accuracy results averaged by 30 runs with different sampling rates. As observed, profiling 10% or more co-location combinations as dataset is sufficient to train an accurate simulator with prediction MSE $\leq 0.15\%$.

We further investigate how the simulator's prediction accuracy, given various sampling coverage, may affect the quality of RL training. We evaluate the RPS performance of Metis with three traces that respectively cover 5%, 20%, and 90% possible combinations of container co-locations. As shown in Fig. 13b, the more co-location cases a trace can cover, the more accurate predictions the simulator makes, and the higher RPS the learned scheduler achieves. In this case, having a log trace covering 20% of the co-location cases in one machine is sufficient for Metis to achieve the near-optimal performance.

VIII. Conclusion and Future Work

In this paper, we demonstrated the inefficiency of scheduling LRA containers by means of satisfying placement constraints. We presented Metis, an intelligent scheduler driven by novel *hierarchical reinforcement learning* (HRL) techniques tailored to LRA scheduling. This not only eliminates the manual specification of rather complex container interactions, but also, for the first time, offers concrete quantitative scheduling guidelines on placing LRA containers with performance prediction and iterative refinement. Specifically, Metis decomposes a complex scheduling problem into a hierarchy of simple tasks, in which it progressively learns the container placement at different levels of granularity, from sub-clusters to individual machines. Metis trains dedicated RL agents for quality decision-making at a high level which schedules containers to sub-clusters. It then uses a pretrained unified RL agent to quickly narrow down their placements in a sub-cluster. EC2 deployment shows that Metis substantially improves the performance of LRA containers over the existing constraint-based schedulers, and can scale to a large cluster running thousands of containers.

Despite the promising performance in LRA scheduling, Metis's HRL design can only afford to handle dozens of core applications that are performance-critical. Yet, production clusters run thousands of LRA services, yielding a huge number of combinations of the cluster state and the interference patterns. Also, our current design does not account for the diverse workload patterns that may change over time. We plan to tackle these issues in our future work.

Acknowledgement

We thank Yinghao Yu and Haiyang Ding from Alibaba Cloud for their detailed feedbacks that helped improve the earlier version of this paper. We also thank the anonymous reviewers of SC20 for their valuable comments. This research was supported in part by RGC GRF grants under the contracts 26213818, 16206417 and 16207818. Qizhen Weng was supported in part by the Hong Kong PhD Fellowship Scheme.

References

[1] "Apache flink," https://flink.apache.org.
[2] "Apache storm," https://storm.apache.org.
[3] "Apache kafka," https://kafka.apache.org.

[4] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proc. ACM SOSP*, 2013.

[5] S. Sanfilippo, "Redis: an open source, in-memory data structure store," https://redis.io.

[6] "Memcached," https://memcached.org.

[7] "Apache hbase," https://hbase.apache.org.

[8] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proc. USENIX OSDI*, 2016.

[9] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[10] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *ArXiv*, vol. abs/1512.01274, 2015, https://mxnet.incubator.apache.org.

[11] "Model server for apache mxnet," https://github.com/awslabs/mxnet-model-server.

[12] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system." in *Proc. USENIX NSDI*, 2017.

[13] H. Tian, M. Yu, and W. Wang, "Continuum: A platform for cost-aware, low-latency continual learning," in *Proc. ACM SoCC*, 2018.

[14] A. Gujarati, S. Elnikety, Y. He, K. S. McKinley, and B. B. Brandenburg, "Swayam: distributed autoscaling to meet slos of machine learning inference services with resource efficiency," in *Proc. ACM/IFIP/USENIX Middleware*, 2017.

[15] "Apache hadoop," https://hadoop.apache.org.

[16] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark sql: Relational data processing in spark," in *Proc. ACM SIGMOD*, 2015.

[17] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. USENIX NSDI*, 2012.

[18] P. Garefalakis, K. Karanasos, P. Pietzuch, A. Suresh, and S. Rao, "Medea: scheduling of long running applications in shared production clusters," in *Proc. ACM EuroSys*, 2018.

[19] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao, "Who limits the resource efficiency of my datacenter: an analysis of alibaba datacenter traces," in *Proc. ACM IWQoS*, 2019.

[20] Q. Liu and Z. Yu, "The elasticity and plasticity in semi-containerized co-locating cloud workload: a view from alibaba trace," in *Proc. ACM SoCC*, 2018.

[21] H. Wu, W. Zhang, Y. Xu, H. Xiang, T. Huang, H. Ding, and Z. Zhang, "Aladdin: Optimized maximum flow management for shared production clusters," in *Proc. IEEE IPDPS*, 2019, pp. 696–707.

[22] Y. Cheng, Z. Chai, and A. Anwar, "Characterizing co-located datacenter workloads: An alibaba case study," *arXiv preprint arXiv:1808.02919*, 2018.

[23] "Spark streaming," https://spark.apache.org/streaming.

[24] "Kubernetes: Production-grade container orchestration," https://kubernetes.io.

[25] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proc. ACM SoCC*, 2013.

[26] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proc. ACM Eurosys*, 2015.

[27] T. Knaup and F. Leibert, "Marathon: A container orchestration platform for Mesos and DC/OS," http://mesosphere.github.io/marathon, 2019.

[28] "Apache aurora," https://aurora.apache.org.

[29] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *Proc. ACM ASPLOS*, vol. 48, no. 4, 2013, pp. 77–88.

[30] C. Delimitro and C. Kozyrakis, "Quasar: resource-efficient and qos-aware cluster management," *Proc. ACM ASPLOS*, vol. 49, no. 4, pp. 127–144, 2014.

[31] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," *Proc. ACM ISCA*, vol. 41, no. 3, pp. 607–618, 2013.

[32] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proc. ACM ISCA*, vol. 43, no. 3, 2015, pp. 450–462.

[33] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini, "Deepdive: Transparently identifying and managing performance interference in virtualized environments," in *Proc. USENIX ATC*, 2013.

[34] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: managing performance interference effects for qos-aware clouds," in *Proc. ACM Eurosys*, 2010.

[35] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters," in *Proc. ACM EuroSys*, 2016.

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

[37] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proc. ACM HotNets*, 2016.

[38] Z. Hu, J. Tu, and B. Li, "Spear: Optimized dependency-aware task scheduling with deep reinforcement learning," in *Proc. IEEE ICDCS*, 2019.

[39] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proc. ACM SIGCOMM*, 2019.

[40] L. Chen, J. Lingys, K. Chen, and F. Liu, "Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization," in *Proc. ACM SIGCOMM*, 2018.

[41] Y. Bao, Y. Peng, and C. Wu, "Deep learning-based job placement in distributed machine learning clusters," in *Proc. IEEE INFOCOM*, 2019.

[42] "Docker swarm," https://github.com/docker/swarm.

[43] "Amazon elastic compute cloud (amazon ec2)," https://aws.amazon.com/ec2.

[44] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino, "Apache tez: A unifying framework for modeling and building data processing applications," in *Proc. ACM SIGMOD*, 2015.

[45] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: Sql and rich analytics at scale," in *Proc. ACM SIGMOD*, 2013.

[46] "Alibaba production cluster data," https://github.com/alibaba/clusterdata.

[47] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proc. ACM ASPLOS*, 2019.

[48] A. Cockcroft, "Microservices workshop: Why, what, and how to get there," https://www.slideshare.net/adriancockcroft/microservices-workshop-craft-conference.

[49] Y. Amannejad, D. Krishnamurthy, and B. Far, "Detecting performance interference in cloud-based web services," in *Proc. IFIP/IEEE IM*, 2015.

[50] J. Han, S. Jeon, Y.-r. Choi, and J. Huh, "Interference management for distributed parallel applications in consolidated clusters," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 443–456, 2016.

[51] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, Í. Goiri, S. Krishnan, J. Kulkarni *et al.*, "Morpheus: Towards automated slos for enterprise clusters," in *Proc. USENIX OSDI*, 2016.

[52] F. C. et al., "Isr," https://github.com/idealo/image-super-resolution, 2018.

[53] "hashlib — secure hashes and message digests," https://docs.python.org/3/library/hashlib.html.

[54] B. Castellano, "Pyscenedetect: Python and opencv-based scene cut/transition detection program & library." https://github.com/Breakthrough/PySceneDetect, 2019.

[55] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proc. ACM SoCC*, 2010.

[56] L. Liu and H. Xu, "Elasecutor: Elastic executor scheduling in data analytics systems," in *Proc. ACM SoCC*, 2018.

[57] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," in *Proc. ACM SIGCOMM*, 2017.

[58] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," in *Proc. ICML*, 2017.

[59] Y. Gao, L. Chen, and B. Li, "Spotlight: Optimizing device placement for training deep neural networks," in *Proc. ICML*, 2018.

[60] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Proc. NeurIPS*, 2000.

[61] H. Mao, S. B. Venkatakrishnan, M. Schwarzkopf, and M. Alizadeh, "Variance reduction for reinforcement learning in input-driven environments," *arXiv preprint arXiv:1807.02264*, 2018.

[62] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[63] T. Degris, M. White, and R. S. Sutton, "Off-policy actor-critic," *arXiv preprint arXiv:1205.4839*, 2012.

[64] Anonymous, "Metis: Learning to schedule long-running applications in shared container clusters at scale (supplemental material)," https://bit.ly/3blZMgc.

[65] L. Thamsen, B. Rabier, F. Schmidt, T. Renner, and O. Kao, "Scheduling recurring distributed dataflow jobs based on resource utilization and interference," in *Proc. IEEE Big Data*, 2017.

[66] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[67] A. G. Barto and S. Mahadevan, "Recent advances in hierarchical reinforcement learning," *Discrete event dynamic systems*, vol. 13, no. 1-2, pp. 41–77, 2003.

[68] C. Diuk, A. Schapiro, N. Córdova, J. Ribas-Fernandes, Y. Niv, and M. Botvinick, "Divide and conquer: hierarchical reinforcement learning and task decomposition in humans," in *Computational and robotic models of the hierarchical organization of behavior*. Springer, 2013, pp. 271–291.

[69] Y. Wu, E. Mansimov, R. B. Grosse, S. Liao, and J. Ba, "Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation," in *Proc. NeurIPS*, 2017.

[70] L. Li, T. J. Walsh, and M. L. Littman, "Towards a unified theory of state abstraction for mdps." in *Proc. ISAIM*, 2006.

[71] T. G. Dietterich, "Hierarchical reinforcement learning with the maxq value function decomposition," *Journal of artificial intelligence research*, vol. 13, pp. 227–303, 2000.

[72] T. J. Walsh, S. Goschin, and M. L. Littman, "Integrating sample-based planning and model-based reinforcement learning," in *Proc. AAAI*, 2010.

[73] O. Nachum, S. Gu, H. Lee, and S. Levine, "Data-efficient hierarchical reinforcement learning," in *Proc. NeurIPS*. Curran Associates, Inc., 2018, pp. 3307–3317.

[74] A. S. Vezhnevets, S. Osindero, T. Schaul, N. Heess, M. Jaderberg, D. Silver, and K. Kavukcuoglu, "Feudal networks for hierarchical reinforcement learning," in *Proc ICML*, vol. 70. JMLR. org, 2017, pp. 3540–3549.

[75] O. Nachum, S. Gu, H. Lee, and S. Levine, "Near-optimal representation learning for hierarchical reinforcement learning," *arXiv preprint arXiv:1810.01257*, 2018.

[76] A. C. Li, C. Florensa, I. Clavera, and P. Abbeel, "Sub-policy adaptation for hierarchical reinforcement learning," *arXiv preprint arXiv:1906.05862*, 2019.

[77] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[78] H. Mao, P. Negi, A. Narayan, H. Wang, J. Yang, H. Wang, R. Marcus, R. Addanki, M. K. Shirkoohi, S. He, V. Nathan, F. Cangialosi, S. B. Venkatakrishnan, W.-H. Weng, S.-W. Han, T. Kraska, and M. Alizadeh, "Park: An open platform for learning-augmented computer systems," in *Proc. NeurIPS*, 2019.

[79] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[80] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[81] "User space software for intel(r) resource director technology," https://github.com/intel/intel-cmt-cat.

[82] "Redis-benchmark," https://redis.io/topics/benchmarks.

[83] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE CVPR*, 2016.

[84] Y. Zhang, Y. Tian, Y. Kong, B. Zhong, and Y. Fu, "Residual dense network for image super-resolution," in *Proc. IEEE CVPR*, 2018.

[85] E. L. Lawler and D. E. Wood, "Branch-and-bound methods: A survey," *Operations research*, vol. 14, no. 4, pp. 699–719, 1966.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

1 DESCRIPTION

We implement our proposed Metis as a pluggable scheduler in Docker Swarm using Python 3.

The experiment package includes:

(1) Seven LRA workloads and cluster framework with scheduler
(2) Scheduling Algorithms: Metis
(3) Scheduling Baseline Algorithms: Medea, Paragon, and Vanilla RL. In specific, our Medea implementation takes the input of placement constraints we profiled and solves an ILP problem using a MATLAB solver. Paragon is not open-sourced, so we implemented its algorithms in Python, including the interference scoring and the greedy scheduling policy.

2 HARDWARE CONFIGURATION

The hardware information in reproducing the experimental results can be found here:

```
$ http://github.com/
$ Metis-RL-based-container-sche/
$ Metis/blob/master/environment_info
```

3 INSTALLATION

The experiment package can be obtained from Github.

The whole project is associated with the DOI:

```
$ http://dx.doi.org/10.5281/zenodo.3859335
```

Clone from Github source:

```
$ git clone https://github.com/
Metis-RL-based-container-sche/Metis.git
```

Install Requirements:

```
$ pip3 install -r requirement.txt
```

4 EXPERIMENT WORKFLOW

Our experiment is built with three steps:

(1) Set up the cluster with our seven LRAs, collecting the container co-location samples and establish the simulator.
(2) Run scheduling algorithms to obtain the container placement decision.
(3) Place containers in the scheduling group according to the scheduling decision, collecting the scheduling performance, e.g., average container RPS.

We next describe these three steps in more detail.

4.1 Real-World LRA Cluster Setup and Profiler Establishment

1. Launch a cluster of tens of nodes on Amazon EC2 consisting of at least 1 manager node, several worker nodes, and some client nodes. Record the public DNS address or ip address of each node and make sure the manager and client nodes are accessible through the SSH key: `$HOME/.ssh/id_rsa`.

```
$ export MANAGER=xxx.xxx.xxx.100
$ export WORKER1=xxx.xxx.xxx.101
$ export CLIENT1=xxx.xxx.xxx.201
$ ssh ubuntu@$MANAGER
(manager)$ git clone https://github.com/
Metis-RL-based-container-sche/Metis.git
```

2. If Docker has not been installed, install Docker on each manager, worker, and client node:

```
(manager)$ sudo Cluster/scripts/install_docker.sh
(worker1)$ sudo Cluster/scripts/install_docker.sh
```

3. Coordinating manager and worker nodes through Docker Swarm to form a Swarm cluster.

```
$ ssh $MANAGER docker swarm init
```

Output:

```
To add a worker to this swarm, run the following command:
    docker swarm join --token SWMTKN-···
    ··· xxx.xxx.xxx.100:2377
```

Then execute the suggested command on each worker.

```
$ ssh $WORKER1 docker swarm join --token
SWMTKN-······ xxx.xxx.xxx.100:2377
```

Output:

```
This node joined a swarm as a worker.
```

4. Build docker images of seven workloads on the manager node.

```
(manager)$ cd Cluster/workloads
(manager)$ ./build-all.sh
```

The seven workloads are: 1. Redis; 2. MXNet Model Server; 3. Model Checksum; 4. Image Super Resolution; 5. Yahoo! Cloud Streaming Benchmark A; 6. Yahoo! Cloud Streaming Benchmark B; 7. Video Scene Detection. They are modified to respond to HTTP requests and execute standard processes.

5. Launch certain workloads on certain worker nodes from the manager node.

```
(manager)$ cd Cluster/scripts
```

For example, launching one container for each workload on `WORKER1`, where `0` indicates idle and 1 to 7 indicate different workloads.

```
(manager)$ ./service-launching.sh
$WORKER1 0 1 2 3 4 5 6 7
```

As another example, launch 3 workload-1, 2 workload-2, and 1 workload-3 container on `WORKER2`:

```
(manager)$ ./service-launching.sh
$WORKER2 0 0 1 1 1 2 2 3
```

6. Sending requests from the client node to the worker node

```
$ ssh ubuntu@$CLIENT1
(client1)$ git clone https://github.com/
Metis-RL-based-container-sche/Metis.git
(client1)$ export WORKER1=xxx.xxx.xxx.101
```

Send single request for testing:

```
(client1)$ curl $WORKER1:8081
```

Or pressure the application with locust or other tools, e.g.,

```
(client1)$ cd Cluster/scripts
(client1)$ python3 parallel_locust.py $WORKER1
```

Default log path lies in `Cluster/scripts/log`

7. (Optional) Collecting performance benchmark datasets through automatically deploying, profiling, terminating, and re-deploying.

Here the `0-1-2-3-4-5-6-7` or `0-0-1-1-1-2-2-3` means combination of different workloads on each node (as described step 5). Each node has the maximum container capacity of 8.

```
(manager)$ ./profiling-go.sh
$WORKER1 "0-1-2-3-4-5-6-7 0-0-1-1-1-2-2-3"
```

8. Terminate the swarm cluster on the manager node.

```
(manager)$ docker swarm leave --force
```

4.2 Schedule Containers with Metis

Metis is implemented with both divide-and-conquer and sub-scheduler techniques.

1. First check the data collected by Real-World LRA cluster is stored in the folder:

```
$ cd Experiments/
$ ls ./simulator/datasets/
***_sample_collected.npz
```

2. Train sub-schedulers in a 27-node sub-cluster:

```
$ cd Experiments/
$ ./shell/TrainSubScheduler.sh
```

Output: the well-trained sub-scheduler models, as well as corresponding log files will be store in the folder:

```
$ ls ./checkpoint/
subScheduler_*/
```

3. High-level model training based on previously well-trained sub-schedulers.

(0) Check the sub-scheduler models are stored in the folder:

```
$ cd Experiments/
$ ls ./checkpoint/
subScheduler_*/
```

Check the container batches data is stored in the folder or create your own batches:

```
$ ls ./data
batch_set_200.npz batch_set_300.npz
batch_set_400.npz batch_set_1000.npz
batch_set_2000.npz batch_set_3000.npz
```

(1) High-level training in a medium-sized cluster of 81 nodes:

```
$ ./shell/RunHighLevelTrainingMedium.sh 200
$ ./shell/RunHighLevelTrainingMedium.sh 300
$ ./shell/RunHighLevelTrainingMedium.sh 400
```

Output: the training log files including the RPS, placement matrix, training time duration, etc. will be store in the folder:

```
$ ls ./checkpoint/
81nodes_*_*/
```

(2) High-level training in a large cluster of 729 nodes:

```
$ ./shell/RunHighLevelTrainingLarge.sh 1000
$ ./shell/RunHighLevelTrainingLarge.sh 2000
$ ./shell/RunHighLevelTrainingLarge.sh 3000
```

Output: the training log files including the RPS, placement matrix, training time duration, etc. will be store in the folder:

```
$ ls ./checkpoint/
729nodes_*_*/
```

4.3 Schedule Containers with Vanilla RL

Vanilla RL is built directly upon Policy Gradient without our Hierarchical designs.

1. High-level training in a medium-sized cluster of 81 nodes:

```
$ ./shell/RunVanillaRLMedium.sh 200
```

Output: the training log files including the RPS, placement matrix, training time duration, etc. will be store in the folder:

```
$ ls ./checkpoint/
Vanilla_81_*_*/
```

4.4 Schedule containers Divide-Conquer (DC) only

DC Method does not use sub-schedulers. Our code below shows its behaviors in a medium-sized cluster of 81 nodes. Each cluster is hierarchically divided into three subsets.

1. High-level training in a medium-sized cluster of 81 nodes:

```
$ ./shell/RunDCMedium.sh 200
```

Output: the training log files including the RPS, placement matrix, training time duration, etc. will be store in the folder:

```
$ ls ./checkpoint/
DC_81_*_*/
```

4.5 Schedule Containers with Medea

Medea is implemented using Matlab, due to its outstanding performance in solving the Integer Linear Programming (ILP) problem.

1. Generate the performance-constraints used in Medea:

```
$ cd Experiments
$ ./shell/GenerateInterference.sh
$ ls
interference_applist.csv
interference_rpslist.csv
```

2. Run Medea in the folder:

```
$ cd testbed/Medea
$ Matlab Medea.m
```

Output: the scheduling decision log files including the allocation matrix, constraint violations, time duration .etc will be store in the folder.

4.6 Schedule Containers with Paragon

Paragon is re-implemented in Python. For the sake of fair comparison, we feed it with the full interference matrix information as Medea.

1. Make sure `interference_applist.csv` has been generated in former Medea setup:

```
$ ls interference_applist.csv
```

Otherwise, generate the performance-constraints used in Medea:

```
$ cd Experiments
$ ./shell/GenerateInterference.sh
$ ls interference_applist.csv
```

2. Run Paragon of Medium size or Large size:

```
$ cd Experiments/shell
$ ./RunParagonMedium.sh 200 # Medium size
$ ./RunParagonLarge.sh 2000 # Large size
```

Output: the default output shows the average throughput for each testing group as well as the scheduling latency.

For detailed output including container placement and per-container throughput breakdown for each node, please add -v after each python script:

```
$ cd Experiments
$ python3 ParagonExp.py --batch_set_size
200 --batch_choice 0 --size medium --verbose
```

ARTIFACT AVAILABILITY

Software Artifact Availability: All author-created software artifacts are maintained in a public repository under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: All author-created data artifacts are maintained in a public repository under an OSI-approved license.

Proprietary Artifacts: None of the associated artifacts, author-created or otherwise, are proprietary.

Author-Created or Modified Artifacts:

```
Persistent ID: https://github.com/Metis-RL-based-con⌋
↪  tainer-sche/Metis
Artifact name: Metis: Learning to Schedule
↪  Long-Running Applications in Shared Container
↪  Clusters with at Scale

Citation of artifact:
↪  http://dx.doi.org/10.5281/zenodo.3859335
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: 2nd Generation Intel® Xeon® Scalable Processors (Cascade Lake)

Operating systems and versions: Ubuntu 18.04 running Linux kernel 4.15.0

Compilers and versions: GCC 7.4.0

Applications and versions: Docker v19.03

Libraries and versions: TensorFlow v1.12.0

Key algorithms: policy gradient