UniSched: A Unified Scheduler for Deep Learning Training Jobs With Different User Demands

Wei Gao, Zhisheng Ye, Peng Sun, Tianwei Zhang, *Member, IEEE*, and Yonggang Wen, *Fellow, IEEE*

***Abstract*—The growth of deep learning training (DLT) jobs in modern GPU clusters calls for efficient deep learning (DL) scheduler designs. Due to the extensive applications of DL technology, developers may have different demands for their DLT jobs. It is important for a GPU cluster to support all these demands and efficiently execute those DLT jobs. Unfortunately, existing DL schedulers mainly focus on part of those demands, and cannot provide comprehensive scheduling services. In this work, we present UniSched, a unified scheduler to optimize different types of scheduling objectives (e.g., guaranteeing the deadlines of SLO jobs, minimizing the latency of best-effort jobs). Meanwhile, UniSched supports different job stopping criteria (e.g., iteration-based, performance-based). UniSched includes two key components: Estimator for estimating the job duration, and Selector for selecting jobs and allocating resources. We perform large-scale simulations over the job traces from the production clusters. Compared to state-of-the-art schedulers, UniSched can significantly decrease the deadline miss rate of SLO jobs by up to 6.84×, and the latency of best-effort jobs by up to 4.02×, To demonstrate the practicality of UniSched, we implement and deploy a prototype on Kubernetes in a physical cluster consisting of 64 GPUs.**

***Index Terms*—Distributed systems, deep learning, GPU cluster scheduling.**

I. Introduction

The tremendous progress of deep learning (DL) technology makes DL training (DLT) an indispensable workload in research institutes and commercial cloud providers. Training a production-level DL model usually demands huge efforts in terms of time and GPU resources. Consequently, these companies and institutes typically establish large-scale GPU clusters to satisfy intensive demands of DLT jobs from different users. A scheduler is required to manage the execution of DLT jobs and allocate resources to them.

Manuscript received 1 March 2023; revised 22 January 2024; accepted 28 January 2024. Date of publication 29 February 2024; date of current version 10 May 2024. This work was supported in part by the National Key R&D Program of China under Grant 2022ZD0160201, and in part by the RIE2020 Industry Alignment Fund - Industry Collaboration Projects (IAF-ICP) Funding Initiative, as well as cash and in-kind contribution from the industry partner(s). Recommended for acceptance by B. Childers. *(Corresponding author: Tianwei Zhang.)*

Wei Gao is with the School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798, and also with the S-Lab, Nanyang Technological University, Singapore 639798 (e-mail: gaow0007@ntu.edu.sg).

Tianwei Zhang and Yonggang Wen are with the School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798 (e-mail: tianwei.zhang@ntu.edu.sg; ygwen@ntu.edu.sg).

Zhisheng Ye is with Peking University, Beijing 100871, China (e-mail: yezhisheng@pku.edu.cn).

Peng Sun with the Shanghai AI Lab & SenseTime, Beijing 100080, China (e-mail: sunpeng1@sensetime.com).

Digital Object Identifier 10.1109/TC.2024.3371794

TABLE I
Categorization of DLT Jobs in Modern GPU Clusters, and Their Corresponding Scheduling Solutions

Latency Demands \ Stopping Criteria	Iteration-Based	Performance-Based
Service Level Objective	[14], [17]	[16], [25]
Best-Effort	[1], [2], [3], [4], [5], [6] [7], [8], [9], [10]	[19], [26], [27]

As DL models are practically used in different scenarios for different purposes, users can have distinct demands for the scheduling and execution of their DLT jobs in the GPU cluster. These demands can be categorized from two perspectives, as summarized in Table I. First, users may have different expectations for the *scheduling latency*. In particular, some users hope their jobs to be completed within specified deadlines. These jobs are mainly for production development, DL competitions and challenges, and research paper submissions. These jobs are referred to as *Service Level Objective (SLO) jobs*. In contrast, other jobs are expected to be completed as soon as possible without specific deadlines. We call them *best-effort jobs*. Second, as DLT is an iterative process, users may have different *stopping criteria* to complete the training job. For instance, some users may specify the number of training iterations for their jobs. Other users prefer to stop the training jobs when the models meet the desired performance indicated by some metrics (e.g., accuracy, mAP, loss).

A shared GPU cluster can contain a mixture of the above jobs with different demands. Then the question we aim to answer in this paper is: *how can we efficiently schedule those jobs and satisfy both their scheduling latency requirement and stopping criteria?* Unfortunately, existing works mainly focus on certain specific demands, and cannot cover all the types simultaneously. Particularly, the majority of DL schedulers aim to reduce the scheduling latency [1], [2], [3], [4], [5], [6] or maintain job fairness [7], [8], [9], [10] for best-effort jobs. Thus they miss the opportunities of guaranteeing the deadlines of SLO jobs.

To satisfy the deadline requirement of SLO jobs, prior studies propose deadline-aware schedulers for traditional big data jobs [11], [12], [13], which could be extended to schedule DLT workloads. However, these solutions do not consider the unique features of DLT jobs, and cannot achieve optimal efficiency.

Recently researchers propose deadline-guaranteed scheduling systems tailored for DLT jobs. GENIE [14] automatically identifies the optimal resource allocation for SLO jobs. However, it requires modifications of the underlying DL frameworks (e.g., tensorflow [15]), and ignores the resource requirements from users. HyperSched [16] aims to improve the performance of Hyper-Parameter Optimization jobs, and cannot be directly adapted to generic DLT jobs. Our recent work, CHRONUS [17], can satisfy generic SLO and best-effort jobs simultaneously. However, it only considers the iteration-based stopping criterion, while ignoring the performance-based criterion. The main objective of Hydra [18] is to meet the deadline while reduce the latency for SLO jobs in a heterogeneous GPU cluster. However, it does not consider best-effort jobs, and fails to support the performance-based criterion.

This paper presents UNISCHED, a novel scheduling system that can satisfy all the scheduling latency demands and stopping criteria in a unified way. UNISCHED is improved over CHRONUS [17]. For a mixture of different types of jobs in a shared GPU cluster, UNISCHED is able to guarantee the SLO jobs' deadlines, minimize best-effort jobs' latency, and support both iteration-based and performance-based stopping criteria. To achieve these goals, UNISCHED needs to address three key challenges. First, *the lack of job runtime information misleads the job selection and resource allocations of UNISCHED.* Chronus is built upon the high intra-job predictability of DLT jobs. Hence, it is feasible to mathematically model the execution speed of distributed training jobs with any resource allocations [6], [14], [19]. However, runtime estimator proposed in CHRONUS performs not satisfactorily for two reasons. (1) The preemption overhead of DLT workloads will prolong the job execution time. For example, the preemption overhead of GPT-3 [20] can be up to several minutes. The unawareness of the preemption overhead will increase the job runtime prediction error. The inaccurate runtime estimation further misleads the scheduler to make ineffective decisions. (2) CHRONUS cannot handle the performance-based criteria jobs due to the lack of the number of training iterations. To address this challenge, we propose `Estimator` to improve the job runtime accuracy. Specifically, we devise the *sr-aware estimator* to incorporate the preemption overhead into the job runtime prediction. The core idea is to use the statistical expected value of the preemption overhead. We also design the *training iteration estimator* for performance-based criteria jobs to estimate the number of training iterations needed to reach the targeted performance. It uses a technique from [21] to characterize the relationship between the number of training iterations and performance metric in an online manner, and then approximates the job duration.

Second, *the mixture of profiler jobs, best-effort jobs and SLO jobs complicates the job scheduling.* To date, Chronus is the only DL scheduler that accounts for a mixture of best-effort and SLO jobs. Profiler jobs are necessary for online profiling of job runtime, as adopted in some works [6], [7], [19]. CHRONUS employs resource reservation, shortest remaining time first, and mixed integer linear programming (MILP) to handle these three job types separately. However, these ad-hoc techniques increase the scheduling complexity and miss joint optimization opportunities. We propose to redesign the reward functions for different job types, where the difference between jobs is represented by the reward value over time. This transforms the scheduling of all job types into an MILP optimization problem, alleviating the error-prone ad-hoc design and simplifying the implementation.

Third, *the execution speed of a distributed training job can be affected by the GPU allocation topology.* In other words, training jobs are placement-sensitive, and can achieve faster speed on consolidated GPUs due to reduced local communication costs. However, previous deadline-aware schedulers [11], [12], [13], [22] only take into account the amount of available resources, rather than their topology. While CHRONUS considers the placement sensitivity of SLO jobs and enforces the strict consolidation placement through the round-up technique, it sacrifices the placement efficiency of best-effort jobs. Existing DL schedulers for deadline guarantee [16], [18] also do not provide efficient placement strategies for best-effort jobs. UNISCHED relaxes the strict consolidation placement constraint for SLO jobs and introduces a novel approach for identifying appropriate resource allocations for both best-effort and SLO jobs. This methodology, inspired by Hived [4] allows for flexible resource allocations within the MILP optimization framework. Unlike CHRONUS, which executes job selection and resource allocation sequentially, UNISCHED optimizes both processes simultaneously through a unified solver.

To evaluate UNISCHED, we perform large-scale simulations on Helios [23] and Philly [2] traces from SenseTime and Microsoft respectively. Evaluation results demonstrate that UNISCHED can reduce up to 6.84× deadline miss rate. Compared with existing deadline-aware schedulers, UNISCHED reduces up to 4.02× latency of best-effort jobs. We further implement UNISCHED as a custom scheduler with the Kubernetes system [24], and deploy it on a physical cluster consisting of 64 GPUs. This cluster supports various common DL models for computer vision, natural language processing. Evaluations show that UNISCHED can effectively guarantee SLO jobs' deadlines and maintain best-effort jobs' execution latency. The contributions of this paper are:

- UNISCHED features the `Estimator` that can predict job duration for various stopping criteria, including iteration-based and performance-based ones.
- UNISCHED explicitly takes the overhead of suspension and resumption into account when estimating the duration of jobs.
- UNISCHED unifies job profiling, scheduling and resource allocation into one MILP framework, and makes efficient joint optimization to determine when and how to execute DLT jobs.

II. MOTIVATION

We discuss the categorization of DLT workloads in modern GPU clusters, the importance of performance-based stopping criteria jobs, and the advantages of joint optimization.

A. Categorization of DLT Workloads

We categorize DLT workloads from two perspectives. The first one is *scheduling latency*. According to the survey in

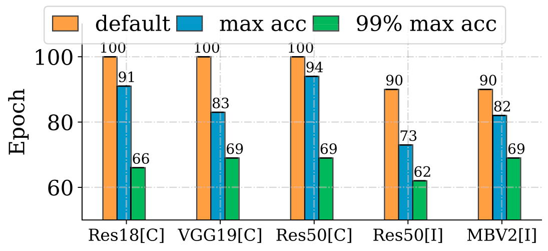


Fig. 1. Comparison of training epochs using three stopping criteria: default iteration-based stopping, stopping at maximum accuracy, and stopping at 99% of maximum accuracy over tasks. [C] and [I] indicates cifar10 [30] and imagenet [31] respectively.

[17], there can be two types of scheduling objectives: (i) Users expect their jobs to be scheduled as soon as possible. These are exploratory jobs for debugging and testing purposes, so users hope to receive the execution feedback promptly and then adjust their programs or hyperparameters. These jobs are generally called "best-effort jobs". (ii) Users do not need their jobs to be scheduled immediately. Instead, they set specific deadlines, before which these jobs should be completed. Those jobs are mainly involved in scenarios where certain deadlines are enforced, such as product development pipeline, research paper submission, AI challenges, competition, etc. These jobs are referred to as "SLO jobs". In addition, the survey in [17] discloses the existence of soft SLO jobs: users can tolerate the deadline violation of DLT jobs to certain extent, giving the scheduler more flexibility to schedule SLO jobs.

The second categorization perspective is *stopping criteria*. There are also two common strategies for users to determine the completion of a DLT job. (i) Iteration-based criterion. The users just specify fixed numbers of iterations. Then the cluster executes the DLT jobs for the required iterations. Note that the model after the final iteration may not be the optimal one due to the overfitting phenomenon. The system will make checkpoints at different iterations so the users can select the best model during training. (ii) Performance-based criterion. The users specify the expected performance for the resulting model. Then the training job will be early stopped if the model reaches the performance requirement at a certain iteration. Existing DL frameworks including ray [28] and optuna [29] provide an interface to terminate a job when the performance metric reaches a target value. RubberBand [25] and HyperSched [16] also account for early stopping to terminate a job when the performance metric converges. Note that the users are required to set a maximal number of training iterations to avoid unreachable performance requirements.

Comparison between different stopping criteria. The adoption of the iteration-based stopping criteria simplifies the job runtime prediction. But it should be noted that the ultimate objective of DL training is to attain high-performing DL models. While the iteration-based stopping criteria is widely used, the performance-stopping criteria may result in a reduction of training. As demonstrated in Fig. 1, using max accuracy for performance-stopping criteria can reduce the number of training iterations by up to 22% compared to the default training iteration. The epoch reduction can be up to 31% when the targeted accuracy is 99% of the max accuracy. CHRONUS can lead to a potentially significant consumption of GPU resources and delay in the execution of other jobs in the future, due to the adoption of the maximal training iteration to approximate job runtime.

B. *Advantages of Joint Optimization*

A key distinction between UNISCHED and CHRONUS lies in the joint optimization. UNISCHED implements joint optimization through two aspects.

First, joint job selection in UNISCHED benefits both profiler and best-effort jobs without affecting the attainment of SLO. This approach helps to avoid the situation where online profiling becomes a bottleneck for deadline guarantees. In contrast, CHRONUS reserves a fixed number of GPUs (up to 16) for profiling purposes. However, when the GPU cluster has a limited amount of resources to meet the deadline guarantees of SLO jobs, a surge in SLO job submissions can occur. The reserved GPU nodes may not be sufficient to handle the profiling of these bursty job submissions, leading to a large number of pending SLO jobs and potential SLO violations. Scaling profiling resources adaptively in an isolated manner might be another solution to address the bursty submission issue. This solution would impose an extra burden on system maintenance. Differently, our proposed unified approach is elegant to integrate scaling profiling resources adaptively without any extra engineering effort. Additionally, CHRONUS does not distinguish between the importance of best-effort jobs, which is not realistic in a production environment.

Second, the joint optimization approach in UNISCHED can improve the latency efficiency of best-effort jobs while still meeting the deadline requirements of SLO jobs. As an example, consider a scenario where four 6-GPU SLO jobs compete for access to three 8-GPU nodes. The round-up technique used in CHRONUS can only allocate GPU resources to three of the SLO jobs due to its strict consolidated placement constraint. However, UNISCHED leverages `Estimator` to predict the job runtime under different resource allocations, enabling it to satisfy the deadline requirements of all four SLO jobs. Similarly, in a scenario where there are three 6-GPU SLO jobs and one 6-GPU best-effort job, CHRONUS cannot allocate resources to all the jobs. In contrast, UNISCHED can relax the consolidated placement constraint for one of the SLO jobs without violating its corresponding deadline, and allocate consolidated resources to the best-effort job to reduce the corresponding latency (i.e., maximizing its reward value).

III. SYSTEM DESIGN

UNISCHED is a new scheduler to achieve various scheduling goals. UNISCHED is improved over CHRONUS [17], and addresses its following limitations: (1) CHRONUS can satisfy the mixture of both SLO and best-effort jobs, but only accept the iteration-based stopping criterion. UNISCHED can handle jobs with the stopping criterion of different performance metrics. (2) CHRONUS performs job profiling, selection, and resource allocation separately in an ad-hoc way. In contrast, UNISCHED introduces a unified MILP framework, which jointly optimizes all the stages with better efficiency. We begin by introducing our system assumptions and providing an overview in Section III-A.

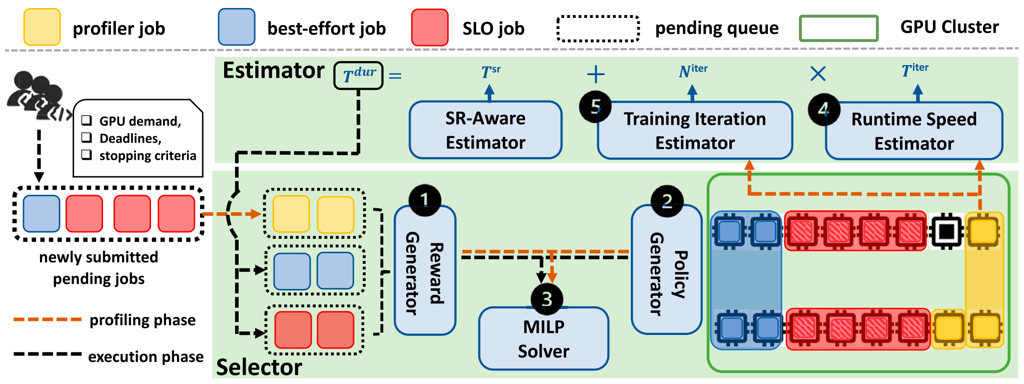


Fig. 2. UNISCHED consists of two components to manage DLT jobs: `Estimator` for predicting the job duration and `Selector` for job selection and resource allocation. Each job experiences two phases: *profiling* phase (orange dashed line) for collecting job information to estimate the job duration and *execution* phase (black dashed line) for job execution.

Subsequently, we delve into the details of each component in Section III-B–III-C.

A. System Assumptions and Overview

UNISCHED makes certain assumptions regarding DLT jobs and GPU clusters. (1) The memory of a single GPU can support at least single-sample training. (2) For simplicity, UNISCHED assumes that the DLT jobs perform a data-parallel distributed training approach and adopt the all-reduce to synchronize the gradients. (3) Our consideration involves a shared cluster that has *homogeneous* GPU resources and physical network connections. Our system can also be generalized to heterogeneous GPU clusters (Section VIII).

Fig. 2 shows the workflow of UNISCHED. It consists of two main components: `Estimator` for predicting the job duration and `Selector` for selecting jobs and allocating resources to them for execution. Each job experiences two phases in its lifecycle. The first phase is *profiling* (orange dashed lines in Fig. 2). All the newly submitted jobs are treated as profiler jobs. (1) In the `Selector`, the jobs are placed in the profiler job queue. The *reward generator* is called to assign a reward to each job (❶). The *policy generator* then generates all possible resource allocation solutions for each job (❷). Finally, a MILP solver is utilized to identify an effective solution (❸) so the selected job will be scheduled for profiling. (2) In the `Estimator`, the *runtime speed estimator* predicts the runtime speed of each profiler job over different resource allocations (❹). The *training iteration estimator* predicts the number of training iterations for jobs with performance-based criterion (❺). Based on such information, the estimated job duration is produced.

The second phase is *execution* (black dashed lines in Fig. 2). The estimated duration is forwarded to the `Selector`. The job is then placed in either the SLO job queue or best-effort job queue, depending on its scheduling latency requirement specified by its user. The following procedure is similar to the *profiling* phase: the `Selector` generates the reward and allocation policy for the job and adopts the MILP solver to identify the optimal scheduling solution. The MILP solver also requires the estimated job duration from the *profiling* phase for the solution generation. Then the selected job will be placed on the assigned GPUs for execution.

UNISCHED unifies the scheduling workflow in two aspects. First, in the *profiling* phase, UNISCHED processes the best-effort and SLO jobs in a unified way. All the jobs are referred to as profiler jobs. They are only distinguished in the *execution* phase. Second, the `Selector` processes the *profiling* and *execution* phases in a unified way, i.e., they adopt the same methodology to generate the reward and allocation policy regardless of the phases. These unified strategies make it easy to manage, implement and maintain the entire system workflow.

Before elaborating our approach, we summarize the relevant symbols used in this paper in Table II if not particularly specified.

B. Estimator

Formally, we consider a set of N jobs: $\mathcal{J} = \{j_0, j_1, \ldots, j_{N-1}\}$. Assume the vector of job duration for $\mathcal{J}$ is $\mathbf{T}^{\text{dur}}$, the vector of training iteration for $\mathcal{J}$ is $\mathbf{N}^{\text{iter}}$, the vector of time cost of suspension and resumption for $\mathcal{J}$ is $\mathbf{N}^{\text{sr}}$, the vector of time cost per iteration for $\mathcal{J}$ is $\mathbf{T}^{\text{iter}}$.

The `Estimator` is responsible for predicting the duration T_i^{dur} of j_i. This is calculated as follows:

$$T_i^{\text{dur}} = T_i^{\text{sr}} + N_i^{\text{iter}} \cdot T_i^{\text{iter}}. \tag{1}$$

Note that the number of training iterations N_i^{iter} is directly specified by users for iteration-based criterion, or indirectly predicted for performance-based criterion. We estimate $\mathbf{T}^{\text{iter}}$, $\mathbf{N}^{\text{iter}}$ and $\mathbf{T}^{\text{sr}}$ by the *runtime speed estimator*, *training iteration estimator*, and *SR-aware estimator*, respectively. UNISCHED only needs to allocate at most 2 GPUs for each job during profiling stage, regardless of its actual resource demands. We will discuss how to schedule these jobs during profiling stage in Section III-C.

1) Runtime Speed Estimator: DLT jobs exhibit an iterative and repetitive pattern during training. This motivates UNISCHED to use a simple yet effective way to estimate $\mathbf{T}^{\text{iter}}$. The `Estimator` executes profiler jobs on actual machines for a fixed time, which is empirically set as five minutes.

TABLE II
Summary of Notations

Sym.	Definition
$\lceil\cdot\rceil$	ceiling
$\lfloor\cdot\rfloor$	floor
$\mathbf{T}^{\text{dur}}$	vector of job duration
$\mathbf{T}^{\text{sr}}$	vector of time cost of suspension and resumption
$\mathbf{T}^{\text{iter}}$	vector of time cost per iteration
$\mathbf{T}^{\text{comp}}$	vector of computation time cost per iteration
$\mathbf{T}^{\text{lease}}$	vector of lease length
$\mathbf{N}^{\text{iter}}$	vector of training iterations
$\mathbf{N}^{\text{gpu}}$	vector of GPU request
$\mathbf{N}^{\text{node}}$	vector of GPU node request
$\mathbf{N}^{\text{cell}}$	vector of cell count
$\mathbf{N}^{\text{con}}$	vector of cell request
$\mathcal{J}$	job set
$\mathcal{J}^{\text{slo}}$	SLO job set
N	job count
M	total GPU count in the cluster
j_i	i_{th} job in $\mathcal{J}$
j_i^{slo}	i_{th} job in $\mathcal{J}^{\text{slo}}$
F_i	deadline count of j_i
$F_{\max}$	maximal deadline count across all jobs
$D_{f,i}$	f_{th} deadlines of j_i
$V_{f,i}$	reward value for deadline $D_{f,i}$
$Q_{f,i}$	lease term count of deadline $D_{f,i}$
L_i	lease term count to complete j_i
P_i	resource allocation count of job j_i
$\mathcal{A}_i$	resource allocation set of job j_i
$A_{i,p}$	p_{th} allocation policy of job j_i
A_i^*	optimal allocation policy for job j_i
$\mathbf{S}$	binary matrix to indicate which deadline each job hits
$x_{k,i}$	indicator of whether j_i obtains the k_{th} lease
$y_{k,i}$	indicator of whether to select policy $A_{i,p}$
R^{slo}	weighted deadline miss rate

Let $\mathbf{N}^{\text{gpu}}$ and $\mathbf{N}^{\text{node}}$ be the vector of GPU request and GPU node request for $\mathcal{J}$ respectively. We consider two scenarios for j_i.

First, this is a single-GPU job ($N_i^{\text{gpu}} = 1$). Then UniSched allocates one GPU in profiling, and measures its computation time T_i^{comp} as the time cost per iteration, i.e., $T_i^{\text{iter}} = T_i^{\text{comp}}$.

Second, this is a multi-GPU job ($N_i^{\text{gpu}} \geq 2$). Then we should consider both computation time and communication time. There are also two possibilities: (i) this job will be executed on one machine in the execution phase. Then we allocate two GPUs on the same machine to this profiler job ($N_i^{\text{node}} = 1$), and measure the gradient communication time T_i^1; (ii) this job will be distributed to multiple machines in the execution phase ($N_i^{\text{node}} \geq 2$). Then we allocate two GPUs from two machines to this profiler job and measure the corresponding gradient communication time T_i^2. To summarize, the time cost per iteration for j_i can be modeled as:

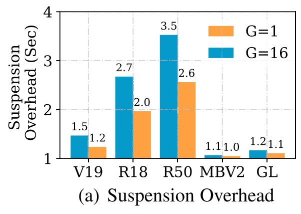
$$T_i^{\text{iter}} = \begin{cases} T_i^{\text{comp}} & \text{if } N_i^{\text{gpu}} = 1, \\ T_i^{\text{comp}} + (N_i^{\text{gpu}} - 1) \cdot T_i^1 & \text{if } N_i^{\text{node}} = 1,\ N_i^{\text{gpu}} \geq 2, \\ T_i^{\text{comp}} + (N_i^{\text{gpu}} - 1) \cdot T_i^2 & \text{otherwise.} \end{cases} \tag{2}$$

Previous works [6], [32] also adopt similar performance modeling with Eq. 2 to estimate the job runtime speed. The key idea is that we can just use two GPUs to capture the intra-node and inter-node communication overheads (T_i^1 and T_i^2), then the total timing cost for a job with an arbitrary number of GPUs can be derived accordingly. Another point is that our system testbed only focuses on utilizing PCIe and RDMA for communication. There are cluster designs adopting the underlying GPU topology of non-unified communication cost [33] including PCIe, NVLink, and GPUDirect. We leave the modeling of non-unified communication cost as our future work. These profiling results are reported to the MILP solver to determine the placement policy for each job.

Discussion. We further demonstrates the effectiveness of Eq. 2 and how Eq. 2 handles some exceptional cases. (1) Our Eq. 2 is a simplified version of runtime speed estimator in [6], where we intentionally disregard the overlap between gradient computation and network communication overhead. If they are overlapped, Eq. 2 may result in an overestimation of T_i^{iter}, which can secure the deadline guarantees for SLO jobs and improve SLO guarantee objective. (2) We only consider the data-parallel distributed training with all-reduce to synchronize the gradients. How to extend our solution to other parallelism mechanisms (e.g., tensor parallel, pipeline parallel) and model the execution time is our future work. (3) Eq. 2 cannot model the PCIe bandwidth saturation scenario, which is very rare in practice. In case it happens, we can update T_i^{iter} during the execution stage to account for PCIe bandwidth saturation. (4) Our empirical evaluations in Section VI-A prove the estimation error of Eq. 2 is acceptable.

2) Training Iteration Estimator: For the iteration-based stopping criterion, the user directly specifies N^{iter}. For the performance-based criterion, it is non-trivial to predict N^{iter} from the specified performance requirement. The performance metric is typically non-linear to the number of training iterations [34]. We adopt a method from [21] to predict the relationship between the performance metric and training progress. The basic idea is to model the observed performance metrics using an ensemble of probabilistic learning curve models, e.g., Weibull, log-power. These models can extrapolate future performance via only a few observed performance metrics. This method is robust to different performance metrics (accuracy, mAP, F1-score, loss) and optimization techniques (SGD, Adam). Several schedulers [26], [27] have adopted it to predict when the performance metric of the DLT job will satisfy the stopping criterion.

UniSched first uses the performance metric observed in the profiling phase to predict the required number of training iterations. However, just using such metric in this phase can result in significant prediction errors, as demonstrated in Fig. 7(c). The prediction error comes from two aspects: (1) we change the batch size in the profiling phase to collect the accurate job computation time per iteration, and (2) the number of collected metrics is limited during the profiling phase. We notice that even we use the training hyper-parameters and the number of required GPUs, the prediction error is still significant (shown Fig. 7(c) when x-axis is 20%). Hence, we also collect the performance metrics in the executing phase to gradually eliminate the prediction error.



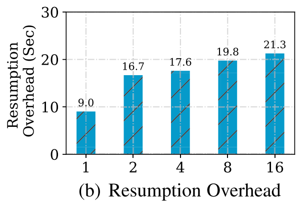


Fig. 3. The overhead of job suspension and resumption: (a) the job suspension overhead (y-axis) of training VGG19, ResNet18, ResNet50, MobileNetV2, GoogLeNet over cifar10 on 1 GPU and 16 GPUs; (b) the job resumption overhead (y-axis) of training VGG19 over cifar10 on different numbers of GPUs (x-axis).

3) SR-Aware Estimator: UNISCHED allows a DLT job j_i to be suspended and resumed during the training progress, which increases the scheduling flexibility but inevitably brings a certain overhead of suspension and resumption operations, denoted as t_i^{sr}. Fig. 3 shows the overhead of job suspension and resumption. In Fig. 3(a), the suspension overhead of various models over cifar10 with different numbers of GPUs remains consistently within a range of 4 seconds. Fig. 3(b) illustrates that scaling the number of allocated GPUs increases the resumption overhead of training VGG19 over cifar10. Overall, the resumption overhead is much larger than the suspension overhead. Note that t_i^{sr} represents the combined overhead of job suspension and resumption, rather than that of job resumption or suspension. Practically, we use such combined overhead during the profiling phase and update it in the execution phase. According to Fig. 3, the difference in the combined overhead during the profiling (2-GPU) and execution (16-GPU) phases is within 5 seconds for training VGG19 over cifar10. This suggests that directly using the combined overhead during the profiling phase is acceptable compared to long training time.

For an SLO job j_i, we assume it runs for n lease terms, and its corresponding deadline is m lease terms ($n \leq m$). A lease term is the smallest unit for a job to run continuously, which will be explained in detail in Section III-C1. The overhead of suspension and resumption operations for an SLO job j_i is up to t_i^{sr}.

We assume the occurrence of suspending and resuming a DLT job follows a uniform probability distribution. Hence the probability that an SLO job is suspended and resumes for k times is $\frac{C_{n-1}^k C_{m-n+1}^{k+1}}{C_m^n}$, where $k \in [0, \min(n-1, m-n)]$. Therefore, we can approximate the overhead of job suspension and resumption T^{sr} as follows:

$$T_i^{\text{sr}} = \sum_{k=0}^{\min(n-1,m-n)} k \cdot t_i^{\text{sr}} \cdot \frac{C_{n-1}^k C_{m-n+1}^{k+1}}{C_m^n}. \quad (3)$$

For a best-effort job that requires n lease terms, the probability that suspension and resumption occurs is $\frac{1}{2}$. Hence, its corresponding T^{sr} is $\frac{n}{2} \cdot t_i^{\text{sr}}$. To summarize, Estimator offers three unique contributions. (1) It predicts the speed of DLT jobs across various resource allocation topologies with at most 2 GPUs. (2) It approximates the number of training iterations required to achieve a target validation metric. This estimation is particularly valuable for jobs with performance-based stopping criteria. (3) It considers the significant overhead of suspension and resumption in job execution. By accounting for these factors, our estimator effectively minimizes the gap between the predicted duration of a job and its actual execution time.

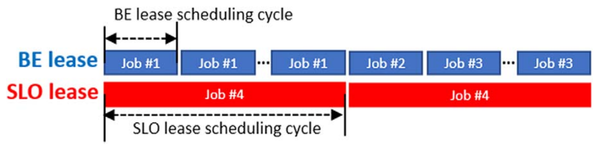


Fig. 4. Illustration of lease terms. The duration of the SLO lease term is set as an integral multiple of that of the BE lease.

C. Selector

The Selector is primarily responsible for producing resource-time scheduling decisions for profiler jobs in the profiling phase, and SLO jobs and best-effort jobs in the *execution* phase. It adopts the lease-based training scheme to convert job scheduling into the MILP optimization problem, and designs reward generator to successfully manage all three types of jobs. It also uses the policy generator to select the job and resource allocation jointly.

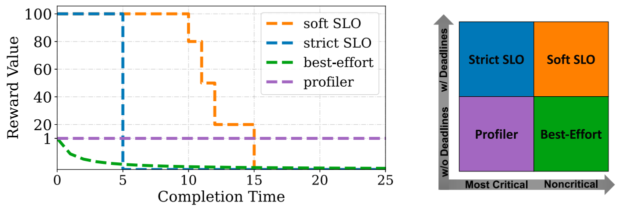
1) Lease-Based Training: A DLT job is split into multiple periods (i.e., lease terms) which have the equal length. A job is allowed to run only if the scheduler assigns a lease term to it. It needs to renew the lease when it expires. The job can continue the execution if the renewal is successful, and suspended and yield the resources otherwise.

UNISCHED implements two sorts of leases: SLO lease for SLO jobs, and BE lease for best-effort and profiler jobs. During each scheduling cycle, the expired leases are allocated to the chosen jobs by the Selector. To make it easy to manage, the length of an SLO lease is set as an integral multiple of that of a BE lease. In this setting, expiration of a BE lease may not cause the expiration of an SLO lease, while expiration of an SLO lease occurs simultaneously with the expiration of a BE lease. Fig. 4 shows an example of the two leases.

2) Reward Generator: Previous deadline-aware schedulers [11], [12], [13] only take into account the strict deadline requirement, i.e., a job must be finished before the specific time. Based on a user survey in [17], users expect to have the soft deadline requirement, where the DLT jobs can be completed after the deadlines with some penalty.

To enable this demand, a reward function is introduced in UNISCHED to formulate various types of requirements (profiler, best-effort, strict SLO, and soft SLO). Cluster users can also give such functions to the scheduler during job submission. The reward is defined as a step function with the values ranging between 0 and 100. Fig. 5(a) illustrates the functions of different requirements.

A profiler job expects a short waiting time to achieve the runtime speed information as soon as possible, and thus is regarded as a best-effort job with a fixed remaining time (e.g., 5 minutes). Therefore we set the reward of all profiler jobs as a fixed reward



(a) Reward (b) Categorization

Fig. 5. The illustration of different types of jobs: (a) the relationship between completion time and reward value for different types of jobs; (b) A two-by-two matrix to categorize these types.

value 1. Such reward design handles well the starvation of jobs while maintaining the deadline guarantee for SLO jobs. We have two scenarios to consider: (1) if the cluster-wide GPU resources available are only sufficient to meet the deadlines of SLO jobs, newly submitted jobs may experience resource starvation until certain jobs are completed, otherwise users have the option to assign an exceptionally high reward value, thereby increasing the likelihood of their job being executed quickly; (2) if there exist some extra cluster-wide GPU resources in addition to the ones used for meeting deadlines of SLO jobs, the reward generator of UNISCHED gives priority to newly submitted jobs (profiler jobs) over best-effort jobs. This prioritization strategy effectively prevents job starvation. Best-effort jobs are expected to be completed as soon as possible. Their reward values are as a reciprocal of the corresponding estimated remaining time. Strict SLO jobs need to be finished before the deadlines ($=100$). Their reward decreases gradually and gives longer delays in completion time[1].

To ensure that newly submitted jobs and best-effort jobs do not impact the deadlines of SLO jobs, we assign a significantly lower reward value to profiler and best-effort jobs compared to SLO jobs (using a ratio of 1 out of 100). Additionally, to expedite the completion of profiler jobs, we set their reward value higher than any best-effort jobs. For best-effort jobs, the reward value is reciprocally proportional to the remaining time, prioritizing jobs with the shortest remaining time. In fact, how to determine the reward of any job depends upon the practical needs. Setting extremely high values for SLO jobs would discourage users from submitting best-effort jobs. Setting small values for SLO jobs would encourage UNISCHED to satisfy more best-effort jobs to maximize the reward values and violate the deadlines for SLO jobs. We follow the prior work [13] and account for our user survey to determine the reward value. There is no complete answer to the selection of reward values. We leave it as our future work.

Our reward function enables the `Selector` to manage all types of DLT jobs in a unified way, as shown in Fig 5(b). The best-effort jobs can be counted as the noncritical profiler job. Similar to the profiler job, the strict SLO job has a constant reward value besides exceeding the deadline. The Soft SLO job can be considered as the noncritical strict SLO job.

[1]Users may have other expressions of reward functions for their soft SLO jobs. Note that any functions can always be approximated as the step function in UNISCHED.

3) Policy Generator: The policy generator produces all possible resource allocation solutions for each job. Following the buddy cell idea in HiveD [4], we denote 8-GPU, 4-GPU, 2-GPU, 1-GPU compute nodes as level-4, level-3, level-2, level-1 cells respectively. Such resource abstraction enables us to allocate resources considering GPU affinity not just the number of GPUs.

We consider a job j_i with N_i^{gpu} GPUs to explain how to leverage this resource abstraction to generate resource allocation policies. We use a quadruple (c_0, c_1, c_2, c_3) to denote any allowable resource allocation for this job, which represents the requested numbers of level-0, level-1, level-2, and level-3 cells, respectively. For example, for a job requesting 6 GPUs, possible resource allocations are $(0,0,0,1)$, $(0,1,1,0)$, and $(6,0,0,0)$. For N_i^{gpu} GPUs, the policy generator outputs the allocation policies by enumerating all quadruples. To reduce the optimization complexity of making allocation decisions, the policy generator is only applicable to the job with $N_i^{\text{gpu}} \leq 16$.

4) Joint Optimization of Job Selection and Allocation: With the assistance of the reward generator and policy generator, we can model the process of job selection and resource allocation as a MILP problem. At each BE scheduling cycle, the `Selector` is responsible for combining all the jobs (including SLO jobs at the SLO scheduling cycle) and making decisions for job status update and GPU resource assignment.

Consideration of rewards. We consider at one scheduling cycle there are N jobs: $\mathcal{J} = \{j_0, j_1, j_2, \ldots, j_{N-1}\}$ and M available GPUs. Each job j_i requires N_i^{gpu} GPUs, with the duration T_i^{dur} estimated by the `Estimator`. We denote the deadline count of job j_i as F_i. When the job j_i is completed right before the corresponding f_{th} deadline, it can obtain the reward value $V_{f,i}$. Further, we use $F_{\max}$ to represent the max number of deadlines across all jobs. Assume the vector of lease length for $\mathcal{J}$ is $\mathbf{T}^{\text{lease}}$. For job j_i, we set T_i^{lease} as BE lease length for best-effort and profiler jobs, and SLO lease for SLO jobs respectively. For each job j_i, it requires $L_i = \lceil T_i^{\text{dur}} / T_i^{\text{lease}} \rceil$ lease terms to complete. It also needs $Q_{f,i} = \lfloor D_{f,i} / T_i^{\text{lease}} \rfloor$ lease terms to complete before each deadline, where $f \in [F_i]$[2].

We denote a binary matrix $\mathbf{S} \in \mathbb{B}^{F_{\max} \times N}$, where $s_{f,i}$ denotes whether j_i hits the corresponding f_{th} deadline. A binary variable $x_{k,i}$ is used to represent whether j_i gets the k_{th} lease. The MILP solver is expected to produce a solution for the following problem:

$$\max_{\mathbf{S}} \sum_{i \in [N]} \sum_{f \in [F_i]} s_{f,i} V_{f,i}, \tag{4}$$

subject to:

$$x_{k,i}, s_{f,i} \in \{0,1\}, \forall i \in [N], f \in [F_i], \tag{5}$$

$$\sum_{f \in [F_i]} s_{f,i} \leq 1, \forall i \in [N], \tag{6}$$

$$\sum_{k \in [Q_{f,i}]} s_{f,i} x_{k,i} \leq s_{f,i} L_i, \forall i \in [N], f \in [F_i]. \tag{7}$$

[2]We define $[N] = \{0, 1, \ldots, N-1\}$ in this paper, where N can be different positive integers.

Objective (4) aims to maximize the total reward values of all jobs in the cluster. Constraint (5) restricts $x_{k,i}$ and $s_{f,i}$ as binary values. Constraint (6) ensures each job gets at most one feasible solution to meet the (soft) deadline. Constraint (7) guarantees all jobs need to be finished before the (soft) deadlines.

Consideration of resource allocations. Next, we discuss how to formulate resource allocation constraints. For a job j_i, UNISCHED adopts the policy generator to produce the resource allocation set $\mathcal{A}_i$, which contains P_i allowable resource allocation solutions. We denote as A_i^* the optimal allocation that meets the consolidation requirement. We use $\phi(j_i, A_{i,p})$ to represent the runtime speed of j_i under an allocation $A_{i,p} \in \mathcal{A}_i$. We can leverage the `Estimator` to estimate $\phi(j_i, A_{i,p})$. Then we formulate the normalized runtime speed $\bar{\phi}$ to quantify the correlation between the job throughput and resource allocation as follows:

$$\bar{\phi}(j_i, A_{i,p}) = \frac{\phi(j_i, A_{i,p})}{\phi(j_i, A_i^*)}. \tag{8}$$

A higher $\bar{\phi}(j_i, A_{i,h})$ indicates job j_i runs faster under the allocation $A_{i,h}$.

We introduce a binary variable $y_{i,p}$ to represent whether we select the solution $A_{i,p}$ for j_i with resource allocation set $\mathcal{A}_i$. We denote the vector of total cell request as $\mathbf{N}^{con}$ and the vector of free cell count as $\mathbf{N}^{cell}$. The requested number of level-g cells for resource allocation $A_{i,p}$ is denoted as $A_{i,p}(g)$. Then we can add the following constraints into the optimization problem:

$$y_{i,p} \in \{0,1\}, \forall i \in [N], p \in [P_i], \tag{9}$$

$$\sum_{g=0}^{3} 2^g \cdot N_g^{\text{cell}} \leq M, \tag{10}$$

$$N_g^{\text{con}} = \sum_{i\in[N]} \sum_{p\in[P_i]} y_{i,p} A_{i,p}(g), \forall g \in \{0,1,2,3\}, \tag{11}$$

$$\sum_{k\in[Q_{f,i}]} y_{i,p} s_{f,i} x_{k,i} \bar{\phi}(j_i, A_{i,h}) \geq y_{i,p} s_{f,i} L_i, \forall i \in [N], f \in [F_i], p \in [P_i], \tag{12}$$

$$\sum_{p\in[Pi]} y_{i,p} \leq 1, \forall i \in [N]. \tag{13}$$

Constraint (9) enforces $y_{i,p}$ to be a binary value. Constraint (10) guarantees the number of occupied GPUs is no greater than the capacity of the entire cluster. Commonly, we set $N_0^{\text{cell}}, N_1^{\text{cell}}, N_2^{\text{cell}}, N_3^{\text{cell}}$ as $0, 0, 0, M/8$ respectively. Constraint (11) guarantees the feasibility of the resource allocation solution. Constraint (12) guarantees the number of requested leases can ensure the completion of the job under given resource allocations. Constraint (13) ensures each job is assigned with at most one feasible resource allocation solution.

Besides, we also need to ensure the identified solution achieves consolidation placement. In particular, we refer 1-GPU, 2-GPU, 4-GPU, and $8b$-GPU jobs ($b \in \mathbb{Z}^+$) as consolidation-friendly jobs, and other types of jobs are called consolidation-hostile jobs. We say a resource allocation solution enjoys the consolidation feature if each job j_i with N_i^{gpu} GPUs is deployed on $\lceil N_i^{\text{gpu}}/8 \rceil$ nodes. Then the following proposition is given:

Proposition 1: Assume the cluster has N_0^{cell} level-0, N_1^{cell} level-1, N_2^{cell} level-2, and N_3^{cell} level-3 free cells respectively. The pending queue only contains N_0^{con} 1-GPU, N_1^{con} 2-GPU, N_2^{con} 4-GPU, and N_3^{con} 8-GPU consolidation-friendly jobs[3]. There exists a solution that can achieve the consolidation placement when the following constraint (14) is satisfied:

$$\sum_{g=i}^{3} 2^{g-i} \cdot N_g^{\text{con}} \leq \sum_{g=i}^{3} 2^{g-i} \cdot N_g^{\text{cell}}, \forall i \in \{0,1,2,3\}. \tag{14}$$

Proof: It is easy to construct a solution to meet the requirement. We first allocate N_3^{con} level-3 free cells to 8-GPU jobs in a consolidation way such that the allocated nodes have no GPU fragmentation due to $N_3^{\text{con}} \leq N_3^{\text{cell}}$. Then we split the remaining $m'(= N_3^{\text{cell}} - N_3^{\text{con}})$ level-3 cells into $2m'$ level-2 cells, and we have $2m' + N_2^{\text{cell}}$ level-2 cells. According to Eq. 14, the number of level-3 free cells is no less than that of 4-GPU jobs. Recursively, 2-GPU and 1-GPU jobs can satisfy the consolidated placement. □

Solving the optimization. UNISCHED leverages the MILP solver to find a solution that can achieve the Objective (4) while satisfying the Constraints (5-7, 9-14). Based on the solution, UNISCHED identifies the jobs that need to be scheduled at this cycle ($x_{k,i}$), and the optimal resource allocations to host these selected jobs ($y_{i,p}$). The rest jobs are put in a pending queue and will be considered at the next scheduling cycle. In terms of profiling time requirement and BE lease scheduling flexibility, the length of a BE lease term is fixed as 5 minutes. The length of an SLO lease term is critical to the MILP solver efficiency. A short SLO lease causes too many preemption operations for SLO jobs, while a longer SLO lease makes the scheduling less elastic. We set it as 10 minutes empirically.

Note that it takes some time for the MILP solver to generate the optimization solution, which can have an impact on the job execution. In order to mitigate the impact of these delays, UNISCHED employs a caching mechanism for the optimization solution generated during the previous scheduling cycle. If the MILP solver cannot generate a new solution for the current cycle within certain time, UNISCHED assigns the cached solution to select jobs to minimize the search space and computational overhead, and subsequently re-invokes the MILP solver.

IV. IMPLEMENTATION AND EXPERIMENTS

In this section, we discuss the implementation of our simulator and Kubernetes [35] prototype. Then, we describe how to construct our testbed and introduce the metrics and baselines.

A. Implementation Details

We develop a trace-driven simulator with 11,978 lines of python code. It can simulate different scheduling mechanisms in GPU clusters. The implementation of UNISCHED in our simulator comprises of 1,113 lines of Python code. The MILP solver employed as the backend is Gurobi 9.1 [36].

[3]Without loss of generality, an $8b$-GPU job is counted as b 8-GPU jobs.

Our physical prototyping implementation is built on top of Kubernetes [35], which contains three key components: a client-side watcher, controller, and scheduler. (1) A client-side watcher is utilized to monitor the execution of DLT jobs and gather the validation metric and job runtime speed. When the watcher receives notifications from the controller that the lease will expire, it makes checkpoints for the model. The client-side watcher also reports the collected validation metric and runtime speed every 5 minutes. (2) The controller notifies the scheduler when the lease of a DLT job is nearing its expiration. It also communicates with the watcher to trigger job checkpoint. The implementation of job checkpoint is via signal handler function. It talks to the MILP solver to solve Eq. 4 and make decisions about job selection and resource allocations. The MILP solver is implemented with an open-source goop library [37]. (3) The scheduler is provided with scheduling information and events (e.g., estimated remaining time, lease renewal). It is also responsible for job management (e.g., preemption, termination, execution, and assigning resources).

B. Testbed

In this study, we evaluate the performance of two homogeneous GPU clusters, C120 and C96, each consisting of 120 and 96 nodes, respectively, with 8 GPUs per node. To assess the performance of these clusters, we employ two realistic DLT job traces: the Helios trace [23] from SenseTime and the Philly trace [2] from Microsoft. We use the job submission time, job duration, and number of GPUs required in the Helios and Philly trace to construct the workloads for evaluation. As the job traces do not provide deadline information, we generate deadlines for strict and soft SLO jobs using a method that ensures a fair representation of real-world conditions. Specifically, for strict SLO jobs, we randomly generate a deadline within a range of 1.1 to 2 times the job duration, while for soft SLO jobs, we set the first deadline, $D_{0,i}$, in the same way as strict SLO jobs. We then set additional soft SLO deadlines at 1.1, 1.2, and 1.5 times $D_{0,i}$, with corresponding reward values of 80, 50, and 20, respectively, as determined by a user survey [17].

Each job in our simulation trace contains submission time, duration, deadline information, the number of GPUs, user name, job type, model type, and stopping criteria. We consider two stopping criteria: iteration-based, performance-based, and the jobs adopting these criteria account for 80%, 20%, respectively. The Helios and Philly trace do not include explicit information about iteration or performance criteria. Instead, they provide attributes such as “duration” and “name”. For iteration-based jobs, we use the job duration and job runtime speed to deduce the corresponding training iteration. For performance-criterion jobs, we identify a set of performance-aware keywords, e.g., “detection”, “cifar10”, “imagenet”, “face”. Only for these specific jobs do we assign performance-based stopping criteria. For a job with the performance-based criterion, we randomly choose the best metric or 99% best metric throughout the training as the target value. Besides, we use the profiled runtime speed on different GPU allocations and the preemption overhead of a real job trace for evaluation. Note that we scale the job speed for performance-criterion jobs to enforce the duration of performance-criterion jobs to match that from the trace.

Besides, we adopted the same technique as CHRONUS to generate six workloads from Helios and Philly. These workloads included jobs with all strict SLOs (H_SLO and P_SLO); workloads that mixed strict SLOs with best-effort jobs (H_MIX1 and P_MIX1); and workloads that included strict SLOs, soft SLOs, and best-effort jobs (H_MIX2 and P_MIX2).

C. Metrics

Weighted Deadline Miss Rate. This is to assess the level of attainment with the SLO requirements. We consider a set J^{slo} of SLO jobs, where each job $j^{\text{slo}}i$ is assigned a reward value $\mathcal{W}(j_i^{\text{slo}})$ based on its SLO specification, as illustrated in Fig. 5(a). To quantify the effectiveness of meeting these SLO requirements, we introduce the concept of a weighted deadline miss rate R^{slo}, which is defined by Eq. 15. Specifically, we set the bounds of the reward values as $\mathcal{W}_{\text{min}} = 0$ and $\mathcal{W}_{\text{max}} = 100$.

$$R^{\text{slo}} = \frac{1}{|J^{\text{slo}}|} \sum_{j_i^{\text{slo}} \in J^{\text{slo}}} \frac{\mathcal{W}(j_i^{\text{slo}}) - \mathcal{W}_{\text{min}}}{\mathcal{W}_{\text{max}} - \mathcal{W}_{\text{min}}}. \tag{15}$$

Job Completion Time (JCT). This measures the latency efficiency of best-effort jobs to evaluate the scheduling performance. A smaller JCT indicates higher scheduling efficiency. This metric measures the duration between the job submission and job completion. Hence, the profiling overhead is also incorporated to compute R^{slo} and JCT.

D. Baselines

To fully demonstrate the benefits of UNISCHED, we select six mainstream schedulers for comparison, which are classified into two categories. Besides, we also make a detailed comparative analysis between CHRONUS and UNISCHED.

Deadline-aware scheduler: (1) 3Sigma [22] applies the MILP solver to schedule a mix of SLO and best-effort big data jobs. It favors that SLO jobs preempt best-effort jobs, which can remarkably restricts the MILP solver's search space. The scheduling cycle of 3Sigma is set as 60 seconds based on the job time scale in our traces. (2) GENIE [14] proposes an offline prediction model to estimate the processing rate and response latency for various DL jobs. It enables DLT jobs to be executed on different GPU resources in an elastic way and selects the best placement policy. It assigns the highest priority to SLO jobs with the smallest laxity but does not consider best-effort jobs. We give best-effort jobs the lowest priority. (3) Hydra [18] aims to reduce the average the job latency while reduce the deadline miss rate. We set the priority of SLO jobs higher than that of best-effort jobs. Also, we adopt shortest remaining time first to manage both type of jobs. We implement it to fit into a homogeneous GPU cluster. Note that, Hydra does not consider preemptive scheduling.

Deep Learning scheduler: (4) Optimus [19] leverages an online fitting model to predict the job training speed and dynamically allocates GPU resources for jobs to prioritize the job to minimize the job completion time. We adopt the same implementation in [6]. (5) Themis [7] introduces a new metric, the finish time fairness, to assess the scheduling fairness. We also use the model proposed in [21] to estimate the duration of jobs with the performance-based stopping criteria. We implement Themis based on the implementation in [38].

V. END-TO-END EVALUATION

We first compare the performance difference between physical and simulator results to validate the fidelity of our simulator (Section V-A). Then, we measure the performance of the entire system using our simulator, and compare it with various baselines (Section V-B).

A. Physical Evaluation

Cluster testbed. We set up a cluster consisting of 16 GPU nodes, and each node has 4 Tesla V100-32GB GPUs, 1 $\times$ 200 Gbs HDR InfiniBand, 64 CPU cores, and 256 GB memory, connected via PCIe 3.0 x16. Our prototype deploys upon Kubernetes 1.18.2 and adopts CephFS 14.2.8 to establish a ceph distributed storage cluster to store checkpoints and resume the job progress. When the job experiences lease expiration, it will receive the notification from the scheduler to save the training state into the distributed storage. We choose the H_MIX2 workload to compare the evaluation results between our simulator and Kubernetes prototype. The MIX2 workload contains a mixture of best-effort, strict SLO and soft SLO jobs, which is a realistic scenario. Furthermore, the proportion of distributed DL training is higher than Philly [23], and distributed DL training involves many complex placement decisions. To synthesize our evaluation workload, we randomly sample a number of jobs from the H_MIX2 workload, and assign random common DL models (ResNet18, ResNet50, MobileNetV2, VGG19, BERT) over different datasets (Cifar10, ImageNet, WikiText2) to them. We sample the job whose number of requested GPUs is below 16 and the duration of which ranges between 5 minutes and 180 minutes. We follow the Helios's job arrival pattern, and only sample jobs the submission time of which is before eight o'clock. We also vary the job submission density and compare the performance between Kubernetes implementation and simulation.

Evaluation results. Table III reports R^{slo} of SLO jobs and average JCT of DLT jobs from simulation as well as Kubernetes implementation. We consider configurations ($T[m]$) with different job densities with a fixed cluster capacity of 64 GPUs: $T[m]$ denotes m jobs are submitted within the first 8 hours. For R^{slo}, the gap between simulation and Kubernetes prototype is at most 2.57%. For average JCT, the maximal relative performance difference between simulation and Kubernetes is 5.38%. For small submission density, the deadline guarantee of the simulator performs slightly worse than that of Kubernetes prototype. For $T[360]$ workloads, we observe that Kubernetes prototype fails to satisfy the deadlines of certain long-duration SLO jobs, and instead leaves more resources for other jobs as a result of deadline guarantee performance improvement. For $T[720]$ workloads, the high submission density can lead

TABLE III
PERFORMANCE COMPARISONS BETWEEN SIMULATION AND KUBERNETES IMPLEMENTATION IN R^{slo} AND AVERAGE JCT OVER DIFFERENT WORKLOAD SUBMISSION DENSITIES

Job Load	$T[360]$		$T[720]$	
Metric	R^{slo} (%)	Avg JCT (min)	R^{slo} (%)	Avg JCT (min)
Simulator	4.97	266.39	13.52	253.98
Kubernetes	3.92	274.42	16.11	267.40
Relative Diff	1.08%	2.93%	2.57%	5.38%

to heavy resource contention, and the simulator can use the predicted information to make more accurate scheduling decisions. Therefore, the simulator present better deadline guarantee performance. Overall, the difference is not significant and does not alter the conclusions from simulations.

B. Simulator Evaluation

SLO Enforcement. We compare R^{slo} of UNISCHED with other baseline systems for the six workloads in Fig. 6(a). We observe that UNISCHED gives the almost best results in all the workloads. In contrast, DL schedulers are poor at guaranteeing deadlines, as their designs do not take SLO into consideration.

Deadline-aware schedulers are more effective than DL schedulers. (1) For SLO workloads, GENIE is superior to 3Sigma and Hydra, but not as good as UNISCHED due to the utilization of the preemption feature. UNISCHED obtains 1.17 - 4.82 $\times$ reduction in R^{slo} compared to these baselines over SLO workloads. (2) For both MIX1 and MIX2 workloads, the existence of best-effort jobs further reduces R^{slo} because deadline-aware schedulers can free more GPUs for SLO jobs by sacrificing best-effort jobs. In comparison to deadline-aware schedulers including 3Sigma, Hydra, and GENIE, UNISCHED attains 0.95 - 2.77 $\times$ reduction in R^{slo}. Compared to DL schedulers, the reduction of R^{slo} in UNISCHED is much higher, i.e., 2.01 - 6.84 $\times$. Particularly, UNISCHED achieves 6.84X improvement in R^{slo} compared to Themis on the P_MIX1 workload. There is no clear dominant winner among 3Sigma, Hydra, and GENIE. Additionally, GENIE cannot execute preemptive scheduling, hence its effectiveness in deadline guarantee is not satisfactory in a mixed workload scenario. (3) Compared to MIX1 workloads, UNISCHED significantly reduces R^{slo} of SLO jobs in MIX2 workloads, due to the introduction of soft deadlines.

Best-effort job performance. Fig. 6(b) displays the average JCT of best-effort jobs, normalized to that of UNISCHED. It can be observed that, in comparison to other schedulers, UNISCHED remains the most effective, and obtains 1.18 - 4.02 $\times$ reduction in latency over different workloads. It outperforms DL schedulers by 1.18-3.11 $\times$ because it has sufficient GPU resources to minimize the latency of best-effort jobs without violating the SLO requirements. Optimus can achieve shorter latency in Helios workload in that Helios trace contains a larger proportion of distributed DL jobs than Philly trace. UNISCHED reduces the latency of deadline-aware schedulers by 1.66 - 4.02 $\times$, as it seriously sacrifices these jobs to meet the requirements of more SLO jobs.

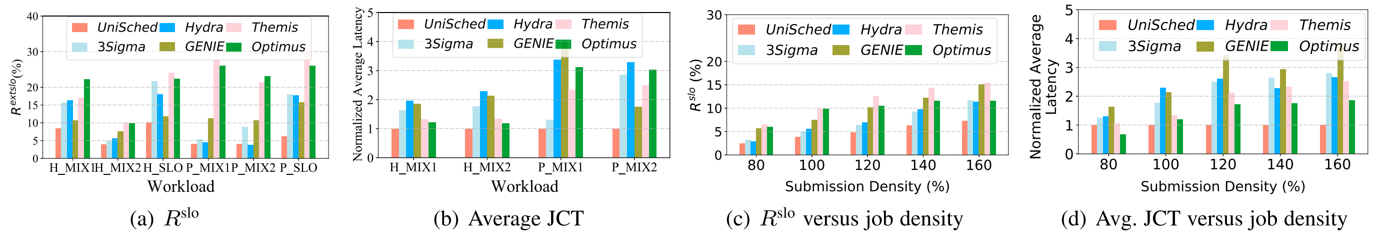


Fig. 6. Comparisons between different schedulers. UNISCHED outperforms other baselines in R^{slo} and average JCT over different workloads (a-b) and submission densities (c-d).

Impact of the job density. We evaluate the performance of various schedulers with different job densities with the H_MIX2 workload. In order to evaluate the performance of our system under various job densities, we conduct experiments where we randomly remove 20% of jobs to reduce the job density to 80%, and also inject additional jobs to increase the densities to 120%, 140%, and 160%, as described in [39]. Fig. 6(c) shows the results of SLO enforcement over different job submission densities. UNISCHED reduces R^{slo} by 1.18-2.67 $\times$ compared to other schedulers. A higher job density can increase R^{slo} of all scheduling systems, and a lower density favors the SLO enforcement of 3Sigma and GENIE. However, UNISCHED performs the best SLO enforcement in various job densities.

Fig. 6(d) shows the average JCT of best-effort jobs, normalized to that of UNISCHED. In terms of latency reduction, UNISCHED outperforms GENIE by up to 3.78 $\times$ when the submission density reaches 160%. Our UNISCHED gives the lowest JCT for most configurations. An exceptional scenario occurs when Optimus exhibits a latency that is 0.67 $\times$ that of UNISCHED at a submission density of 80%. Compared to deadline-aware schedulers, UNISCHED is able to release sufficient GPU resources for best-effort jobs without violating the requirement of SLO jobs. Compared to DL schedulers, UNISCHED can schedule best-effort jobs more effectively based on the profiling information.

Analysis of suspension and resumption. Fig. 8 shows the average numbers of suspension and resumption events over different workloads. For best-effort jobs, UNISCHED tends to allocate GPU resources to shorter jobs. Hence, these jobs are prone to renewal the leases with short remaining time. Differently, SLO jobs experience an average of 2-6 suspensions and resumptions, which is significantly higher compared to best-effort jobs. This is because UNISCHED tends to allocate GPU resources to emergent SLO jobs. As a result, when newly submitted jobs arrive, UNISCHED needs to reallocate GPUs in order to satisfy more SLO jobs. Consequently, SLO jobs experience more suspension and resumption on average across different workloads.

VI. PERFORMANCE BREAKDOWN

We first investigate the contribution of the `Estimator` and `Selector` in Section VI-A and VI-B, respectively. Then we compare between UNISCHED and CHRONUS in Section VI-C, and analyze the advantages of UNISCHED over CHRONUS.

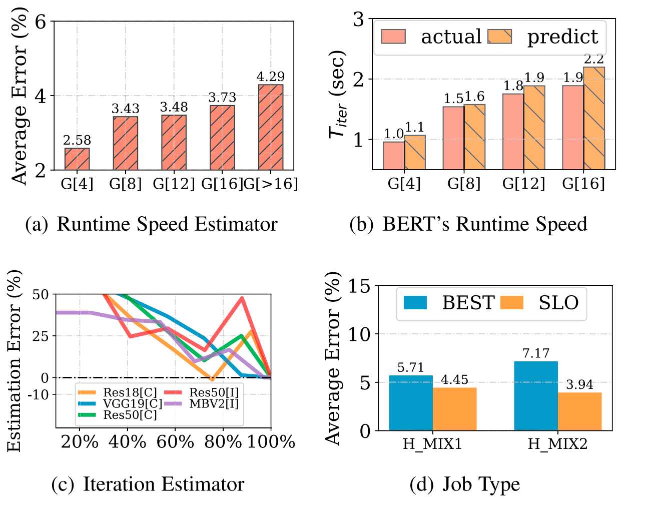


Fig. 7. Error analysis of predictor: (a) the average estimation error (y-axis) of job speed over different GPUs (x-axis); (b) the speed estimation error (y-axis) of BERT over varying GPUs (x-axis); (c) the estimation error (y-axis) of training iteration predictor over training progress (x-axis) across different tasks; (d) the estimator's estimation error on best-effort and SLO jobs.

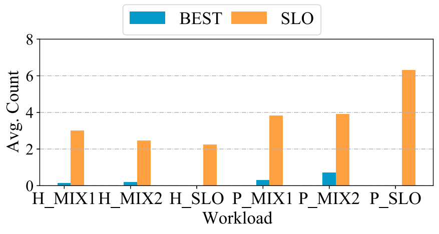
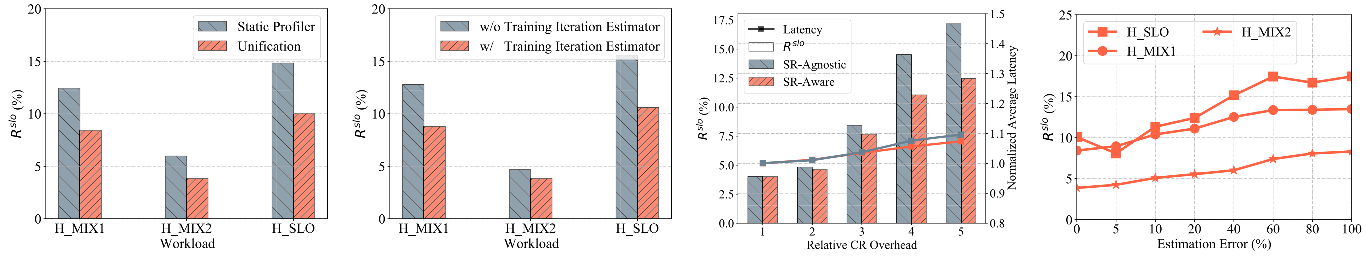


Fig. 8. Average numbers of suspension and resumption over different workloads.

A. Estimator Evaluation

We first evaluate the effectiveness of the `Estimator`, where the job runtime estimation is conducted.

Error analysis of predictor. We analyze the accuracy of the `Estimator` from different perspectives. Fig. 7(a) shows the average estimation errors of the job runtime speed (y-axis) for our evaluated DL models via profiling two GPUs over different allocated GPUs (x-axis, G[x] represents the number of allocated GPUs is x). The increase in allocated GPUs widens the gap between prediction and actual job runtime speed. The average prediction error is within 5%. Furthermore, the runtime speed



(a) Impact of Unification (b) Impact of Training Iter. Esti. (c) Impact of SR-Awareness (d) Sensitivity analysis

Fig. 9. Analysis of the `Estimator`. (a) R^{slo} comparison between the unification mechanism and static profiler; (b) the impact of the training iteration estimator on R^{slo}; (c) the impact of the SR-aware estimator on R^{slo}; (d) the impact of the estimation error on R^{slo}.

estimator performs the worst on BERT with a local batch size per GPU of 12. Fig. 7(b) compares its actual and prediction results across varied numbers of allocated GPUs. The error is up to 16.3% when 16 GPUs are assigned.

Fig. 7(c) presents the prediction error of training iteration with the increase of training progress. Note that we disable the training iteration prediction for training BERT due to a small number of epochs. The prediction error presents a decreasing trend when the `Estimator` collects more validation performance information.

Fig. 7(d) shows the Estimator's prediction performance on best-effort and SLO jobs across different Heilos traces. Considering the large estimation error of the iteration estimator at the initial stage of the training, we compare the prediction results in the middle of the training with the actual execution time. The average prediction error is still within 10%. Overall, our designed Estimator presents accurate predictions across various GPU demands, models, and job types.

Impact of unifying different types of jobs. In the profiling phase, UNISCHED uses the reward generator to schedule the profiler jobs together with the best-effort and SLO jobs in a unified way. This reward generator enables UNISCHED to make dynamic resource allocations to profiler jobs. To demonstrate its superiority, we compare UNISCHED with a system that statically allocates a fixed number of compute nodes (2 in our experiments) for job profiling. Fig. 9(a) shows R^{slo} between UNISCHED and such static profiler. We observe that UNISCHED achieves better R^{slo} compared to the static profiler. This is because UNISCHED can dynamically adjust the resource scale for profiler jobs by planning all jobs globally. Besides, our experiment suggests that UNISCHED can significantly decrease the longest pending time from 2,105 seconds to 840 seconds, so the `Estimator` can respond to the jobs promptly.

Effectiveness of the training iteration estimator. Our `Estimator` can support the performance-based stopping criterion by predicting the number of training iterations. To evaluate the effectiveness of this mechanism, we consider a baseline where the system directly executes each job with the maximal number of training iterations provided by its user. Fig. 9(b) shows R^{slo} of these jobs with and without the training iteration estimator. We observe that R^{slo} is reduced by 0.7%-5.1% when UNISCHED estimates the number of iterations. This results from that the training iteration estimator can inform the `Selector` to leverage more accurate time-resource information to satisfy the deadlines.

Effectiveness of the SR-aware estimator. We evaluate our SR-aware estimator in the `Estimator` (Section III-B3). Fig. 9(c) shows R^{slo} of SLO jobs and latency of best-effort jobs with and without the SR-aware estimator. The x-axis represents the ratio between the experimental suspending/resuming overhead and actual overhead. We manually increase the overhead and observe that our SR-estimator can effectively reduce the deadline miss rate. The SR-aware estimator can provide a more reasonable runtime estimation and lead UNISCHED to make time-dimension resource allocations more accurately.

Estimation accuracy analysis. The scheduling performance can be affected by the prediction accuracy of the `Estimator`. We perform a sensitivity analysis to evaluate this dependency. We perturb the profiled job runtime with random Gaussian noise, and present the scheduling result for different traces in Fig. 9(d). In this figure, x-axis denotes the standard deviation of the injected noise and y-axis shows R^{slo} of SLO jobs. We can see UNISCHED demonstrates strong robustness at the noise scale smaller than 40%. The `Estimator` can easily achieve this in practice.

B. Selector Evaluation

Impact of the SLO lease length. We consider how the SLO lease length could influence the deadline enforcement. Fig. 10(a) shows the JCT of best-effort jobs and R^{slo} of SLO jobs with the H_MIX1 workload. We observe that a short lease term (leq 5 minutes) can cause more frequent preemption operations with large overhead, leading to higher R^{slo} for SLO jobs. A longer lease term could also increase R^{slo} as it restricts the scheduling opportunities. Additionally, a similar experiment on the H_MIX2 workload is also conducted. The performance of R^{slo} and latency is small between 10 and 30 minutes, but the 10-minute SLO lease length still achieves the lowest R^{slo} and latency.

Effectiveness of the MILP solver. The MILP solver can effectively improve the SLO enforcement by maximizing the total reward value (Eq. 4). Here, we consider two scenarios for the MILP solver: (1) maximizing the objective subject to the constraints. (2) only finding a feasible solution to obey the constraints. Fig. 10(b) and 10(c) show R^{slo} of SLO jobs and the

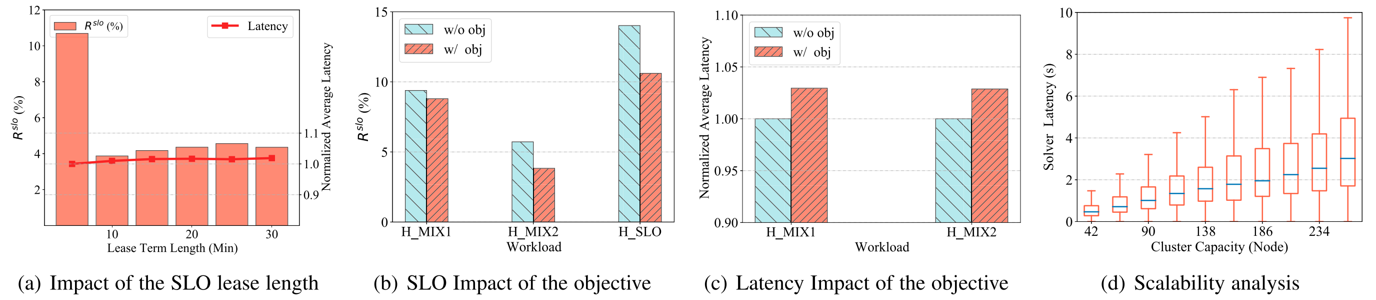

(a) Impact of the SLO lease length (b) SLO Impact of the objective (c) Latency Impact of the objective (d) Scalability analysis

Fig. 10. Performance analysis of the Selector. (a) Impact of the SLO lease length on R^{slo} and job latency; (b) impact of the objective on R^{slo}; (c) impact of the objective on the job latency; (d) impact of the cluster capacity on the MILP solver latency.

latency of best-effort jobs respectively over different workloads with and without the consideration of the objective. Our observation is that maximizing the objective can significantly reduce R^{slo} of SLO jobs, and slightly increase the latency of best-effort jobs. As we set a high reward value for SLO jobs, the scheduler sacrifices the latency of best-effort jobs to maximize the total reward value.

The latency of the MILP solver has impact on the scalability of UNISCHED. When the cluster has a larger scale and higher job submission rate, the MILP solver demands more time to find the solutions, which could possibly cause larger pending overhead and scheduling inefficiency. To evaluate this impact, we select H_MIX2 and adjust the number of jobs to be proportional to the capacity of the cluster. Fig. 10(d) shows the solver latency under different scales of clusters and jobs. We observe that the maximal latency induced by the MILP solver is less than 10 seconds, which is negligible compared to the long training time. This implies that UNISCHED demonstrates high scalability in handling

Effectiveness of joint optimization. The Selector adopts joint optimization to decide on the job selection and resource allocation simultaneously. To demonstrate its effectiveness, we compare this strategy with the consolidation placement solution adopted in CHRONUS [17]. We adjust the requested GPU amounts of some jobs in the H_MIX2 workload to get various ratios of consolidation-hostile jobs. Fig. 11(a) presents the average JCT of best-effort jobs (lines) and R^{slo} of SLO jobs (bars) respectively for the two mechanisms. We have two observations:

1) The joint optimization technique can remarkably decrease R^{slo} of SLO jobs. Without this technique, the Selector will fail to obtain a consolidation solution for certain SLO jobs. Then these jobs will be placed in the pending state, which could cause the violation of deadline requirements. When joint optimization is applied, the MILP solver will allocate appropriate cell resources to SLO jobs without violating their deadline constraint. Then R^{slo} becomes smaller.
2) The performance gap between consolidation and co-optimization techniques grows with the increase of the consolidation-hostile proportion. This demonstrates that consolidation-hostile jobs are sources to undermine performance but co-optimization can mitigate them.

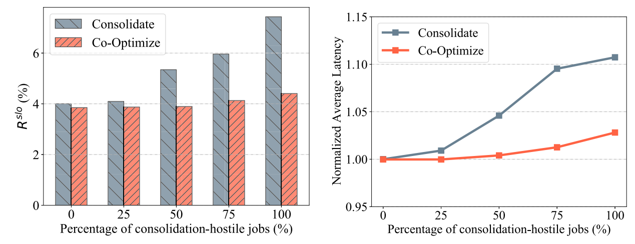
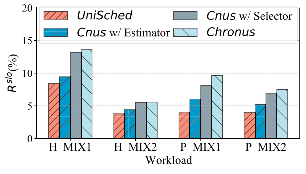

(a) R^{slo} (b) Normalized Average Latency

Fig. 11. Performance comparison between co-optimizing technique and consolidation in R^{slo} (a) and normalized average latency (b) over different percentages of consolidation-hostile jobs.

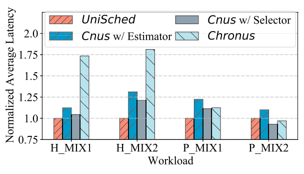
C. Comparison Between UNISCHED and CHRONUS

Since UNISCHED is improved over CHRONUS, we make a comparison between the two systems. To clearly present the performance impact of our proposed new designs, we make a detailed performance comparison between UNISCHED and CHRONUS for a mix of SLO and best-effort workloads. Our proposed Estimator still can contribute to the scenario where the cluster only accommodates SLO jobs, however, the benefit of Selector is limited in such a scenario. Fig. 12(a) shows R^{slo} of these designs for the mix workloads. We observe that UNISCHED can reduce up to 5.2% R^{slo} compared to CHRONUS. To explore the performance gap between UNISCHED and CHRONUS, we integrate the Estimator with CHRONUS to predict the job duration, especially for jobs with performance-based stopping criteria. Our observation is that the Estimator plays an important role in reducing R^{slo}: CHRONUS + Estimator gets a maximum R^{slo} reduction of 4.2% in H_MIX1 trace compared to CHRONUS. Besides, we also perform an analysis of the combination of the Selector with CHRONUS. The benefit of the Selector is not comparable to the Estimator, and the maximal R^{slo} reduction brought by the Selector is 1.5% in P_MIX1 trace compared to CHRONUS.

Fig. 12(b) presents the average JCT of UNISCHED and CHRONUS as well as other variants over the mix workloads. The reduction of the DLT job latency arises from two aspects: (1) we use the accurate job duration time estimation for best-effort jobs (Estimator), and (2) we distinguish the SLO and best-effort jobs, and it would provide more GPU resources to



(a) R^{slo}



(b) Normalized Average Latency

Fig. 12. Comparison between UNISCHED, CHRONUS w/ the Estimator, CHRONUS w/ the Selector, and CHRONUS in R^{slo} (a) and normalized average latency (b) over workloads.

best-effort jobs when SLO jobs are not emergent (Selector). We observe the Estimator improves the throughput of best-effort jobs up to $1.73\times$ compared to CHRONUS for the Helios trace. However, UNISCHED enjoys relatively moderate performance gains. Higher R^{slo} of CHRONUS also indicates that more resources are allocated to best-effort jobs. Hence, CHRONUS can even outperform UNISCHED in the P_MIX2 trace. Furthermore, early work [23] points out that the job duration distribution of Helios is more unbalanced than that of Philly. In this context, accurate job duration prediction offers notable advantages in Helios with unbalanced job duration distribution. Additionally, the Selector balances the resource allocation for SLO and best-effort jobs well, and it shows positive effects on the latency reduction over different simulated traces, and speeds up the throughput of best-effort jobs by $1.04 - 1.66\times$. Overall, our Estimator is beneficial to both SLO jobs across different workloads, offering superior SLO enforcement compared to the Selector. Additionally, it contributes to reducing latency and achieving competitive performance compared to CHRONUS in terms of latency reduction for best-effort jobs. This mainly attributes to the accurate job duration. The Selector always presents a positive impact on the latency reduction for best-effort jobs and deadline guarantee for SLO jobs.

VII. RELATED WORKS

Deadline-aware scheduling. Deadline-aware scheduling was investigated in the big data scenario, and modeled it as an MILP problem [13], [22]. However, these solutions are not tailored to DLT jobs and become less effective in GPU cluster scheduling. For example, they cannot precisely predict the duration of DLT jobs, and ignore the impacts of GPU topology and job preemption. Recent research [14], [16], [18] switched to the SLO requirements of DLT jobs. However, these solutions do not consider the mixture of SLO and best-effort jobs with different stopping criteria, which are practical in GPU clusters. Different from the above systems, UNISCHED is an end-to-end scheduler that can satisfy the scheduling goals of different jobs and support various stopping criteria.

Deep learning schedulers. Various DLT job scheduling systems have been developed to achieve different goals. Some systems focus on improving resource utilization, such as Gandiva [1] and Antman [5]. Other systems aim to boost job performance, such as Tiresias [3] and Optimus [19]. Several systems have also been proposed to maintain resource allocation fairness, such as Themis [7], $Gandiva_{fair}$ [8], and ASTRAEA [40]. However, none of these solutions are effective in SLO enforcement, which motivated us to develop UNISCHED tailored to DLT jobs.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we design and implement UNISCHED, a novel DLT scheduling system to satisfy various user demands and stopping criteria of DLT jobs. We propose innovative techniques to estimate job duration and allocate resources in an effective and efficient way. We conduct comprehensive simulations to show that UNISCHED outperforms various state-of-the-art schedulers. The prototype implementation of UNISCHED on Kubernetes further validates the practicability of our system.

We consider the following directions as future work. (1) This paper mainly considers and evaluate the homogeneous GPU clusters. It is easy to extend UNISCHED to heterogeneous clusters. A new binary variable is needed in the constraint and objective to denote the type of GPU resources. We will implement UNISCHED on heterogeneous GPU clusters in the near future. (2) Auto-scaling allows a user to specify a range of GPUs for his DLT job. To enable this flexible mechanism, several binary variables can be introduced to represent the selection of every value in that range in the MILP optimization. This may potentially incur larger search costs.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their valuable comments.

REFERENCES

[1] W. Xiao et al., "Gandiva: Introspective cluster scheduling for deep learning," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation (OSDI)*, Carlsbad, CA, USA, A. C. Arpaci-Dusseau and G. Voelker, Eds., Carlsbad, CA, USA: USENIX Association, 2018, pp. 595–610.
[2] M. Jeon et al., "Analysis of large-scale multi-tenant GPU clusters for DNN training workloads," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Renton, WA, USA, D. Malkhi and D. Tsafrir, Eds., Renton, WA, USA: USENIX Association, 2019, pp. 947–960.
[3] J. Gu et al., "Tiresias: A GPU cluster manager for distributed deep learning," in *Proc. 16th USENIX Symp. Netw. Syst. Des. Implementation (NSDI)*, Boston, MA, USA, J. R. Lorch and M. Yu, Eds., Boston, MA, USA: USENIX Association, 2019, pp. 485–500.
[4] H. Zhao et al., "HiveD: Sharing a GPU cluster for deep learning with guarantees," in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation (OSDI)*, USENIX Association, 2020, pp. 515–532.
[5] W. Xiao et al., "AntMan: Dynamic scaling on GPU clusters for deep learning," in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation (OSDI)*, USENIX Association, 2020, pp. 533–548.
[6] A. Qiao et al., "Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning," in *Proc. 15th USENIX Symp. Operating Syst. Des. Implementation (OSDI)*, A. D. Brown and J. R. Lorch, Eds., USENIX Association, 2021, pp. 1–18.
[7] K. Mahajan et al., "THEMIS: Fair and efficient GPU cluster scheduling," in *Proc. 17th USENIX Symp. Netw. Syst. Des. Implementation (NSDI)*, Santa Clara, CA, USA, R. Bhagwan and G. Porter, Eds., Santa Clara, CA, USA: USENIX Association, 2020, pp. 289–304.
[8] S. Chaudhary, R. Ramjee, M. Sivathanu, N. Kwatra, and S. Viswanatha, "Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning," in *Proc. 15th EuroSys Conf. (EuroSys '20)*, Heraklion, Greece, A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds., New York, NY, USA: ACM, 2020, pp. 1: 1–1:16.
[9] T. N. Le, X. Sun, M. Chowdhury, and Z. Liu, "AlloX: Compute allocation in hybrid clusters," in *Proc. 15th EuroSys Conf. (EuroSys '20)*,

Heraklion, Greece, A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds., New York, NY, USA: ACM, 2020, pp. 31: 1–31:16.
[10] Z. Ye et al., "ASTRAEA: A fair deep learning scheduler for multi-tenant GPU clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 11, pp. 2781–2793, Nov. 2022.
[11] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao, "Reservation-based scheduling: If you're late don't blame us!" in *Proc. ACM Symp. Cloud Comput.*, Seattle, WA, USA, E. Lazowska, D. Terry, R. H. Arpaci-Dusseau, and J. Gehrke, Eds., New York, NY, USA: ACM, 2014, pp. 2: 1–2:14.
[12] D. Li, C. Chen, J. Guan, Y. Zhang, J. Zhu, and R. Yu, "DCloud: Deadline-aware resource allocation for cloud computing jobs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 8, pp. 2248–2260, Aug. 2016.
[13] 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. 11th Eur. Conf. Comput. Syst. (EuroSys)*, London, U.K., C. Cadar, P. R. Pietzuch, K. Keeton, and R. Rodrigues, Eds., New York, NY, USA: ACM, 2016, pp. 35: 1–35:16.
[14] Z. Chen et al., "Deep learning research and development platform: Characterizing and scheduling with QoS guarantees on GPU clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 1, pp. 34–50, Jan. 2020.
[15] M. Abadi et al., "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," in *Proc. 12th USENIX Conf. Operating Syst. Des. Implementation*, Savannah, GA, USA: USENIX Association, 2016, pp. 265–283.
[16] R. Liaw et al., "HyperSched: Dynamic resource reallocation for model development on a deadline," in *Proc. ACM Symp. Cloud Comput. (SoCC)*, Santa Cruz, CA, USA. New York, NY, USA: ACM, 2019, pp. 61–73.
[17] W. Gao, Z. Ye, P. Sun, Y. Wen, and T. Zhang, "Chronus: A novel deadline-aware scheduler for deep learning training jobs," in *Proc. ACM Symp. Cloud Comput. (SoCC '21)*, Seattle, WA, USA, C. Curino, G. Koutrika, and R. Netravali, Eds., New York, NY, USA: ACM, 2021, pp. 609–623.
[18] Z. Yang, H. Wu, Y. Xu, Y. Wu, H. Zhong, and W. Zhang, "Hydra: Deadline-aware and efficiency-oriented scheduling for deep learning jobs on heterogeneous GPUs," *IEEE Trans. Comput.*, vol. 72, no. 8, pp. 2224–2236, Aug. 2023.
[19] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: An efficient dynamic resource scheduler for deep learning clusters," in *Proc. 13th EuroSys Conf. (EuroSys)*, Porto, Portugal, R. Oliveira, P. Felber, and Y. C. Hu, Eds., New York, NY, USA: ACM, 2018, pp. 3:1–3:14.
[20] T. Brown et al., "Language models are few-shot learners," in *Proc. Adv. Neural Inf. Process. Syst.*, 2020, pp. 1877–1901.
[21] T. Domhan, J. T. Springenberg, and F. Hutter, "Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves," in *Proc. 24th Int. Joint Conf. Artif. Intell. (IJCAI)*, Buenos Aires, Argentina, Q. Yang and M. J. Wooldridge, Eds., Buenos Aires, Argentina: AAAI Press, 2015, pp. 3460–3468.
[22] J. W. Park, A. Tumanov, A. H. Jiang, M. A. Kozuch, and G. R. Ganger, "3Sigma: Distribution-based cluster scheduling for runtime uncertainty," in *Proc. 13th EuroSys Conf. (EuroSys)*, Porto, Portugal, R. Oliveira, P. Felber, and Y. C. Hu, Eds., New York, NY, USA: ACM, 2018, pp. 2: 1–2:17.
[23] Q. Hu, P. Sun, S. Yan, Y. Wen, and T. Zhang, "Characterization and prediction of deep learning workloads in large-scale GPU datacenters," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, New York, NY, USA: ACM, 2021, pp. 1–15.
[24] Kubernetes Contributors. Accessed: Oct. 2022. [Online]. Available: https://kubernetes.io/
[25] U. Misra et al., "RubberBand: Cloud-based hyperparameter tuning," in *Proc. 16th Eur. Conf. Comput. Syst.,* A. Barbalace, P. Bhatotia, L. Alvisi, and C. Cadar, Eds., New York, NY, USA: ACM, 2021, pp. 327–342.
[26] Y. Luan, X. Chen, H. Zhao, Z. Yang, and Y. Dai, "Sched^2: Scheduling deep learning training via deep reinforcement learning," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Waikoloa, HI, USA, Piscataway, NJ, USA: IEEE Press, 2019, pp. 1–7.
[27] P. Zhou, X. He, S. Luo, H. Yu, and G. Sun, "JPAS: Job-progress-aware flow scheduling for deep learning clusters," *J. Netw. Comput. Appl.*, vol. 158, 2020, Art. no. 102590.
[28] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, "Tune: A research platform for distributed model selection and training," 2018, *arXiv:1807.05118*.
[29] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, Anchorage, AK, USA, A. Teredesai, V. Kumar, Y. Li, R. Rosales, E. Terzi, and G. Karypis, Eds., New York, NY, USA: ACM, 2019, pp. 2623–2631.
[30] A. Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., Canadian Institute for Advanced Research, Toronto, Canada, 2009. [Online]. Available: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf
[31] Y. You, Z. Zhang, C. Hsieh, J. Demmel, and K. Keutzer, "ImageNet training in minutes," in *Proc. 47th Int. Conf. Parallel Process. (ICPP)*, Eugene, OR, USA, New York, NY, USA: ACM, 2018, pp. 1: 1–1:10.
[32] J. Li, H. Xu, Y. Zhu, Z. Liu, C. Guo, and C. Wang, "Aryl: An elastic cluster scheduler for deep learning," 2022, *arXiv:2202.07896*.
[33] A. Li et al., "Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 1, pp. 94–110, Jan. 2020.
[34] H. Zhang, L. Stafman, A. Or, and M. J. Freedman, "SLAQ: Quality-driven scheduling for distributed machine learning," in *Proc. Symp. Cloud Comput. (SoCC)*, Santa Clara, CA, USA, New York, NY, USA: ACM, 2017, pp. 390–404.
[35] B. Burns, B. Grant, D. Oppenheimer, E. A. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *ACM Queue*, vol. 14, no. 1, pp. 70–93, 2016.
[36] "Gurobi optimization." Gurobi. Accessed: Oct. 2022. [Online]. Available: https://www.gurobi.com/
[37] "Generalized mixed integer optimization in go." GitHub. Accessed: Oct. 2022. [Online]. Available: https://github.com/mit-drl/goop
[38] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, "Heterogeneity-aware cluster scheduling policies for deep learning workloads," in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation (OSDI),* USENIX Association, 2020, pp. 481–498.
[39] H. Wang, Z. Liu, and H. Shen, "Job scheduling for large-scale machine learning clusters," in *Proc. 16th Int. Conf. Emerg. Netw. EXp. Technol. (CoNEXT '20)*, Barcelona, Spain, D. Han and A. Feldmann, Eds., New York, NY, USA: ACM, 2020, pp. 108–120.
[40] Z. Ye et al., "ASTRAEA: A fair deep learning scheduler for multi-tenant GPU clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 11, pp. 2781–2793, Nov. 2022.

Wei Gao received the B.S. degree from Beihang University, Beijing, China, in 2019. He is currently working toward the Ph.D. degree with Nanyang Technological University, Singapore. His research interests include distributed machine learning systems, cluster resource management, and workload scheduling.

Zhisheng Ye received the B.S. degree in computer science and technology from Peking University, China, in 2019. He is currently working toward the Ph.D. degree with the School of Computer Science, Peking University. His research interests include distributed systems, systems for machine learning, and resource management.

Peng Sun received the Ph.D. degree in computer science from Nanyang Technological University, Singapore. He is currently a Senior Research Scientist with the SenseTime Group Limited. Previously, he worked as a Research Engineer with Nanyang Technological University, Baidu Institute of Deep Learning, and Huawei 2012 Labs. His research interests include cloud computing, computer networking, data center, big data, and large-scale cluster computing systems for machine learning.

Tianwei Zhang (Member, IEEE) received the bachelor's degree from Peking University, in 2011, and the Ph.D. degree from Princeton University, in 2017. He is an Assistant Professor with the School of Computer Science and Engineering, Nanyang Technological University. His research interests include computer system security. He is particularly interested in security threats and defenses in machine learning systems, autonomous systems, computer architecture, and distributed systems.

Yonggang Wen (Fellow, IEEE) received the Ph.D. degree in electrical engineering and computer science from Massachusetts Institute of Technology, Cambridge, MA, USA, in 2008. He is a Professor of computer science and engineering with Nanyang Technological University, Singapore, where he has served as an Associate Dean (Research) with the College of Engineering since 2018. He serves on editorial boards for multiple transactions and journals, including IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY, *IEEE Wireless Communication Magazine*, *IEEE Communications Survey and Tutorials*, and IEEE TRANSACTIONS ON MULTIMEDIA.